INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

Bell & Howell Information and Learning
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA
800-521-0600

HARDWARE/SOFTWARE PARTITIONING UTILIZING BAYESIAN BELIEF NETWORKS

by

John Thomas Olson

A Dissertation Submitted to the Faculty of the

DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING

In Partial Fulfillment of the Requirements
For the Degree of

DOCTOR OF PHILOSOPHY

In the Graduate College

THE UNIVERSITY OF ARIZONA

2000

THE UNIVERSITY OF ARIZONA ®
GRADUATE COLLEGE

As members of the Final Examination Committee, we certify that we have read the dissertation prepared by John Thomas Olson

entitled Hardware/Software Partitioning Utilizing Bayesian Belief Networks

and recommend that it be accepted as fulfilling the dissertation requirement for the Degree of Doctor of Philosophy

Jerzy W. Rozenblit, Ph.D.	4/17/2000 Date
Ralph Martinez, Ph.D.	4/17/00 Date
Kevin M. McNeill, Ph.D.	4/17/00 Date
	Date
	Date

Final approval and acceptance of this dissertation is contingent upon the candidate's submission of the final copy of the dissertation to the Graduate College.

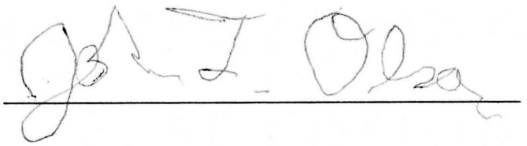
I hereby certify that I have read this dissertation prepared under my direction and recommend that it be accepted as fulfilling the dissertation requirement.

Dissertation Director Jerzy W. Rozenblit, Ph.D.	4/28/2000 Date

STATEMENT BY AUTHOR

This dissertation has been submitted in partial fulfillment of requirements for an advanced degree at The University of Arizona and is deposited in the University Library to be made available to borrowers under rules of the Library.

Brief quotations from this dissertation are allowable without special permission, provided that accurate acknowledgment of source is made. Requests for permission for extended quotation from or reproduction of this manuscript in whole or in part may be granted by the copyright holder.

SIGNED: 

ACKNOWLEDGEMENTS

This work has been supported by the National Science Foundation under grant No. 9554561 "Hardware/ Software Codesign for High Performance Systems."

I would like to thank all of my committee members:

Advisor – Major Department:	Jerzy W. Rozenblit, Ph.D.
Major Committee:	Ralph Martinez, Ph.D.
Major Committee:	Kevin McNeill, Ph.D.
Advisor – Minor Department:	Julie Higle, Ph.D.
Minor Committee:	Jeff Goldberg, Ph.D.

I would also like to thank the ECE graduate advisor, Tami Whelan.

DEDICATION

This Dissertation is dedicated to all of those who have helped me in achieving this goal in my education; especially my mother, Judy, and all of my friends.

TABLE OF CONTENTS

LIST OF FIGURES

ABSTRACT

In heterogeneous systems design, partitioning of the functional specifications into hardware and software components is an important procedure. Often, a hardware platform is chosen and the software is mapped onto the existing partial solution, or the actual partitioning is performed in an ad hoc manner. The partitioning approach presented here is novel in that it uses Bayesian Belief Networks (BBNs) to categorize functional components into hardware and software classifications. The BBN's ability to propagate evidence permits the effects of a classification decision made about one function to be felt throughout the entire network. In addition, because BBNs have a belief of hypotheses as their core, a quantitative measurement as to the correctness of a partitioning decision is achieved. In this research, the motivation and background material are presented first. Next, a methodology for automatically generating the qualitative, structural portion of BBN, and the quantitative link matrices is given. Lastly, a case study of a programmable thermostat is developed to illustrate the BBN approach. The outcomes of the partitioning process are discussed and placed in a larger design context, called model-based Codesign.

Chapter 1 INTRODUCTION

In this dissertation, we propose a new approach to the hardware/software partitioning problem [4][5][6][8][21][24] by utilizing Bayesian Belief Networks for functional component classification into hardware and software. Design of heterogeneous systems entails choosing which functional components should be implemented in hardware and which should be implemented in software. Typically, a hardware platform is chosen and the software is written to make the hardware meet the specified requirements. The problem with this approach, however, is that during system integration, interface and incompatibility problems arise late. Hardware/software partitioning is used to push the implementation decisions back so that the decision of whether to use hardware or software is not made in isolation for each functional component.

In our previous work, we have established a systematic approach to design of heterogeneous systems. Called model-based codesign [5][24], this approach uses simulatable system descriptions as the basis for the generation of design descriptions from which the real system is built. Simulation is used as a primary means of verifying functional requirements of the design. Thus, in parallel to the simulation, classifications to the system model components into hardware or software must be made.

The partitioning approach presented here uses the Bayesian Belief Network (BBN) concept [3][14][20][28] for classification of functional elements within the system model to either hardware or software. The reasons for using the BBN framework are: (1)

its aptitude to represent the causal nature of a functional description (e.g., a function, *A*, calling another function, *B*, is a causal influence from *A* to *B*) and (2) the ability to distribute local evidence (simulation results that drive a particular functional component towards a hardware or software realization) throughout the entire network and thus, make the effects of a local partitioning decision affect partitioning decisions throughout the entire model. This in conjunction with the benefit of having probabilistic measurements as to the degree of belief of classification decisions makes the use of BBNs appropriate.

In the ensuing sections, we first provide the motivation for this new type of partitioning methodology and describe the problem more formally. Afterwards, we describe the principles of BBNs and then address the existing body of hardware/software partitioning work. Finally, we present our formal methodology followed by a detailed example, and make concluding remarks.

MOTIVATION

There has been much work done in the area of hardware/software partitioning. There are three main classes of work in this area: (1) methods in which the partitioning is driven towards a pre-existing hardware platform, (2) flexible methods that use no fixed, pre-existing hardware platform, and (3) methods that are based on traditional VLSI partitioning techniques.

In the methods that drive partitioning onto a pre-existing hardware platform, the configuration usually consists of a single microprocessor to run the software component, and an Application Specific Integrated Circuit (ASIC) to implement the hardware portion. The work of [7][9][12][16][17] is among the most notable in this area of

hardware/software partitioning. The major limitation to this body of work is, of course, the inability to utilize a different hardware configuration, especially in the cases where the design and fabrication of an ASIC is not feasible because of the cost. Therefore, the need for hardware/software partitioning systems that are flexible with regards to the type and number of components comprising the hardware platform is warranted.

Several works have attempted to address the problem of partitioning onto a flexible hardware/software platform. Among them are [15][13][10][29][25][23], and [2], utilizing criticality, dynamic cost functions, hardware effort, object-oriented design, and genetic algorithm based methodologies, just to name a few. Although these works provide a necessary extension to those that assume a fixed hardware platform, they still lack a way for localized decisions regarding partitioning to be felt globally. For example, the fitness function of a genetic algorithm determines "how good" a given chromosome is, but it does not have the ability to calculate how a change in one part of the chromosome affects other parts. The ability to track how local partitioning decisions affect other components in the global system can be useful to determine the cause of problems during system integration.

The third area of related work in hardware/software partitioning uses VLSI circuit partitioning algorithms with some modifications. [1] provides an excellent review of such partitioning algorithms. [27] and [18] provide partitioning methodologies that modify min-cut (an iterative pair swap algorithm) for functional partitioning and add Petri nets to clustering methods, respectively. With these types of partitioning approaches, a cost function is often used and the choice of which partition will be chosen

for a component is based on what will minimize the cost the greatest amount. Again, there is no direct way to measure the effect a partitioning choice has on the other components individually, just the effect on the overall cost. A detailed explanation regarding how the partitioning methods just mentioned is given in the Background Chapter.

The examination of the previous work in the area of hardware/software partitioning has lead to the following conclusion; there is a need for a hardware/software partitioning methodology that allows for partitioning onto a non-fixed hardware platform that also allows local partitioning decisions to affect of system components in a global manner.

PROBLEM FORMULATION

In the design of heterogeneous systems, the choice of how to implement the system architecture can make significant differences in performance and reliability. In the past, a hardware platform was often chosen and then software was written for correcting the inadequacies of the hardware. Currently, however, research has progressed from the idea of partitioning hardware elements within a VLSI design, to that of partitioning a high level functional model of a system. Figure 1 shows an example partitioning into hardware and software, with the system model containing four functional components (A, B, C, D) that are partitioned into hardware (A, C, D) and software (B).

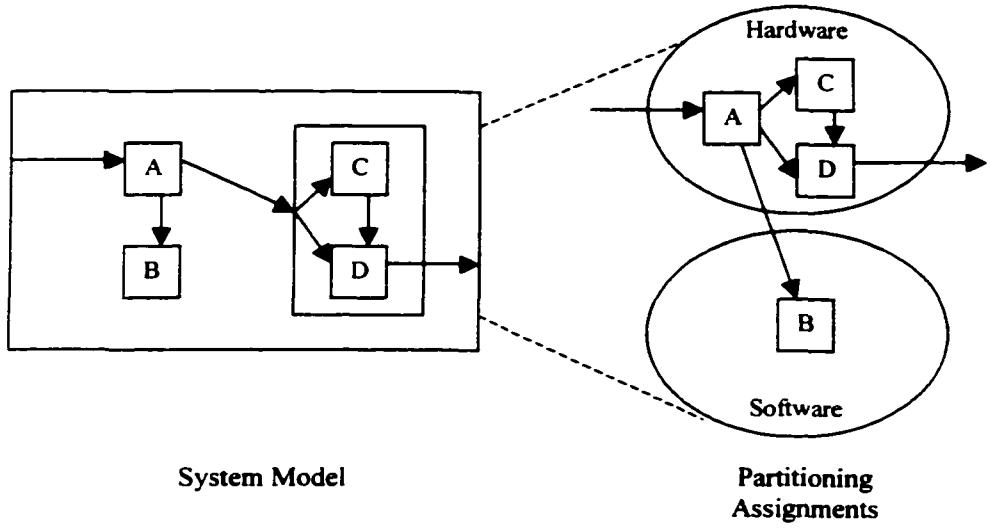


Figure 1: An illustration of the partitioning problem. The labels A, B, C, and D represent functional elements.

The hardware/software partitioning methodology we present here is part of a larger design context called model-based codesign [5][24]. In model-based codesign, a set of requirements and specifications are obtained for the system to be modeled. The system is then described as an abstract model that is a combination of its structural and behavioral specifications. Model components are specified at a high level of abstraction to remain technology independent. The modeling process includes a stepwise refinement of specifications to a desired level of granularity. Then, simulation studies are carried out to gain introspection into how well the model-based specifications meet the system's requirements. At the end of the simulation process, a virtual system's prototype is obtained. Figure 2 shows the methodology for model-based codesign with the area that includes hardware/software partitioning enclosed with a dashed rectangle.

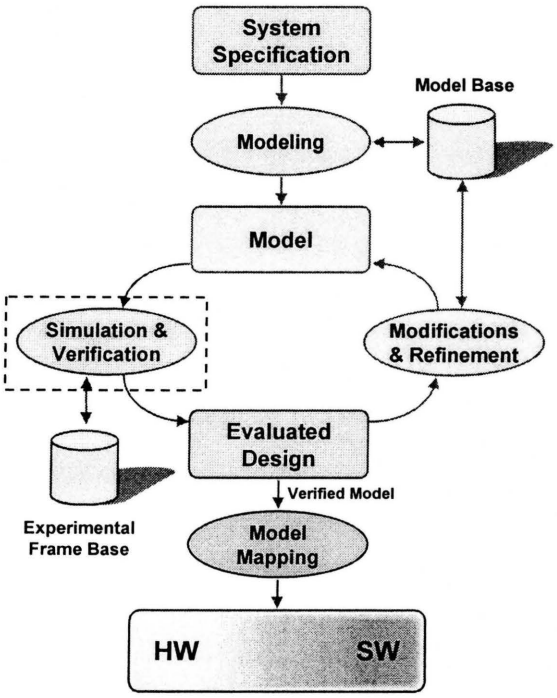


Figure 2: Methodology for model-based codesign with the portion containing the hardware/software partitioning surrounded by a dashed box.

The partitioning methodology presented here takes the high-level functional description along with parameters obtained from simulating the functional model with a tool such as StateMate [11] to create the desired BBN representation. The results from the simulation experiments of the components classified thus far are then used as evidence and propagated throughout the BBN. The hardware and software functional classifications chosen by the BBN framework are mapped into specific hardware and software components. At this point, the abstract model is mapped onto a collection of interconnected, real world components.

The contributions of this dissertation are the following:

- A hardware/software partitioning methodology capable of:

- Partitioning onto a hardware platform that is determined dynamically and not fixed prior to partitioning
- Extending partitioning decisions to apply to the entire system

- Generation of partitioning representation based on Bayesian Belief Networks

Chapter 2 BACKGROUND

BAYESIAN BELIEF NETWORKS

OVERVIEW

In order to fully understand the complexities of a Bayesian Belief Network (BBN), one must first contemplate the underlying mode of thinking involved. In classical expert systems and other knowledge-based systems, the key role of the tool being used is the inference of new knowledge from pre-existing knowledge [22]. In other words, use what is already known about the state of a domain (the facts) to infer (via an inference engine) new knowledge by utilizing rules that describe the domain itself and how facts can be combined (the rule base). Thus, the mode of thinking is, "Given my current knowledge of the domain, what else can I infer as true or false?"

The purpose of BBNs, on the other hand, is to look at the world from a causal point of view. The key role of the tool has changed from that of inferring new knowledge (as is the case with rule-based systems) to one of cause and effect. The causal nature of the domain is captured in the BBN, and the knowledge of the state of the domain (the evidence) is used to confirm a hypothesis with a certain degree of probability. The mode of thinking is therefore, "Given my current knowledge of the domain, what could have caused these facts?"

In the sections that follow, key definitions of the components of a BBN are given. Afterwards, the concepts of Bayes' rule, conditional independence, and joint probability distributions will be explained.

BBN COMPONENTS

A Bayesian Belief Network (BBN) is a directed acyclic graph, representing the causal nature of a problem domain [3][14][20][28]. A BBN is composed of two parts: (1) the graphical representation showing the causal relationship between nodes (the qualitative part), and (2) the conditional matrices associated with each link and the equations that govern the propagation of evidence (the quantitative part). Together, these two parts provide a system capable of translating evidence pertaining to measured results, into the probability that a given hypothesis is true.

In the qualitative portion of a BBN, a directed arc from any node A to another node B (denoted $A \rightarrow B$) denotes the causal influence of A over B. Each node within a BBN represents a statistical random variable, which may comprise of several hypotheses. Therefore, evidence related to the truth of B can be converted into the probability that a hypothesis in A is the cause of B. The true power of the qualitative portion of a BBN lies in the graphical nature in which it is represented. Someone with little experience in the area of probablistic reasoning can easily understand the causal relationships among the nodes, in Figure 3 it is easy to see that node A has a causal influence over nodes B and C.

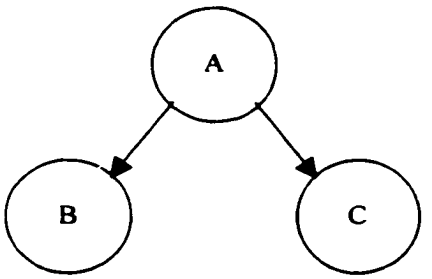


Figure 3: A simple BBN showing that A has a causal influence over B and C.

The quantitative portion of a BBN uses the qualitative part by determining in which direction evidence and causal messages travel throughout the network when distributing probablistic evidence within a BBN. Evidence is a probablistic measure pertaining to the degree of belief for all hypotheses within a given node (random variable) in the BBN. Two types of messages are used: (1) evidence messages carry the effects of newly introduced evidence, and (2) causal messages carry the effects of causal influences. Evidence messages travel against the direction of the arc in the form of λ messages. The causal messages travel with the direction of the arc in the form of π messages. The combination of these two types of messages, along with the prior probabilities and link matrices are used to determine the beliefs associated with each node of the graph. The prior probabilities give the hypothetical beliefs for each node before any evidences have been introduced (usually set to equal probability). The link matrices represent conditional probabilities of choosing a hypothesis given that the values of the hypotheses of a node acting as a causal influence are already known. Using Figure 3, given that we know something about node A, the link matrices would translate that knowledge into information that can be used by node B or C. For the link between A and

B, the link matrix shown Figure 4 in gives the conditional probabilities that comprise the entries of the matrix. For example, the entry in position (1,1) represents the probability that element B should be implemented in hardware given that A is implemented in hardware.

$$\begin{array}{c} A \rightarrow B \\ \\ \\ A \end{array} \begin{array}{c} \\ B \\ \begin{bmatrix} P(B_{HW} \mid A_{HW}) & P(B_{SW} \mid A_{HW}) \\ P(B_{HW} \mid A_{SW}) & P(B_{SW} \mid A_{SW}) \end{bmatrix} \end{array}$$

Figure 4: Definition for the entries in a link matrix.

All of the messages that interact with the link matrices and the directions in which they flow are shown in Figure 5.

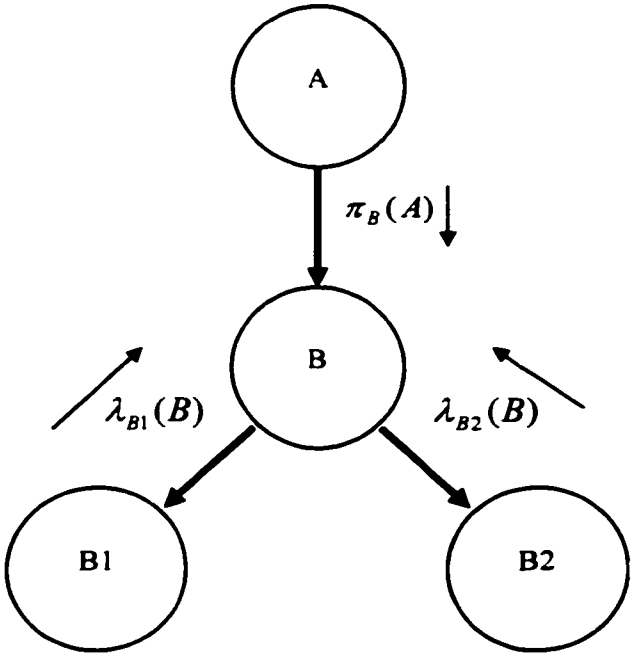


Figure 5: A small BBN with the messages passed to node B illustrated.

Before a detailed explanation of how the belief of an individual node's hypotheses can be calculated, more background into Bayesian theory and probability is necessary.

BAYESIAN THEORY AND OTHER TOOLS

The ability for BBNs to work lies in Bayes rule which is grounded in presenting an easier way to calculate conditional probabilities. The definition of a conditional probability is the following: the probability of event 'A' occurring given that event 'B' has occurred, denoted P[A|B], is equal to the probability of 'A' intersected with the probility of 'B' divided by the probability of 'B,' or, P[AB]/P[B]. Hence, P[A|B] = P[AB]/P[B]. Often, however, it is difficult to find P[AB], which leads us to Bayes rule. What Bayes rule does is powerful, yet also very simple; it allows us to find P[A|B] in terms of P[B|A], which is often easier to calculate than P[AB]. Bayes rule is defined as follows (using the same notation as before): P[A|B] = P[B|A]*P[A]/P[B], another common form is, P[B]*P[A|B] = P[B|A]*P[A]. With Bayes rule defined, we can move onto the ideas of conditional independence and d-separation.

There are several ways to define independence of events. We choose the following because of its relation to conditional probability: given two independent events 'A' and 'B', P[A|B] = P[A], meaning that the probability of 'A' given 'B' does not depend at all on P[B]. Conditional independence is defined as the following: if 'A' and 'B' are conditionally independent given 'e', then P[A|B,e] = P[A|e]; i.e., if a way can be found to show that 'A' and 'B' are conditionally independent given 'e', then we don't have to include 'B' in the calculations. The joint probability distribution represents all combinations of all assignments to the random variables that define a BBN. So all that

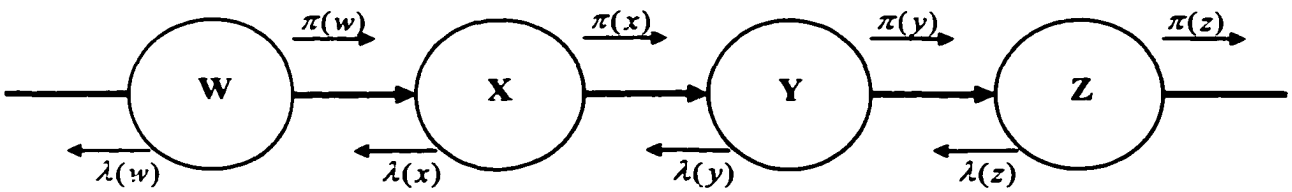
needs to be done to calculate any needed variable, in a network with 'n' nodes, is perform 2^n calculations[20]! Since this task grows very large very quckly, the ability to know when nodes are conditionally independent would do a great service in reducing this task. A method of figuring conditional independence can be performed graphically and is called d-seperation [20].

Since the definition of d-separation is somewhat complicated, knowing what it can mean to the calculation of probabilities gives us reason to show it. According to [22], if every undirected path from node 'X' to node 'Y' is d-separated by 'E', then 'X' and 'Y' are conditionally independent given 'E'. Thus, d-separation helps to show conditional independence between two given nodes in a BBN. Let us define 'E' as a portion of the BBN, where the probabilities of the nodes within 'E' have already been calculated. Therefore, the d-separation of two nodes, {'X' , 'Y'} $\in B$ (where B is defined as a BBN), and 'X' $\neq$'Y', is defined as the condition in which every path between 'X' and 'Y' goes through a node 'Z' ('Z' $\in B$, and 'Z' $\neq$'X' and 'Z' $\neq$'Y') where 'Z' falls into one of the 3 categories:

1. 'Z' lies within 'E' and 'Z' has one arrow in and one arrow out of 'E', i.e., part of a chain.
2. 'Z' lies within 'E' and has both arrows out, i.e., diverging.
3. 'Z' nor any of its descendents are in 'E' and both arrows point in, i.e., converging.

EVIDENCE PROPAGATION

The final step in understanding BBNs is knowing how the belief of each node is calculated. The belief of a node is actually the belief in each hypothesis within the node. Figure 6 shows the simplest kind of BBN structure, a chain, along with the equations that govern the calculation of belief. Note that the term, $M_{y|x}$, represents the link matrix between 'X' and 'Y' which is a conditional matrix with each value representing the probability of an event in 'Y' given the corresponding event in 'X.'



$$BEL(x) \equiv \alpha\, \lambda(x)\pi(x)$$
$$\lambda(x) = M_{y|x} \bullet \lambda(y)$$
$$\pi(x) = \pi(w) \bullet M_{x|w}$$

Figure 6: Actual calculations that must be made to determine the Belief of X.

In the propagation of evidence within trees, it is best to illustrate with a figure the process of evidence propagation and belief updating. Figure 7 shows the introduction of evidence and how that evidence propagation spreads throughout the tree structure. Note that the propagation of evidence (shown by the update messages) to nodes 'A' and 'B1' uses the principles of d-separation. Because evidence has already been propagated to node 'B' (in part a of Figure 7) node 'B' acts as condition 1 for propagation to 'A' and as condition 2 from the definition of d-separation. Therefore equations like those shown in Figure 6 can be used to calculate the new belief values for 'A' and 'B1'.

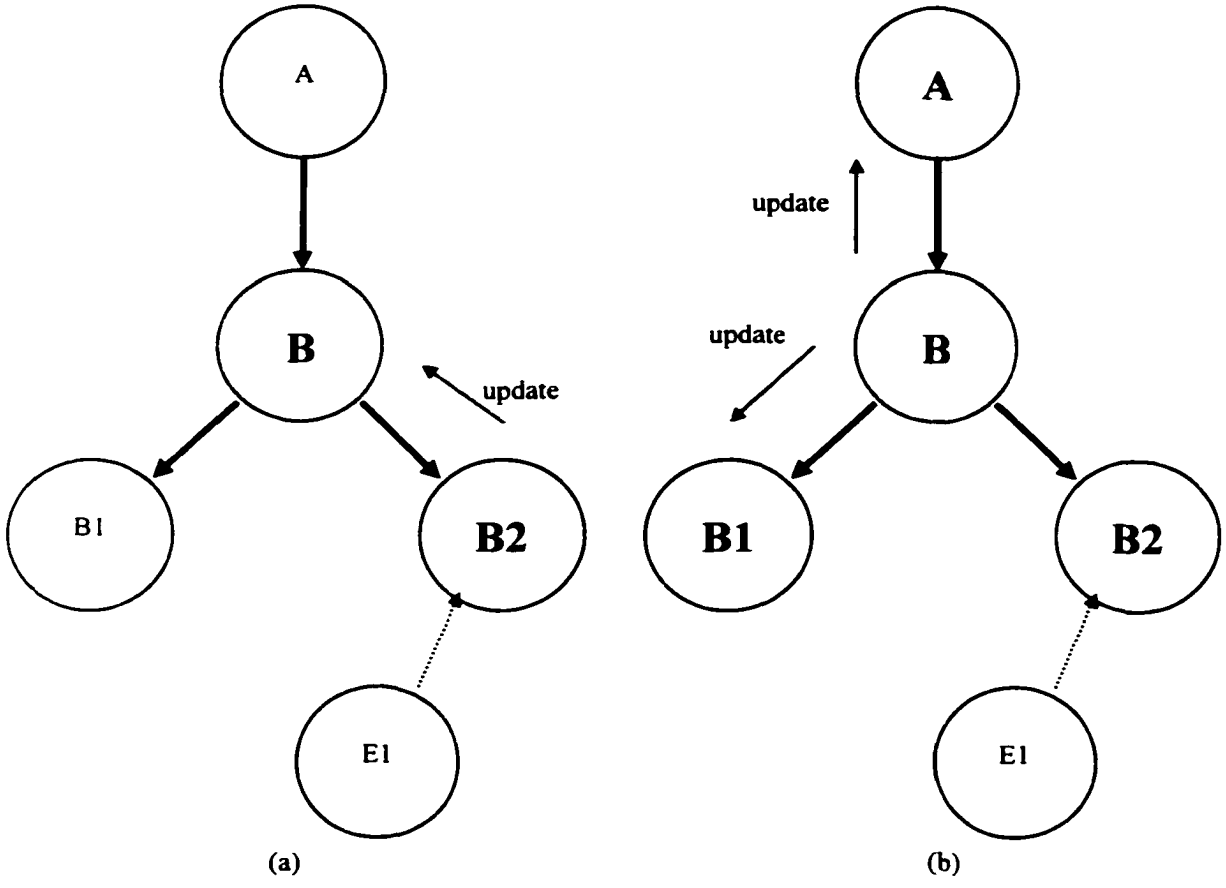


Figure 7: Propagation of evidence from its introduction (shown in a) to its propagation to the rest of the network (shown in b). Nodes shown in bold represent that their beliefs have been updated.

OVERVIEW OF PARTITIONING METHODS

FIXED HARDWARE PLATFORMS

Many previous works present partitioning algorithms that assume a base architecture and then partition according to that architecture. The work of [12] partitions a solution into a software component that runs on a single processor and a hardware component that consists of either an ASIC or FPGA in a system called COSYN [12].

The input to this system is a subset of C with extensions for multiple processes, inter-process communication, and timing constraints. From the input, a set of basic blocks is generated that is then mapped to a colored interpreted Petri net (CIPN). The model is then split into three pieces (behavior, interface, and communication), then simulation is started by placing tokens at the start of each process. Simulation continues until the set of generated transitions is empty. Partitioning begins with a completely software solution and then iteratively moves nodes into a hardware solution utilizing Activation frequency and token residence time along with hardware and software estimators to determine the amount of speedup from moving a given functional block from software to hardware.

The work of [7] is similar in that partitioning occurs to a single microprocessor based software component and a hardware co-processor. In the first step of the partitioning algorithm, time intensive processes are grouped, extracted, and replaced with a new, single process. Next a process graph is produced with weights on both nodes and arcs where the weights represent suitability for hardware implementation and inter-process communication and synchronization, respectively. The process graph is then partitioned using simulated annealing, or "tabu" search where a list of previously explored, or "tabu" nodes are kept to reduce cycling.

The SHAPES environment [17] again uses a single microprocessor paired with an ASIC utilizing a shared memory for a target architecture. The input to the partitioning methodology is an execution flow graph where every node is labeled with information regarding the hardware area, hardware and software execution time, and the average task execution frequency. Edges are also labeled with a communication value. A multi-step

partitioning algorithm is used with the first step consisting of using a rule based system for an initial partitioning. Next, the solution is evaluated using the parameters: estimation of hardware area, final execution time, and the final communication cost. If a partition is not feasible, then re-partitioning takes place based on migration from hardware to software (or vice versa) or based on inter-module communication.

[16] provides a very realistic model of communication including synchronization delays as part of a larger partitioning algorithm called PACE, which is part of the LYCOS system. This work extends the "regular" PACE by using communication parameters (protocol used, area of drivers and frequency of component execution) between any two communicating processors to partition hardware and software. This work also extends the partitioning algorithm by accounting for the hardware area taken by the communication drivers. A more detailed explanation of the actual partitioning methodology is given in the related work below.

[9] presents another application related to LYCOS in which pre-allocation of hardware in the data path is used as a step before hardware/software partitioning. Components within the system are modeled as BSBs (basic scheduling blocks), and the ability to balance the number of BSBs that are placed in hardware with the speedup attained by each individual BSB placed into hardware is the essential problem that is solved. In the main algorithm of the solution the array of BSBs is first prioritized. Following this, the main loop is executed where it first checks if the highest priority BSB is already in hardware, and if so, checks to make sure that it is the most urgent resource and that it meets all allocation constraints. If it is not already in hardware, a check is

made to determine if all of the required resources for the BSB will fit in the hardware area, and if so, it is placed into hardware. If a new BSB has been allocated to hardware then the remaining BSBs are again prioritized and the loop is repeated. The major problem with the whole LYCOS system is that it again partitions into a specific hardware realization consisting of a microprocessor running the software component and an ASIC for the software component.

FLEXIBLE HARDWARE PLATFORMS

The work of [15] differs from that already presented previously in that the partitioning assignment is not limited to a specific set of predetermined hardware. In [15], the input is assumed to be a set of applications, only one of which is active at any given time. The first step in this partitioning methodology is to first find commonality between the nodes within the set of applications. Metrics used to find a measurement of commonality between nodes include tagging each node with a type, using node repetition, performance/area ratio, urgency, and concurrency. Two methods of partitioning are then presented. The first algorithm uses high node repetition and high performance/area ratio as conditions to bias nodes toward hardware. The second method first calculates each application's time criticality and orders the nodes within the applications with highest criticality first. Then, each node is examined in order and if a similar node from a preceding application was placed in hardware, then this node is biased toward hardware. If the preceding similar node was placed in software, then this node is biased toward software. If there is no preceding similar node, then the performance/area ratio is used to partition. This work showed that ordering nodes

according to criticality produces a smaller in area design than clustering repeated nodes into hardware realizations.

[13] presents a unique method for high level estimation for hardware/software partitioning. In this work, the effort is placed on two high-level estimation techniques: hardware effort and hardware/software communication estimation. A control and data flow graph representation that is derived from a C system level description is used to derive a set of modules, registers, multi-plexers, and the control unit. The overall hardware effort is then calculated by adding the hardware efforts of each component listed above. The approach used in the hardware/software communication estimation assumes that software is running on a single processor core and that it must stop and send information to the hardware components when needed. Therefore, the overhead in cost comes from data being sent from software to shared memory and then from shared memory to hardware. All of the estimation techniques are combined into a dynamically weighted cost function where the constant attached to the hardware area that increases as the difference between the system time and the constraint time decreases. In this way, the hardware area component becomes more important as the system time approaches its maximum allowable value.

[10] compiles a HardwareC description to produce graphs similar to data-flow graphs where vertices represent operations, and edges either data or sequencing dependency between vertices. The first step of partitioning is to partition according to points of non-determinism; external points of non-determinism (caused by external input/output operations) are assigned to hardware, and internal points of non-determinism

(caused by internal, data-dependant operations) are assigned to software. Following the initial partitioning (assuming feasibility), operations are migrated from hardware to software in search of a lower cost partition.

The work of [29] uses Object-Oriented techniques for partitioning. [29] begins with a method dataflow graph, execution rates, and a library of available components. Initially, all methods and variables associated with an object are greedily assigned to their own processing elements that is as fast as the fastest method contained within the object needs to be. The cost is then iteratively reduced by either attempting to find a lower cost processing element for a given object, by removing methods from a processing element until all methods can be merged into other existing processing elements and thus removing the need for it, or methods are moved to better balance the system load. The next step is to find the cheapest communication channel for each data path between processing elements. Finally communication channels are allocated and devices are allocated.

Both [25] and [23] use genetic algorithms for hardware/software partitioning. [25] uses task graphs to create tables of estimation times containing entries for software and all available types of hardware implementation. The work of [23] utilizes similar representations for chromosomes as [25] and employs a fitness function based on cost, timing constraints, and concurrency constraints.

More specifically, the work of [25] has as the input to the system a task graph. The first step of the methodology is to perform software and hardware estimation techniques on the individual tasks. These estimation techniques are then used to create a

table of possible hardware and software solutions. The entries within the chromosomes for the GA contain either pointers to entries in the hardware performance table, or NULL indicating a software solution. The fitness function for this GA solution depends on hardware area and execution time constraints.

In the work of [23] the hardware/software partitioning problem is first modeled as a constraint satisfaction problem. Three main types of constraints are used: cost, timing, and concurrency. Because all three constraints must be met simultaneously the solution is an n-tuple in an n-dimensional search space, hence the need for genetic algorithms. The chromosomes used in the GA implementation use binary strings to encode which solution has been assigned to each functional element. The main algorithm starts with a random population and then iteratively chooses two parents, performs crossover and mutation, and finally calculates the fitness of the resulting chromosome. The fitness function used considers a cost penalty, a time penalty, and a concurrency penalty that are evaluated by an associated Control Data Flow Graph (CDFG).

In the work of [2] the first major phase involves subdividing the system into components. After this, a pre-partitioning is carried out to identify which components must be in hardware or software or neither (codesign components). Characterization of the codesign modules using a fuzzy performance estimator that estimates fundamental parameters using VHDL or C descriptions. Finally a decision-making tool is used to partition the system, then synthesization and simulation are performed.

VLSI BASED ALGORITHMS

[1] provides an excellent review of the major classes of partitioning algorithms for VLSI circuit design, but these techniques also apply to any system in which components are grouped and whose inter-group communication must be kept to a minimum. Among these classes of partitioning algorithms are the following: (1) move-based approaches such as greedy and iteritive exchange algorithms, (2) geometric approaches such as vector partitioning, (3) combinatorial approaches such as max-flow min-cut, and (4) clustering-based approaches [1]. In addition, other researchers have augmented the general algorithms.

For example, Vahid [27] modified the min-cut algorithm for functional partitioning by replacing 'cut' with execution time. The first step of partitioning in [27] is to replace the input program (typically in C or VHDL) with a System-Level Intermediate Format Access Graph (SLIF). Each node represents a functional object, such as a subroutine or global variable, and each edge represents an access by one object to another. Each node is then annotated with estimates of internal computation time and sizes for hardware and software implementation. The modified min-cut procedure (replacing cut with execution time) is then carried out where an element from an optimized array is added to one of the partitions and then the array is re-sorted applying gains by neighbors of the placed element.

[18] also modifies a VLSI partitioning algorithm by adding timed Petri nets to partition using a clustering method that groups according to load balancing, inter-processor communication, and precedence relations between processes. The partitioning

begins with applying a set of transformation rules to the occam input to create a set of communicating simple processes that are then translated to a timed Petri net. Next, a set of alternative implementations is created, and one is chosen based on the degree of parallelism or by choice of the designer. Clustering is then performed which groups processes with highest inter-process communication cost, well balanced processes, and processes with highest degree of causal precedence. One process within each cluster is assigned to a processor. The cost function then is calculated for each cluster and the clusters are mapped to processors.

OVERVIEW OF STATEMATE

StateMate [11] is a state space CAD tool that is used for specifying and simulating reactive systems. Statecharts provide an extension to finite state machine models by allowing hierarchy, concurrency, and communication. The basic element in a Statechart is a state, and transitions between states are controlled by conditions and events. Scheduling of events and controlling timing of communication between different Starecharts is also possible. StateMate [11] and the use of Statecharts is important in partitioning research because it allows a method of specifying a functional model independent of any associations with hardware or software.

Chapter 3 PARTITIONING METHODOLOGY

As stated in the Introduction chapter, the hardware/software partitioning methodology presented in this dissertation is part of a larger design context known as hardware/software codesign. In the sections that follow, an overview of codesign will be given, followed by the partitioning methodology.

HARDWARE/SOFTWARE CODESIGN

In traditional hardware and software design, decisions regarding partitioning the hardware and software occur at the beginning (as in Figure 8) of the design loop and are therefore designed separately and later integrated. The difficulty with this type of design scenario is that there are often compatibility and timing problems that are usually encountered during integration and testing. It has been shown that the earlier design problems are found, the lower the overall cost is to fix them [6].

In hardware/software codesign, however, the system to be designed is modeled at a high level of abstraction and partitioning is pushed until as late as possible (as shown in Figure 9). Because of this, there are fewer integration problems, and the ways in which hardware and software interact in the final solution is better known. By knowing software/hardware interaction, the design space can be better optimized in terms of hardware area, which in turn helps with space-constrained, embedded systems.

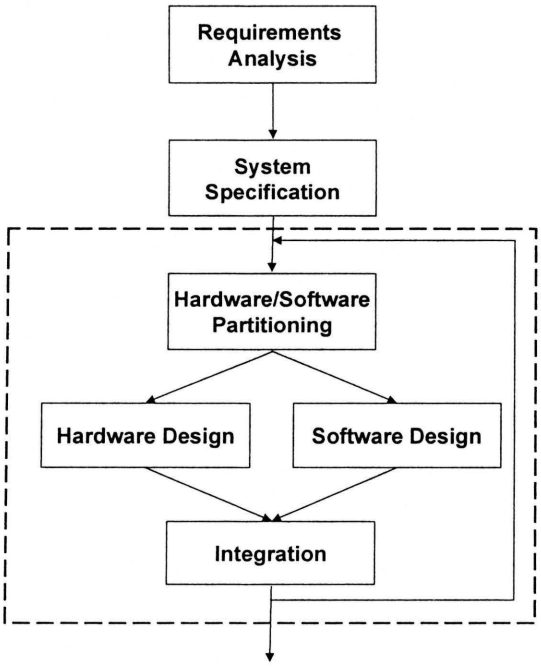


Figure 8: Traditional hardware/software design methodology.

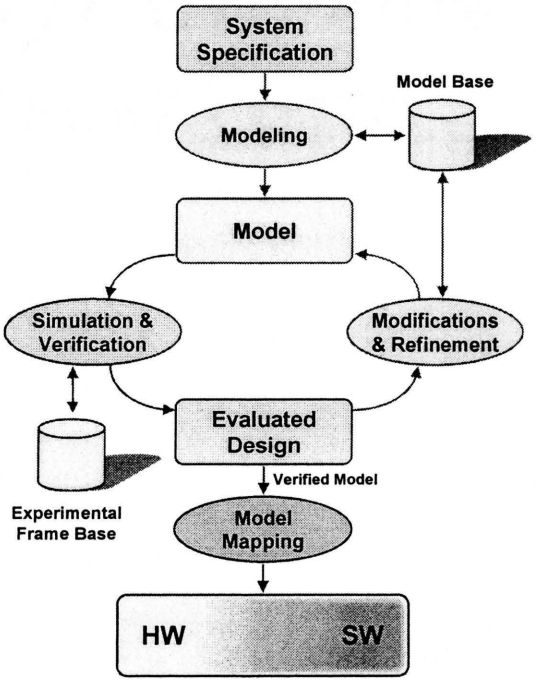


Figure 9: Hardware/software codesign methodology.

HARDWARE/SOFTWARE PARTITIONING METHODOLOGY

The hardware/software partitioning methodology presented here takes an executable functional model (including design characteristics of the functional components) and produces a partitioned model (shown in Figure 10).

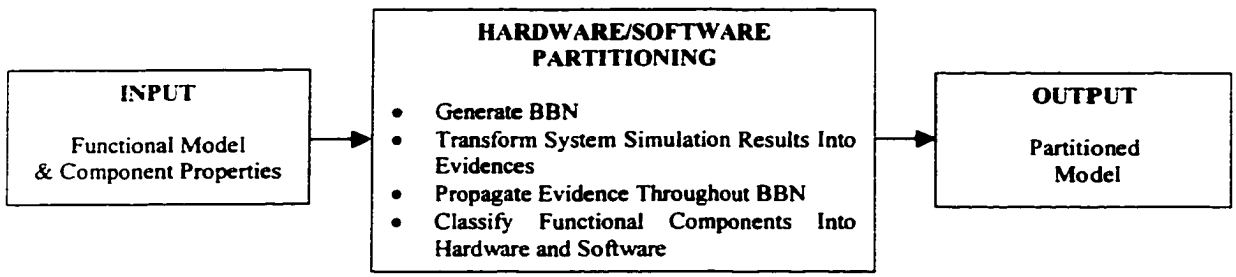


Figure 10: Major phases of BBN driven hardware/software partitioning methodology.

There are four basic steps of the partitioning methodology: (1) generation of the BBN (the subject of Chapter 4), (2) transformation of the results from simulation of the current state of the model into evidence, (3) propagation of the evidence throughout the BBN (as first described by [20] and shown in Figure 7), and (4) classification of each functional component into hardware or software, if possible. The decision of whether to classify a functional component into hardware or software can be made based on the degree of belief for each assignment as given in the belief probabilities associated in each node of the BBN.

In the first step of the methodology, generation of the BBN representation, a functional model is simulated to determine values for the complexity, bandwidth, and frequency of execution for each functional component. The hierarchical structure of the functional model is mapped into flat, BBN structure. The values for complexity,

bandwidth, and frequency are combined to construct the probability matrix associated with each causal link.

Once the representative BBN has been constructed, evidence is propagated throughout the network that is output from the simulation of the model. The model used, however, incorporates the latest belief values; i.e., if the belief that a functional component is 75% in agreement with a hardware solution, then that component is modeled as a piece of hardware in the simulation. The initial values for the beliefs associated with each node of the BBN are set to 50% hardware and 50% software. As evidence is introduced, the beliefs change according to the rules of evidence propagation as set forth by [20].

When there is no more evidence to introduce to the BBN (i.e., simulation does not produce any new results to be added as evidence), the beliefs associated with each node are examined. If there is a clear indication that either hardware or software should be used for the component (a belief having a value greater than 0.7, for example), then it is assigned to the appropriate partition. In the case where there is no clear decision that can be made by the belief values (such as when the difference between hardware belief and software belief is < 0.2), then it doesn't matter which partition is chosen. In this case, the designer should intervene and make the decision based on other criteria such as area available on the circuit board.

Chapter 4 GENERATION OF BBNs

HIERARCHICAL STRUCTURE AND BBN INDEPENDENCE

Figure 11 shows a typical non-hierarchical functional model with its corresponding BBN representation. The functional model representation we have chosen is similar to a StateChart[11]. Here we see that since the functional model is given in a single level of abstraction, the structure of the BBN is a one to one correspondence to that of the functional model. Arrows in the functional model of Figure 11 represent coupling constraints. In the BBN of Figure 11, these coupling constraints are interpreted as causal links, and therefore the arrow between any pair of nodes within the BBN is in the same direction as the corresponding pair of entities within the functional model.

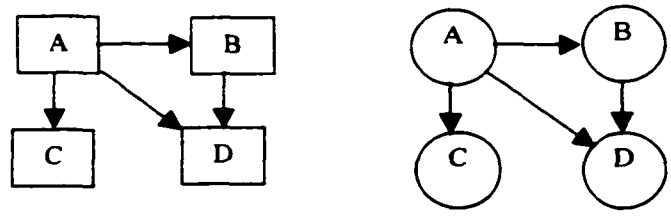


Figure 11: A typical, non-hierarchical functional model with corresponding BBN.

In choosing the BBN representation of an abstract level within a functional model, the meaning of how interactions with the abstract level as a whole affects the components within the level was very important. Figure 12 shows a level of abstraction within a functional model, and the corresponding BBN. Note that a node has been added to represent the encapsulating level (in this case, *A*) and that causal links have been added

to each of the sub-components. These causal links represent the effect of interactions between the abstraction level and the outside world. If information is passed to the level boundary (*A*), then it is appropriate that the level boundary would transmit this information through causal messages to the sub-components.

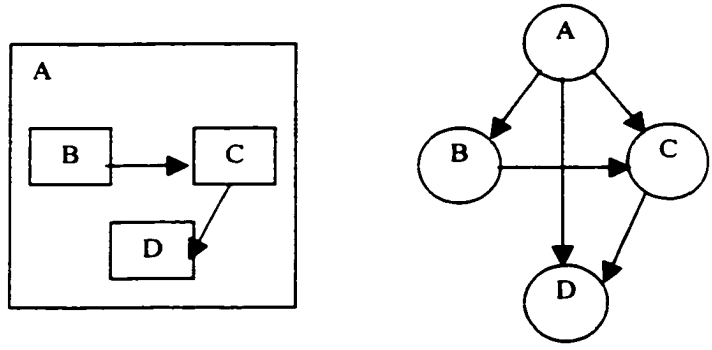


Figure 12: A typical layer of abstraction with corresponding BBN representation.

Figure 13 shows how components outside of a level of abstraction can influence both the abstract level as a whole, and individual sub-components. Note that the output from an abstract level can act as input to another component (e.g., the output of *A* is connected to *E*). Figure 13 also shows how an outside component (*H*) can have causal influence over an entire abstract level (*A*), and over an individual sub-component (*G*).

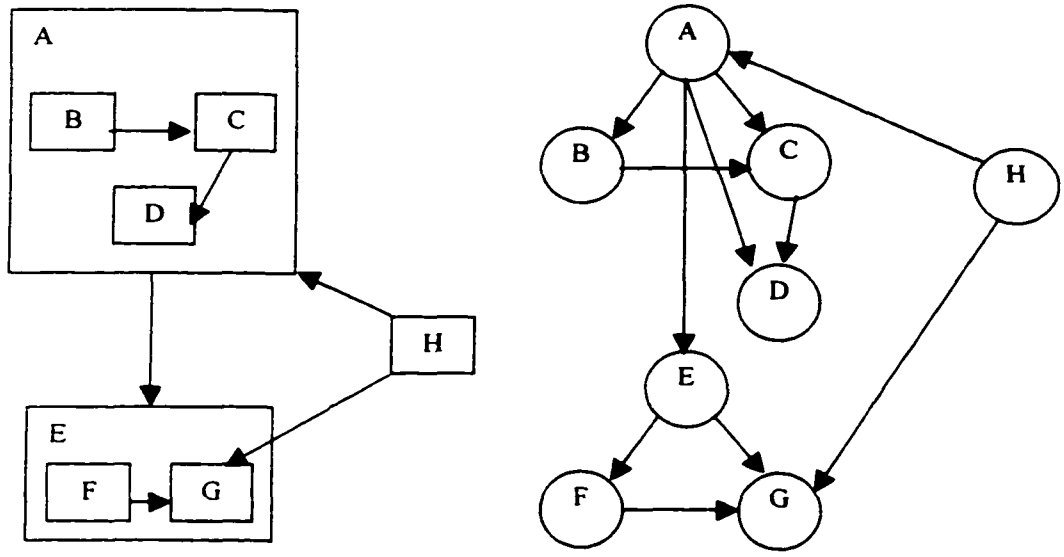


Figure 13: Example functional model containing causal links crossing abstraction boundaries (left), and corresponding BBN representation.

GENERATION OF BBN STRUCTURES

The generation of a BBN structure involves taking the functional_model in the form of a set of func_nodes (both functional_model and func_node are defined below), and converting it first into a hierarchy_tree (defined below). After conversion into a hierarchy_tree, the set of vertices and edges of the BBN are then generated. The following definitions will be of use in the presentation of the algorithm that follows.

Def: vertex – is an element that helps to describe the structure of a graph or tree and is composed of a string which uniquely identifies it within a given graph or tree.

Def: edge – a directed link between two vertices consisting of the following:

- source – a string representing the name of the vertex at the tail end of the edge.
- destination – a string representing the name of the vertex at the head end of the edge.

Def: BBN – is a directed, acyclic graph consisting of a set of vertices, V, and a set of edges, E, where E is defined as, $E \subseteq V \times V$, and the of elements of E are defined as edges.

Def: functional_model – is a hierarchical, system representation consisting of an array of *func_nodes*, where each member can be accessed by an integer between 1 and the number of elements in the array.

Def: func_node – a node within a *functional_model* consists of the following elements:

- name: a string that must be unique with respect to all other nodes in the same *functional_model.*
- level: an integer that represents the level of abstraction at which the current *func_node* sits. This number ranges from 1 to n, where n is the total number of layers of abstraction in the *functional_model*, and the highest level of abstraction has a level of 1.
- children: an array of *names*, where each entry represents a *func_node* that is influenced by the given *func_node* identified by *name*, where each member can be accessed by an integer between 1 and *num_children.*
- num_children: an integer giving the number of entries in the *children* array.
- abs_children: an array of *names* of the *func_nodes* that are contained in the abstraction level below the current *func_node*, where each member can be accessed by an integer between 1 and *num_abs_children.*
- num_abs_children: an integer giving the number of entries in the *abs_children* array.

Def: hierarchy_tree – is a general tree consisting of a set of vertices, tree_v, and a set of edges, tree_e, where tree_e is defined as, $tree\_e \subseteq tree\_v \times tree\_v$, and the elements of tree_e are defined as edges. The root vertex represents level 0, with name equal to the empty string (""), and is only meant as an anchor for the *func_nodes* of the *functional_model*.

There is also a set of functions that is used by the main algorithm responsible for generating BBN structure, i.e., the set of vertices and the set of edges. Definitions for these functions follow:

Def_Function: create_edge(node1_name, node2_name)

- Input – two strings, node1_name is the name of the vertex at the tail of the directed edge, and node2_name is the name of the vertex at the head of the directed edge.
- Output – a directed edge.
- Activity – creates a directed edge that can be added to the set of edges of the resultant BBN, E.

Def_Function: create_vertex(string1)

- Input – a string, string1
- Output – a vertex capable of being added to the set of vertices in either the BBN or hierarchy tree.
- Activity – creates a vertex from a string

Def_Function: get_length_array(array1)

- Input – an array.
- Output – an integer representing the length (number of elements) of the array.

- Activity – counts the number of elements in the input array and returns that number.

The algorithm responsible for generating the set of vertices, V, and the set of directed edges, E, for the BBN is shown below:

Def_Function: generate_bbn_structure(fm)

- Input – a pointer to a *function_model*, fm.
- Output – two sets, the set of vertices, V, and the set of directed edges, E, for the resultant BBN.
- Activity – creates the set of vertices and the set of directed edges for the BBN. Details are shown below:

```
Begin main
        Local Variables:
        set V ← {};             //The set of vertices for the BBN, initially empty
        set E ←{};              //The set of directed edges for the BBN, initially empty
        set tree_v ←{};         //The set of vertices for the hierarchy_tree, initially empty
        set tree_e ←{};         //The set of directed edges for the hierarchy_tree, initially
                                //      empty
        integer deepest_level; //A variable used to find the deepest level of abstraction
        integer x,y,z;          //Variables used to index through arrays
        string st1, st2;        //Temporary strings used to create and inspect the elements
                                //      of edges
        edge e1;                //Temporary variable used in creating and manipulating
                                //      edges
        func_node f1;           //Temporary func_node used to traverse the
                                //      functional_model

        //First, find the deepest level of abstraction
        deepest_level ← 0;
        for(x←1; x ≤ get_length_array(fm); x ← x+1)
                if (fm[x].level > deepest_level) then
                        deepest_level ← fm[x].level;

        //Create the root vertex and add it to tree_v
        tree_v ← tree_v ∪ create_vertex("");
```

```
//Add vertices representing level 1 func_nodes to tree_v and the associated edges
//  to tree_e of the hierarchy_tree
for(x←1; x ≤ get_length_array(fm); x ← x+1)
        if (fm[x].level == 1) then begin
                st1 ← fm[x].name;
                tree_v ← tree_v ∪ create_vertex(st1);
                tree_e ← tree_e ∪ create_edge("",st1);
        end if;

//Go through the functional_model, fm, level by level, until all children of all
//  nodes have been added to tree_v and the associated edges to tree_e.
//  Although there only needs to be a check to ensure that the deepest level is not
//  entered, we felt that the algorithm was easier to understand stepping by levels.
for(x←1; x < deepest_level; x ← x+1)                    //Try all but deepest level
        for(y←1; y ≤ get_length_array(fm); y ← y+1)  //Step through fm
                if (fm[y].level == x) then begin          //Find those for curr. level
                        st1 ← fm[y].name;                 //Add nodes and edges
                        for(z←1; z ≤ fm[y].num_abs_children; z←z+1) begin
                                st2 ← fm[y].abs_children[z];
                                tree_v ← tree_v ∪ create_vertex(st2);
                                tree_e ← tree_e ∪ create_edge(st1,st2);
                        end for;
                end if;
//Now the hierarchy_tree has been constructed
//Next, the vertices and edges from the hierarchy_tree are copied to the set
//  of vertices and edges for the BBN, then the root node and associated edges
//  are removed from the sets, V and E, respectively
V ← tree_v;
E ← tree_e;
V ← V – create_vertex("");
for(x←1; x ≤ get_length_array(fm); x ← x+1)
        if (fm[x].level == 1) then
                E ← E – create_edge("",fm[x].name);
//Now the edges not associated with hierarchy are added by stepping through the
//  functional_model, until all influence related edges have been added
for(x←1; x ≤ get_length_array(fm); x ← x+1) begin   //Step through fm
        st1 ← fm[x].name;
        for(y←1; y ≤ fm[x].num_children; y←y+1) begin  //Step through children
                st2 ← fm[x].children[y];
                E ← E ∪ create_edge(st1,st2);          //Add edge
        end for;
end for;
```

return (V,E);
End main

The functional_model associated with Figure 13 and the corresponding hierarcy_tree are both shown in Figure 14, and the BBN that would be generated from the algorithm above is shown in Figure 13.

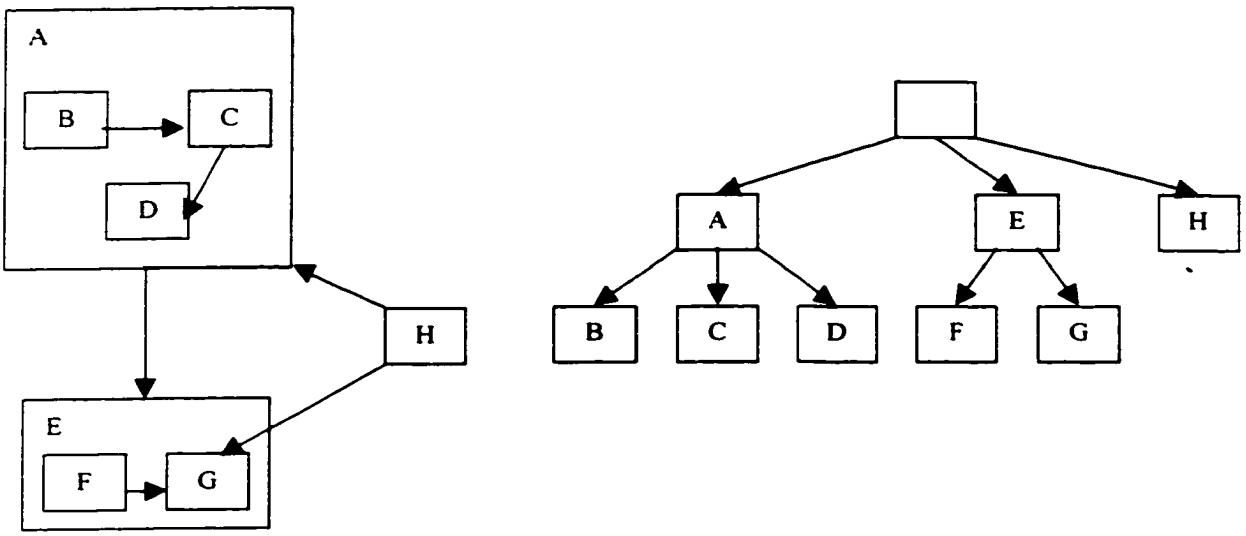


Figure 14: Functional_model (left) with associated hierarchy_tree (right).

GENERATION OF BBN LINK MATRICES

In order to calculate a quantifiable value, such as those contained within the link matrices of a BBN, we must be given some quantifiable input related to the operation of each functional component. Therefore, we have chosen to characterize each functional component with three types of measurement: the complexity of the functional component, the total bandwidth associated with the individual functional component (bits/sec.), and the frequency of execution of the functional component (executions/sec.). Each of these measurements is assumed to be available as output from the functional

simulation of the design model. The first of these measurements is also the most implementation dependant; the complexity of the functional component. Because complexity can mean so many things to so many people, we loosely define it as an estimate of the number of lines of underlying code within a StateChart[11] (also considering the type of code) representing a functional component. We calculate the complexity by adding the weighted code within a given StateChart and multiplying that result by the total number of states used. We rely on a table of costs such as that shown below to weight the lines of code:

- Execute simple instruction: Cost = 1. (Shift, compare, etc.)
- Execute a memory reference: Cost = 3. (Read from or write to memory)
- Execute a simple math calculation: Cost = 3. (Add, subtract, etc.)
- Execute a medium complexity calculation: Cost = 6. (Multiply, divide, etc.)
- Execute a complex math calculation: Cost = 10^r, where the complexity of the calculation is $O(n^r)$
- Execute a switching statement (such as "if...then"): Cost = 7
- Execute a loop: Cost = number_iterations*(7 + complexity of statements contained within loop)
- Execute an event: Cost = 20
- Schedule an event: Cost = 20 + number of ticks until event occurs

The key to remember is that the focus of this research is on the use of the BBN, and not on whether we have the best estimation table for complexity. These values were chosen as an estimate of the number of clock cycles needed to complete the given operation on a basic microprocessor such as a Motorola 68000. The next two measurements (bandwidth within a single functional component and frequency of execution) are easily obtainable from a simulation of the functional model with a tool such as StateMate [11].

Although we have chosen to use complexity, bandwidth, and frequency as our metrics of choice, there are obviously several different metrics that could have been

chosen. [8] shows that some of the most popular types of metrics include hardware area, delay, and power consumption, just to name a few. [8] also shows that closeness metrics are useful when no partition yet exists. Our use of relative complexity, relative bandwidth, and relative frequency, acts as closeness metrics that we use to construct the BBN representation before any partitioning takes place, just as [8] suggests.

When determining how to use these values to calculate the link matrices, it is important to recognize why BBNs were chosen in the first place. The role of the BBN is to show how the implementation decision for each functional component affects the decisions for the other functional components. The link matrices can then be viewed as a way to quantify these influences. Therefore, the individual values of complexity, bandwidth, or frequency for a single functional component are not as critical as their relation to the values of causally connected neighbors. It is the values of these measurements relative to the influenced functional components that are of importance. Thus, we use the following equation as a measurement for how "similar" the complexity of two functional components is (which we call the relative complexity):

$$rel\_comp(a,b) = \frac{1}{\log_{10}(\max(\frac{comp(a)}{comp(b)}, \frac{comp(b)}{comp(a)})) + 0.1} \qquad \text{(eq. 1)}$$

What this equation means is that the relative complexity between two functional components is inversely proportionate to the magnitude of the ratio of their complexities. Therefore, this function goes to a value of 10 for equal complexities, and asymptotically approaches 0 as the difference between the two complexities increases. Similarly, we define the relative bandwidth and relative frequency as shown in equations 2 and 3.

$$rel\_band(a,b) = \frac{1}{\log_{10}(\max(\frac{band(a)}{band(b)}, \frac{band(b)}{band(a)})) + 0.1} \quad \text{(eq. 2)}$$

$$rel\_freq(a,b) = \frac{1}{\log_{10}(\max(\frac{freq(a)}{freq(b)}, \frac{freq(b)}{freq(a)})) + 0.1} \quad \text{(eq. 3)}$$

Now that we know how each functional component relates to one another, the final step is to find a suitable matrix representation. Because some system designers may feel that one or more of the 3 measurements are more critical than the other(s), we allow a weight to be assigned to each sub-matrix (as shown in equation 4).

$$link\_matrix(a,b) = \alpha \begin{bmatrix} rel\_comp(a,b) & \frac{1}{rel\_comp(a,b)} \\ \frac{1}{rel\_comp(a,b)} & rel\_comp(a,b) \end{bmatrix} + \beta \begin{bmatrix} rel\_band(a,b) & \frac{1}{rel\_band(a,b)} \\ \frac{1}{rel\_band(a,b)} & rel\_band(a,b) \end{bmatrix}$$
$$+ \gamma \begin{bmatrix} rel\_freq(a,b) & \frac{1}{rel\_freq(a,b)} \\ \frac{1}{rel\_freq(a,b)} & rel\_freq(a,b) \end{bmatrix} \quad \text{(eq. 4)}$$

Chapter 5 DETAILED EXAMPLE

In this section, we present a programmable thermostat design example. Figure 15 shows the functional model used in this example. Figure 16 gives a block representation of the functional model that will make the generation of the structural portion of the BBN easier to follow.

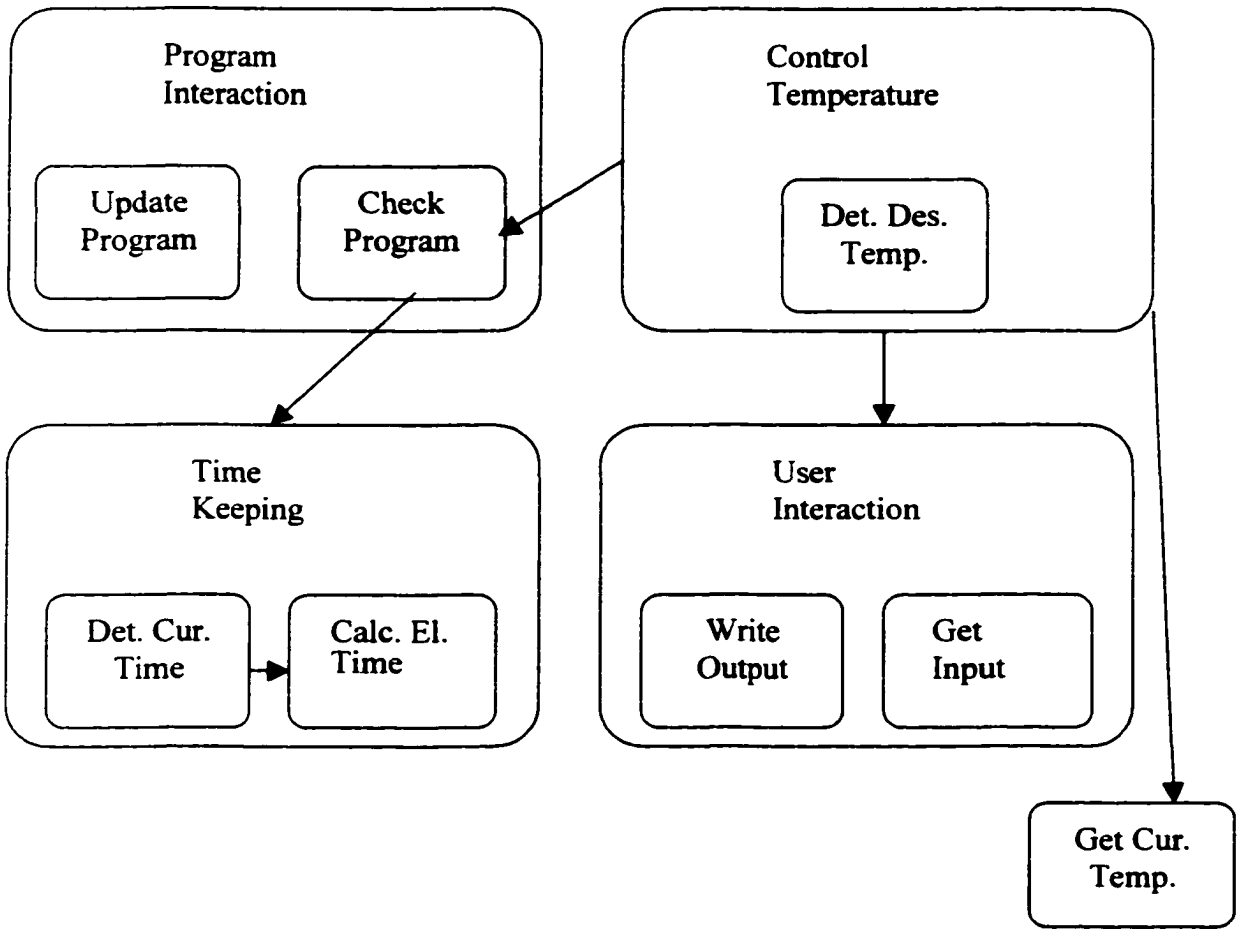


Figure 15: Functional model for programmable thermostat.

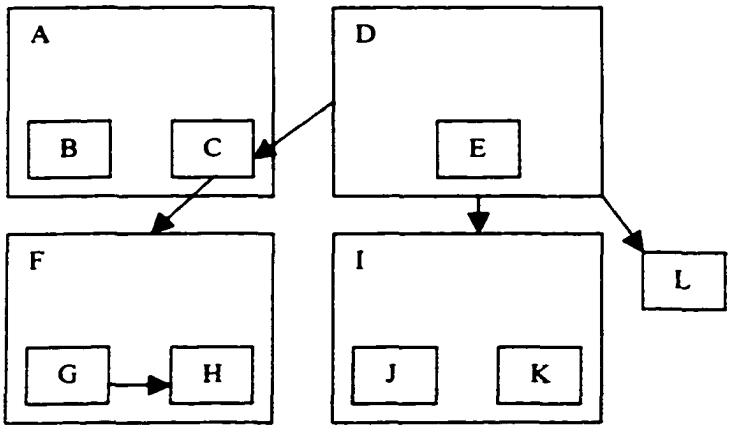


Figure 16: Block representation of functional model.

After executing the first steps of the generation algorithm, Figure 17 (a) shows the two levels of abstraction of the functional model in include five components; one that contains no sub-components (L), and four components that act as abstraction modules. All links between the five top-level components are shown in Figure 17, and since this is the iteration of the algorithm corresponding to the first level of abstraction, there are no links between different levels of abstraction.

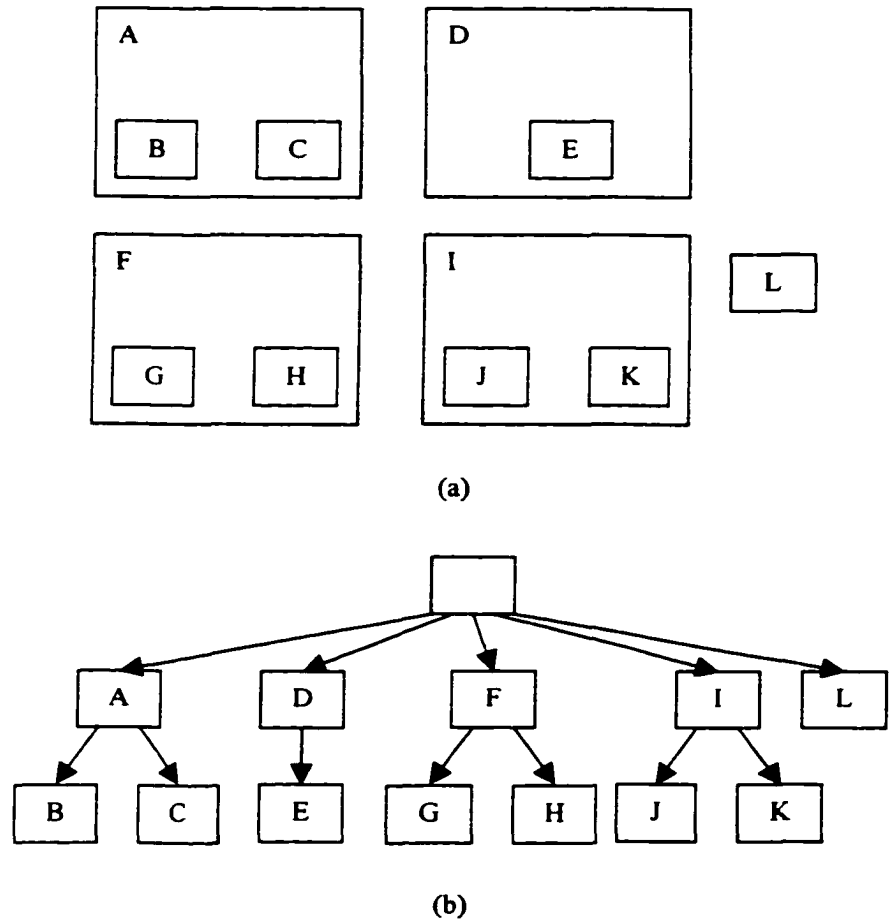


Figure 17: Hierarchical elements from functional model (a) and corresponding hierarchy tree (b).

The second iteration of the BBN structural generation algorithm brings about many new nodes corresponding to the sub-components of 'A,' 'D,' 'F,' and 'I'. Figure 18 shows the system model at the first two levels of abstraction, and the matching BBN. Note that links have been added both for inter-node connections between nodes of the same abstraction level, and the links from the abstract parent nodes to the subordinate nodes representing their sub-components. Since this functional model only contains two levels of abstraction, the algorithm stops at this point.

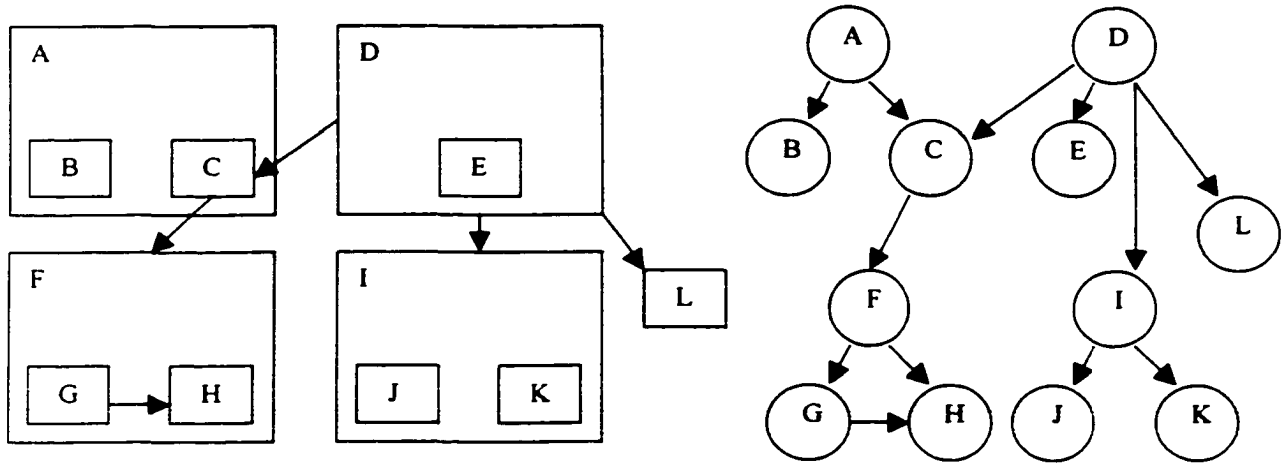


Figure 18: After second iteration of algorithm both levels of the functional model (left) with the corresponding BBN (right).

For the generation of the link matrices and the introduction of evidences, we have chosen to use the reduced functional model shown in Figure 19 because of the reduced computational complexity and better understandability of the effects of evidence propagation.

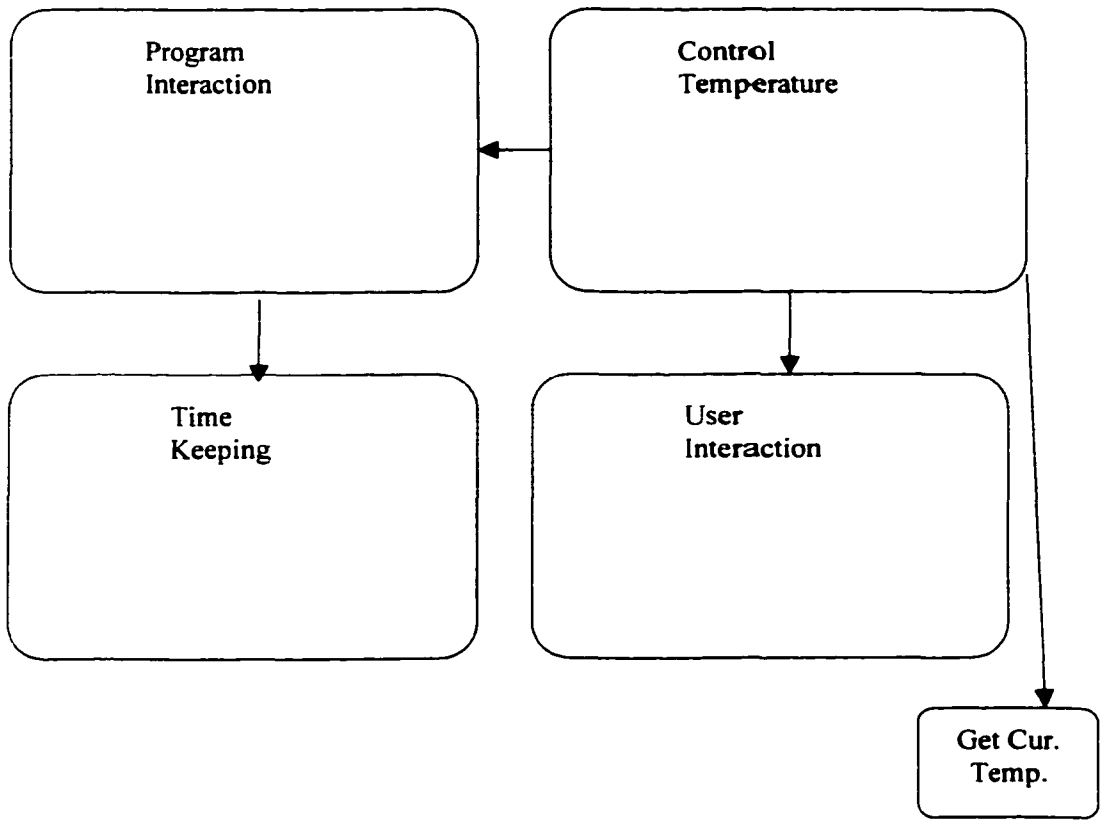


Figure 19: Reduced functional model for programmable thermostat.

Because of the reduced number of nodes, the generation of the structural portion of the BBN becomes trivial. Therefore, the final result of the BBN structural generation algorithm is shown in Figure 20.

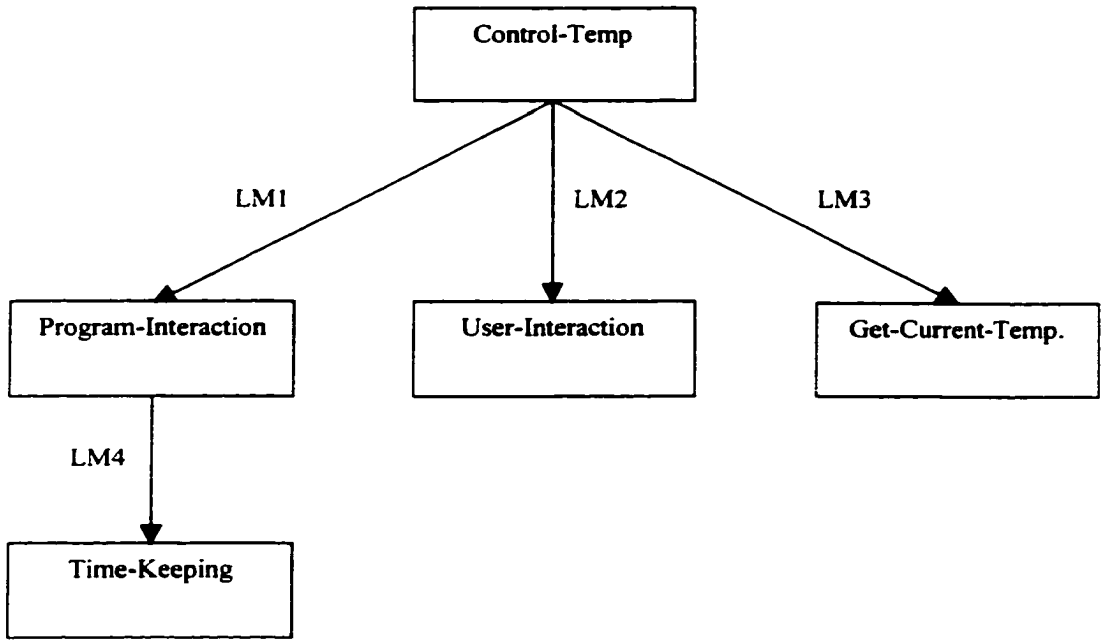


Figure 20: BBN representation of reduced functional model.

In order to use the equations from Chapter 4, the complexity, bandwidth, and frequency of each functional component must be known, along with the values of the weights for equation (4). In order to get realistic values for each functional component, a StateMate [11] representation was created and simulated. Although we created and simulated a StateMate [11] model, this task actually belongs in the step before that which contains hardware/software partitioning. According to the overview of codesign figure (Figure 9), the creation and simulation of the functional model would occur during the "MODEL" phase.

Figure 21 through Figure 26 consist of the main activity chart, four activity sub-charts, and one state chart. The Time-Keeping activity chart consists of the TK_CONTROL state chart (shown in Figure 22) that controls both keeping track of time and provides a mechanism to access the current time. The Program-Interaction activity

contains the PI_CONTROL state chart (shown in Figure 23) is responsible for saving a program that the user may enter, and in checking whether a program is valid given the current time. The User-Interaction activity chart consists of the UI_CONTROL state chart, that is shown in Figure 24, represents the actions of the user entering in a simple program consisting of a start and stop time and a desired temperature to maintain. The Get-Current-Temp activity chart consists of the GCT_CONTROL state chart (shown in Figure 25) which represents obtaining the current temperature from a sensor and converting it to a digital value. GCT_CONTROL also updates the current temperature when the heater or cooler is running. The CONTROL_TEMPERATURE statechart (shown in Figure 26) acts as both the manager of distributing user input, and control whether or not the heater or cooler is run.

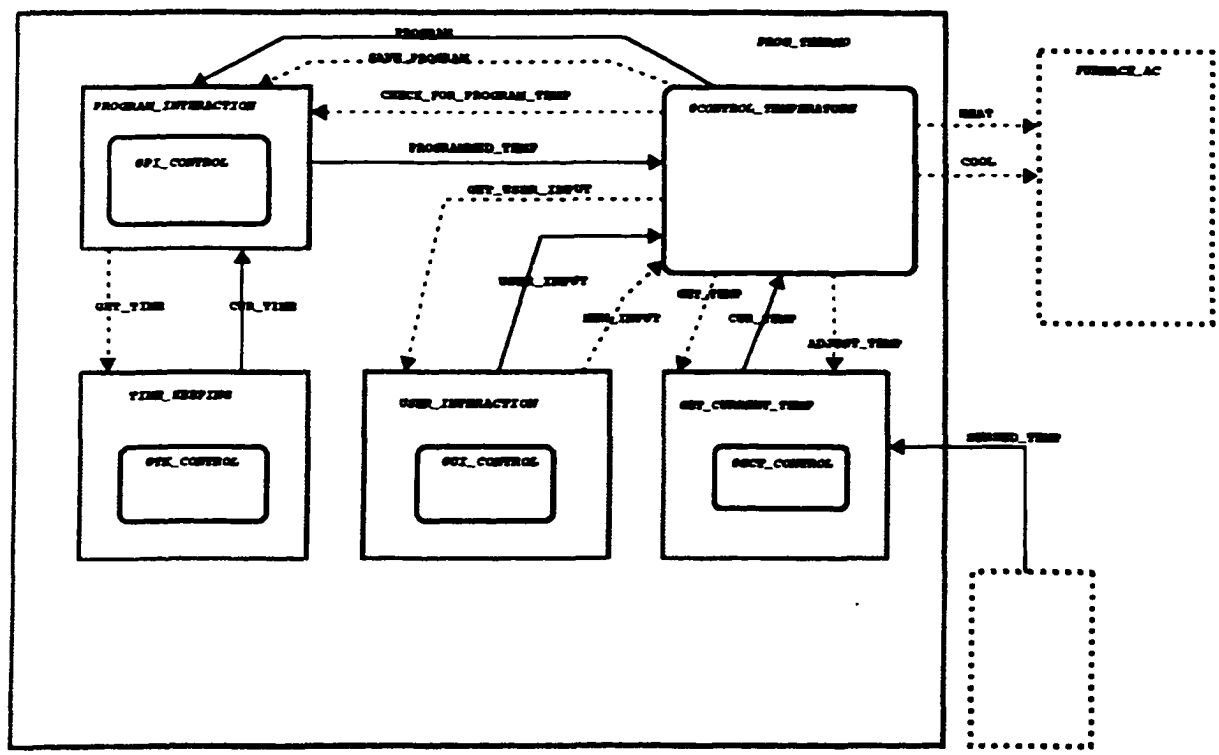


Figure 21: Main activity chart for programmable thermostat example.

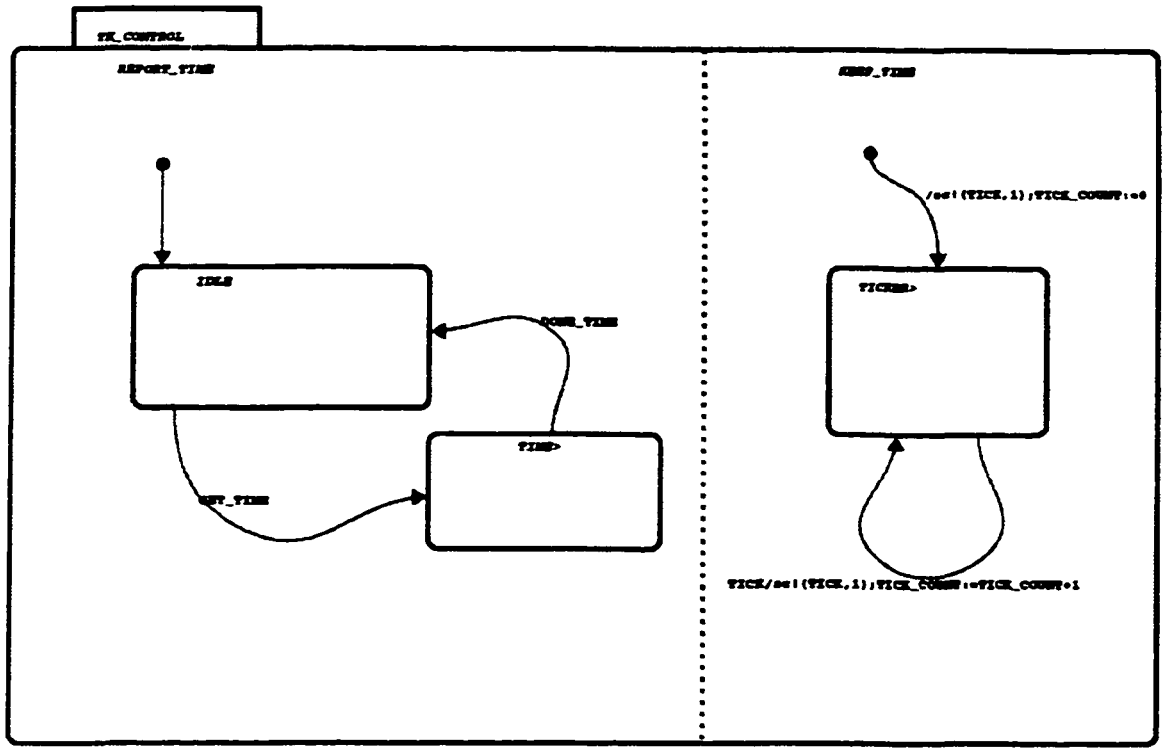


Figure 22: Time keeping control state chart.

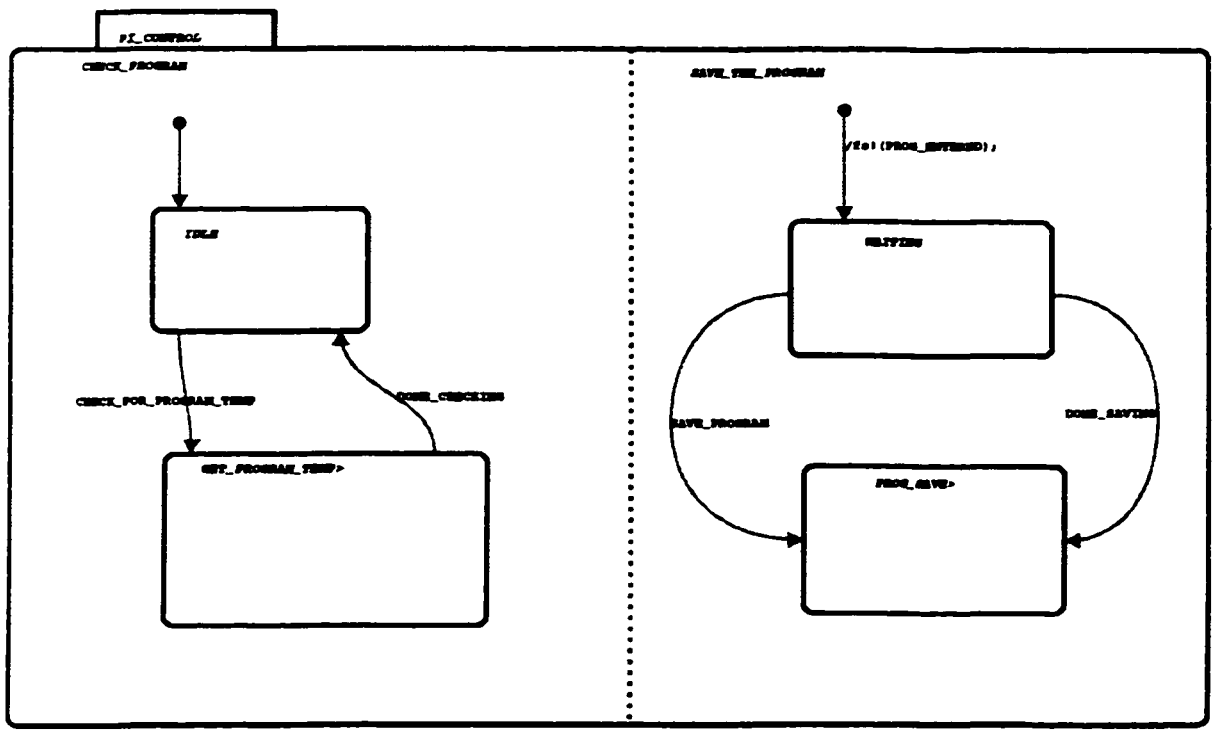


Figure 23: Program control state chart.

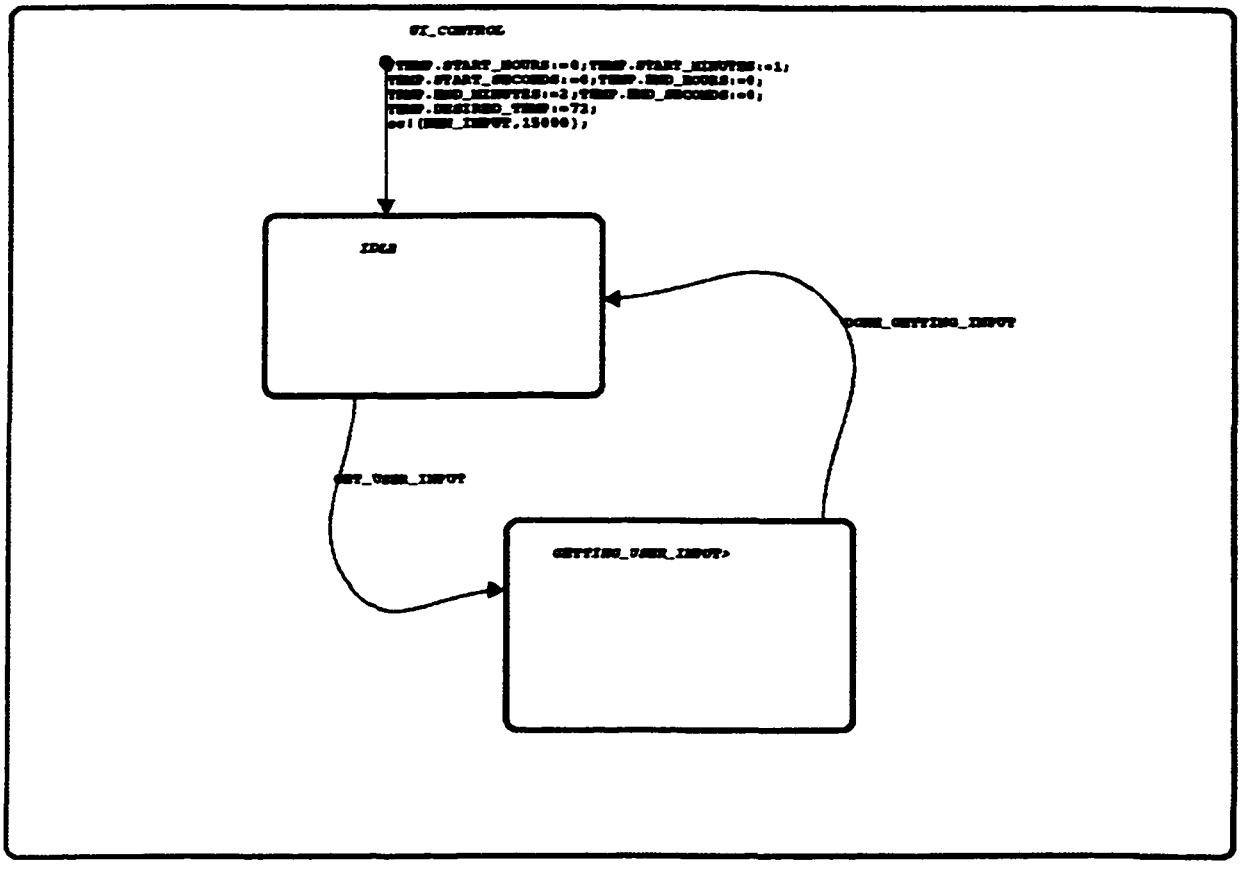


Figure 24: User input control state chart.

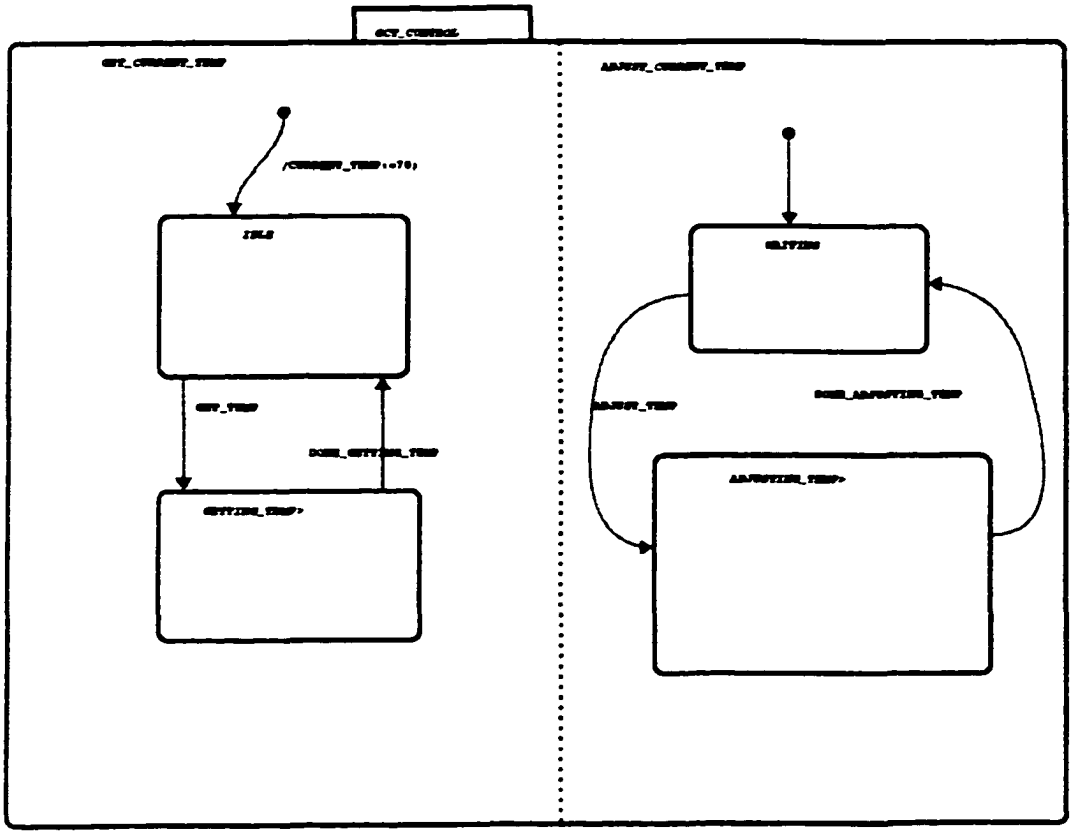


Figure 25: Get current temperature control statechart.

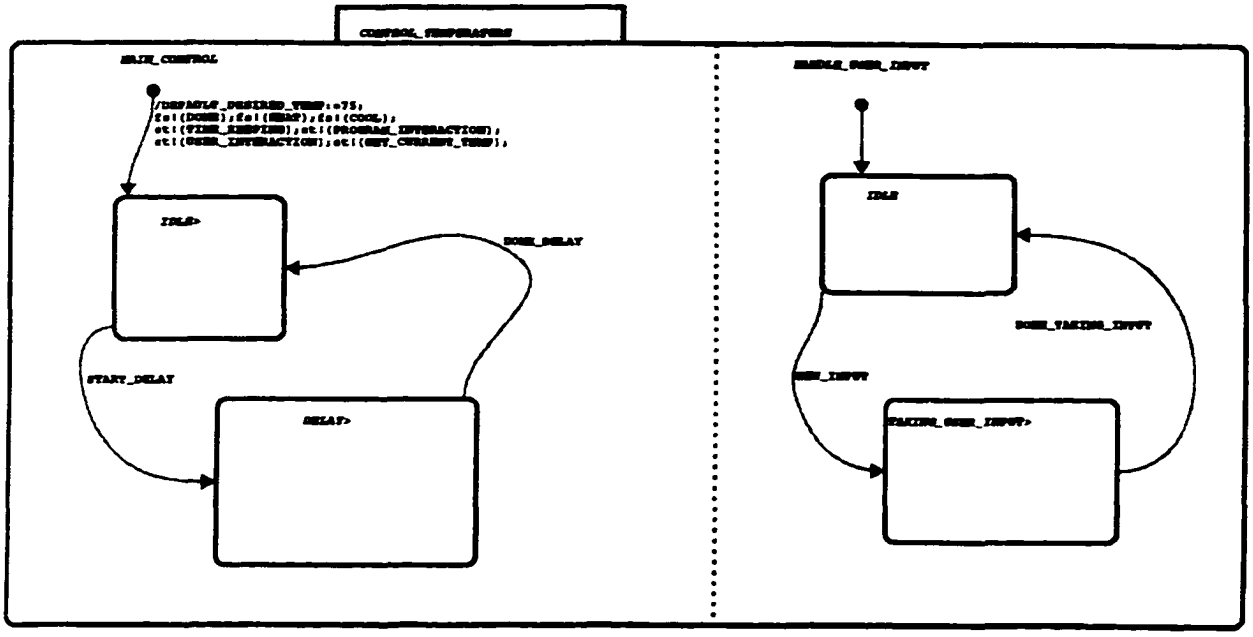


Figure 26: Control temperature statechart.

For this example, we will show how the complexity, bandwidth, and frequency of execution are calculated for the Control-Temperature functional component. First we will calculate the complexity. It can be easily seen that the Control-Temperature statechart of Figure 26 contains four distinct states. We have taken the embedded code from all of the states in the Control-Temperature statechart and placed the associated complexity values for each line next to the line in Figure 27. From adding the numbers in Figure 27 and multiplying by the number of states, four, we get a complexity value of 6368.

To calculate the bandwidth of Control-Temperature, we take all data and control lines coming into and exiting the statechart (as shown in Figure 21), multiply each by the size of the data in bits, and multiply each by the number of times data passes through the particular data or control line each second to obtain a value with units of bits/second. When making these calculations, we assume that all integers are 32 bits long, and that events are coded as integers. One important note to make is that since Control-Temperature executes about once every second, and since it drives the execution of all of the other components, all of the control lines are used once per second. We assume that control lines and data lines associated with intermittent execution are instead executed once per second for ease of calculation, instead of using statistical models.

There are 8 control lines connected to Control-Temperature, each with an event occurring about once a second for a total of 256 bits/second. Both PROGRAM and USER_INPUT (shown in Figure 21) transmit records of seven integers each for a total of 448 bits/second. Finally, the last two data lines, PROGRAMMED_TEMP and

CUR_TEMP each transmit an integer for a total of 64 bits/second. The total bandwidth associated with Control-Temperature is then 768 bits/second.

To determine the frequency of execution, we simply find the fastest executing portion of the statechart associated with a functional model. For Control-Temperature, this corresponds to the time it takes to take in and transmit a program; fives times a second. All of the values just calculated are given in the table of Figure 28, along with the values for the other functional components.

```
/DEFAULT_DESIRED_TEMP := 75;                                        3
fs!(DONE); fs!(HEAT); fs!(COOL);                                    9
st!(TIME_KEEPING); st!(PROGRAM_INTERACTION);                        40
st!(USER_INTERACTION); st!(GET_CURRENT_TEMP);                       40

entering/
        if not DONE then                                            7
                CHECK_FOR_PROGRAM_TEMP;                             20
                GET_TEMP;                                           20
                if GLOBAL_DESIRED_TEMP>0 then                       7
                        if GLOBAL_DESIRED_TEMP>CURRENT_TEMP then    7
                                tr!(HEAT);                          3
                                ADJUST_TEMP;                        20
                        end if;
                        if GLOBAL_DESIRED_TEMP<CURRENT_TEMP then    7
                                tr!(COOL);                          3
                                ADJUST_TEMP;                        20
                        end if;
                        if GLOBAL_DESIRED_TEMP=CURRENT_TEMP then    7
                                fs!(COOL);                          3
                                fs!(HEAT);                          3
                        end if;
                else
                        if DEFAULT_DESIRED_TEMP>CURRENT_TEMP then   7
                                tr!(HEAT);                          3
                                ADJUST_TEMP;                        20
                        end if;
                        if DEFAULT_DESIRED_TEMP<CURRENT_TEMP then   7
                                tr!(COOL);                          3
                                ADJUST_TEMP;                        20
                        end if;
                        if DEFAULT_DESIRED_TEMP=CURRENT_TEMP then   7
                                fs!(COOL);                          3
                                fs!(HEAT);                          3
                        end if;
                end if;
                START_DELAY;                                        20
        end if;


entering/ sc!(DONE_DELAY,1000);                                     1020

entering/ GET_USER_INPUT;                                           20
        SAVE_PROGRAM;                                               20
        sc!(DONE_TAKING_INPUT,200);                                 220
```

Figure 27: Embedded code for Control-Temp statechart with associated complexity values.

	complexity	bandwidth	freq. of exe.
Control-Temp	6368	768	5
Program-Interaction	1708	384	10
User-Interaction	2088	288	1
Get-Current-Temp.	644	128	1000
Time-Keeping	556	64	1000

Figure 28: Table of complexity, bandwidth, and frequency of execution for functional components in programmable thermostat example.

For this example, we have chosen the following weights: $\alpha=3$, $\beta=1$, $\gamma=3$. We obtain the set of link matrices shown in Figure 29.

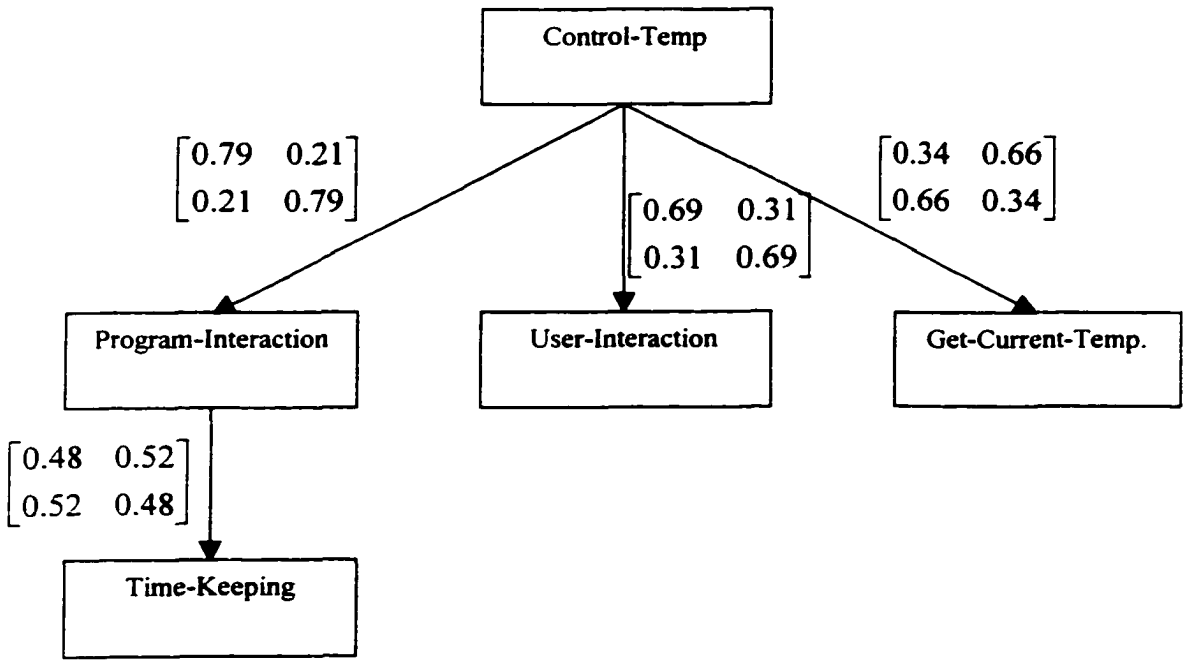


Figure 29: BBN for programmable thermostat after the generation of the link matrices.

In the formulation of these matrices, the emphasis was placed on complexity and frequency. This can be seen especially well in the link matrix on the link from 'Control-Temperature' to 'Get-Current-Temp' where the probability that 'Get-Current-Temp' should be implemented in the same way as 'Control-Temperature' is only 34%.

To continue, we now have a BBN that has been created for the scaled-down thermostat example. We use a BBN calculation tool to perform the propagation of evidence and update of beliefs called Hugin Lite which is a share-ware version of the Hugin System by Hugin Expert A/S. Initially, all beliefs are equal, but let's say that we introduce evidence to 'Get-Current-Temp' that it is 83% likely that it should be implemented in hardware. Figure 30 illustrates how this evidence affects the rest of the BBN.

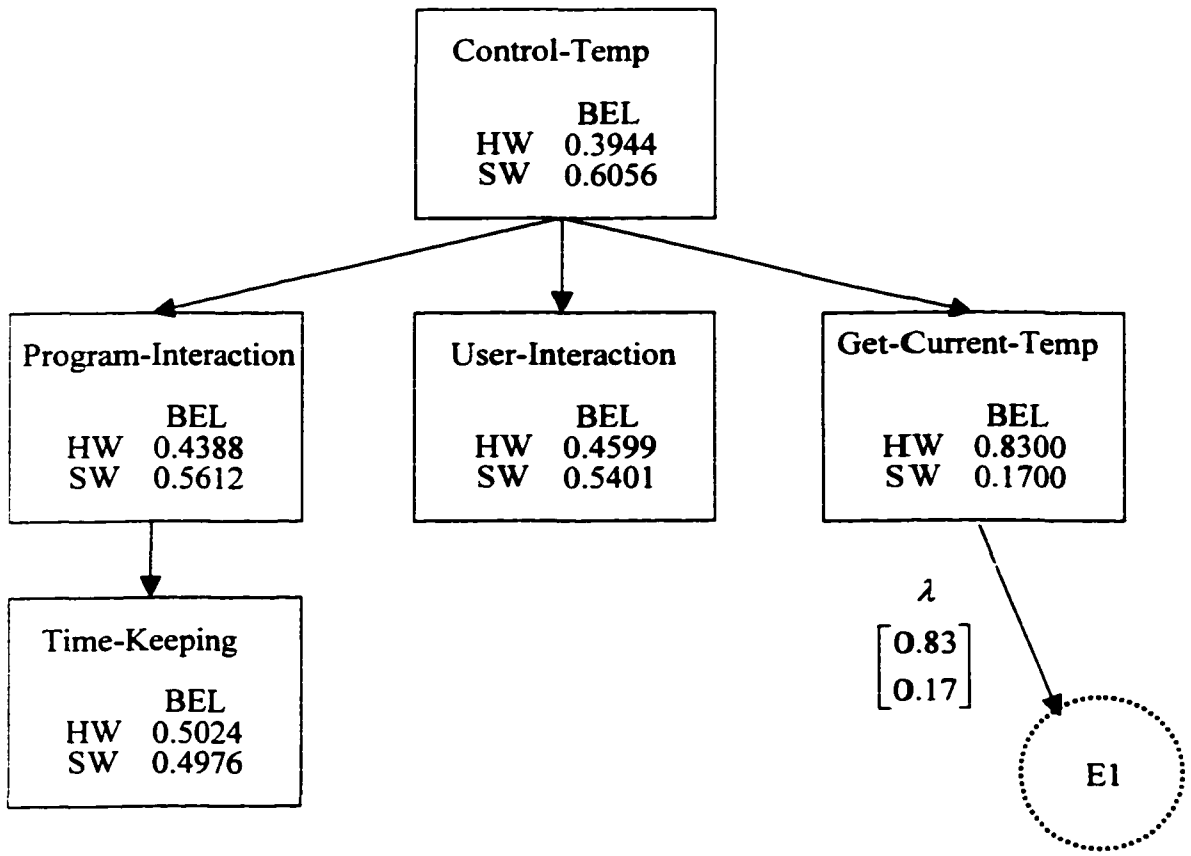


Figure 30: Beliefs associated with each functional component after the introduction of the first piece of evidence.

Because of the vast differences between the 'Get-Current-Temp' and 'Control-Temp' functional components, the introduction of this evidence actually drives 'Control-Temp'

towards a software implementation. Next we introduce evidence that 'Program-Interaction' has a 65% chance that it should be implemented in software, and the result is shown in Figure 31.

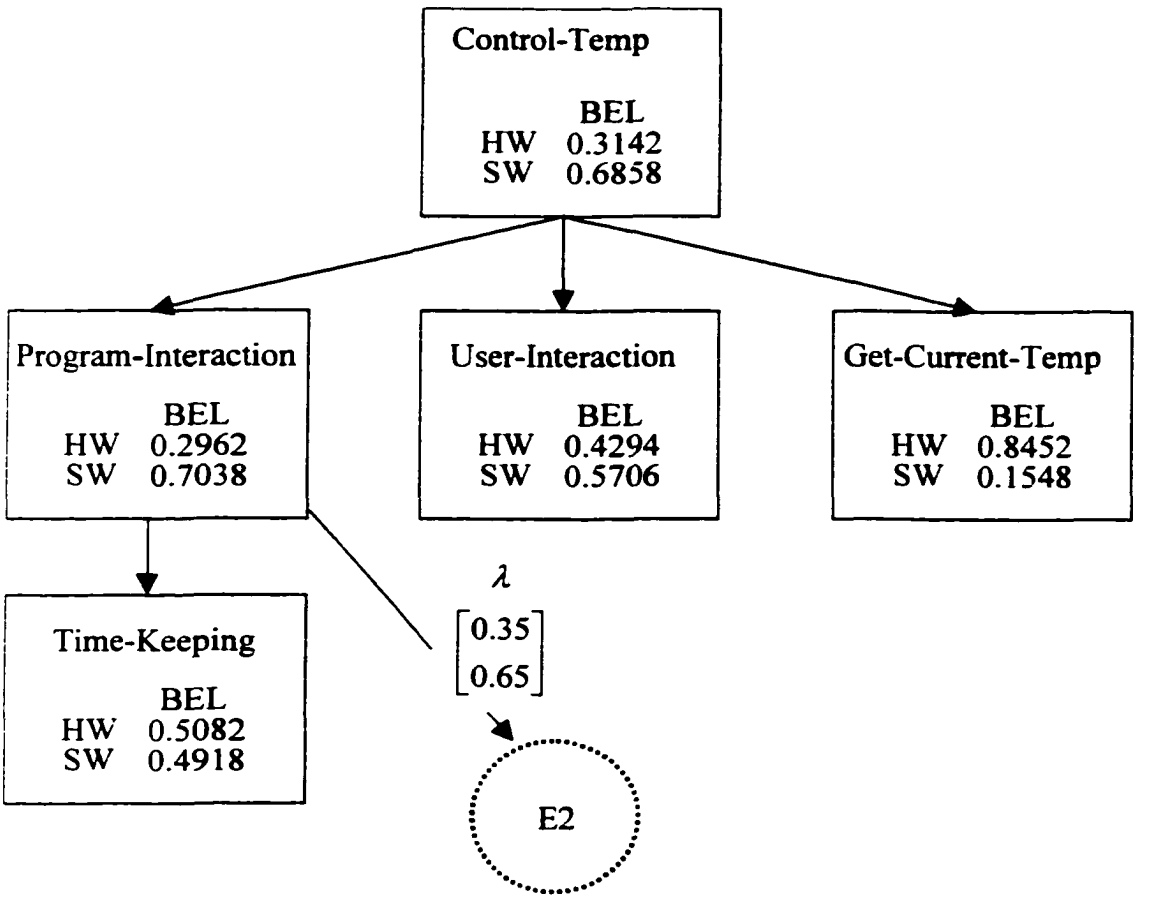


Figure 31: Beliefs for the functional components after the introduction of a second piece of evidence.

Here we see that, although 'Program-Interaction has been driven towards a software implementation, 'Time-Keeping' has been driven slightly towards hardware. This difference can be directly attributed to the discrepancies between the two functional components as illustrated in Figure 29.

The process of propagating evidences throughout the BBN continues until no new data can be introduced. At this point, a decision is made as to whether or not each function should be implemented in hardware or software. This decision is based upon the amount of belief associated with each type of implementation (i.e., if the belief is greater than some threshold, e.g., 75%, then that type of implementation will be chosen). The classification of a function into hardware or software is reflected in the system model, and the process continues until all functions can be classified into either hardware or software.

Chapter 6 CONCLUSION

In this dissertation, we have introduced a new methodology for hardware and software partitioning. Through a detailed example system, we have shown how Bayesian Belief Networks can be used to propagate evidence regarding classification of functions into hardware or software realizations. This propagation permits the effects of a classification decision made about one function to be felt throughout the entire network. In addition, because BBNs have a belief of hypotheses as their core, we know how well a given classification fits into either hardware or software. Knowing that a function with a 75% hardware belief, should be implemented 75% of the time in hardware allows the user to have a measure of the appropriateness of their solution.

This dissertation also introduces a methodology for the generation of both the qualitative and quantitative portions of a BBN in the hardware/software partitioning domain. The generation of the structure of the BBN exploits the ability to use abstract, complex models in the representation of the functional components. The generation of the associated link matrices uses quantifiable aspects of the individual functional components to determine relative properties between components connected by causal links.

Future work includes testing other metrics and combinations of other metrics for the link matrix generation. We would also like the ability to plug our BBN results into a simulation engine that would provide more accurate measurements that can be converted into evidences and fed back into the BBN partitioning methodology.

Appendix A – Code Listing For Matrix Generation

```
// lm_build.c    John Olson
//  Builds the link matrices for the bbn

#include <stdio.h>
#include <string.h>
#include <malloc.h>
#include <math.h>

#define ROWS 2
#define COLS 2
#define MAX_LETTERS 50
#define MAX_NODES 20

typedef double matrix[ROWS][COLS];

typedef struct link *link_ptr;

typedef struct node *node_ptr;

typedef struct node {
        int num;
        char *name;
        double complexity;
        double bandwidth;
        double frequency;
        link_ptr links;
} Node, *Node_Ptr;

typedef struct link {
        int s_num, e_num;
        char *s_name;
        char *e_name;
        matrix mat;
        node_ptr n;
        link_ptr next;
} Link, *Link_Ptr;


//Global Structures and Variables
Node_Ptr Nodes[MAX_NODES];


//alloc_link - allocate a link struct

Link_Ptr alloc_link(int s, int e, char *sn, char *en, Node_Ptr n)
{
        Link_Ptr temp;
        int i,j;
```

```
        temp = (Link_Ptr)malloc(sizeof(Link));
        temp -> s_num = s;
        temp -> e_num = e;
        temp -> s_name = strdup(sn);
        temp -> e_name = strdup(en);
        for (i=0; i<ROWS; i++)
                for (j=0; j<COLS; j++)
                        temp -> mat[i][j] = 0.0;
        temp -> n = n;
        temp -> next = NULL;

        return temp;
}

//alloc_node - allocate a node struct

Node_Ptr alloc_node(int n, char *nm, double c, double b, double f)
{
        Node_Ptr temp;

        temp = (Node_Ptr)malloc(sizeof(Node));
        temp -> num = n;
        temp -> name = strdup(nm);
        temp -> complexity = c;
        temp -> bandwidth = b;
        temp -> frequency = f;
        temp -> links = NULL;

        return temp;
}


//init_nodes - sets all of the array of nodes to NULL

void init_nodes(void)
{
        int i;

        for (i=0; i<MAX_NODES; i++)
                Nodes[i] = NULL;      //global ref
}


//get_nodes - fills in the global Nodes array

void get_nodes(void)
{
        int num_nodes=0;
        int i;
        double temp_c, temp_b, temp_f;
        char temp_name[MAX_LETTERS];

        init_nodes();
```

```
        printf("Enter the total number of nodes> ");
        scanf("%d", &num_nodes);

        for (i=0; i<num_nodes; i++) {
                printf("Enter the name for node number %d > ",(i+1));
                scanf("%s",temp_name);
                printf("Enter the comp., bandwidth, and freq. > \n");
                scanf("%lf%lf%lf", &temp_c, &temp_b, &temp_f);
                Nodes[i]=alloc_node(i+1, temp_name, temp_c, temp_b, temp_f);
        }
}


//get_links - fills in links

void get_links(void)
{
        int s, e, test;
        Link_Ptr p, q;
        Node_Ptr start, end;
        bool done = false;

        while (!done) {
                printf("Enter start and end node numbers for link \n");
                scanf("%d%d", &s, &e);
                start = Nodes[s-1];
                end = Nodes[e-1];
                q = alloc_link(s,e,start->name,end->name,end);
                p = start->links;
                if (p == NULL)
                        start->links = q;
                else {
                   while (p->next != NULL)
                                p = p->next;
                        p->next = q;
                }
                printf("Enter 1 to add another edge, or 0 to quit > ");
                scanf("%d", &test);
                if (test != 1)
                        done = true;
        }
}


//print_mat  - prints the matrices associated with each link
void print_mat(void)
{
        Node_Ptr n;

        Link_Ptr l;

        int i = 0;
```

```
        n = Nodes[i];

        while ((i < MAX_NODES) && ((n=Nodes[i]) != NULL)) {
                l = n->links;
                while (l != NULL) {
                        printf("The link from node %d to node %d has the following value: \n",
                                l->s_num, l->e_num);
                        printf("\t\t [%.2lf  %.2lf] \n",l->mat[0][0], l->mat[0][1]);
                        printf("\t\t [%.2lf  %.2lf] \n\n",l->mat[1][0], l->mat[1][1]);
                        l = l->next;
                }
                i++;
        }
}


//get_weights - get the associated weights for each of the 3 matrices

void get_weights(double *cw, double *bw, double *fw)
{
        printf("Please enter the weights: \n");
        scanf("%lf%lf%lf", cw, bw, fw);
        printf("\n");
}

//calc_matrices - calculates the matrix associated with each link

void calc_matrices(double cw, double bw, double fw)
{
        matrix c_mat, b_mat, f_mat, final_mat;
        double norm, ratio;
        double rel, inv_rel;

        Node_Ptr m;

        Link_Ptr l;

        int i = 0;

        m = Nodes[i];

        while ((i < MAX_NODES) && ((m=Nodes[i]) != NULL)) {
                l = m->links;
                while (l != NULL) {  //calculate for each link

                        //complexity
                        if (m->complexity >= l->n->complexity)
                                ratio = m->complexity/l->n->complexity;
                        else
                                ratio = l->n->complexity/m->complexity;
                        inv_rel = log10(ratio) + 0.1;
                        rel = 1.0/inv_rel;
```

```
                norm = rel + inv_rel;
                c_mat[0][0] = rel/norm;
                c_mat[0][1] = inv_rel/norm;
                c_mat[1][0] = inv_rel/norm;
                c_mat[1][1] = rel/norm;

                //bandwidth
                if (m->bandwidth >= l->n->bandwidth)
                        ratio = m->bandwidth/l->n->bandwidth;
                else
                        ratio = l->n->bandwidth/m->bandwidth;
                inv_rel = log10(ratio) + 0.1;
                rel = 1.0/inv_rel;
                norm = rel + inv_rel;
                b_mat[0][0] = rel/norm;
                b_mat[0][1] = inv_rel/norm;
                b_mat[1][0] = inv_rel/norm;
                b_mat[1][1] = rel/norm;

                //frequency
                if (m->frequency >= l->n->frequency)
                        ratio = m->frequency/l->n->frequency;
                else
                        ratio = l->n->frequency/m->frequency;
                inv_rel = log10(ratio) + 0.1;
                rel = 1.0/inv_rel;
                norm = rel + inv_rel;
                f_mat[0][0] = rel/norm;
                f_mat[0][1] = inv_rel/norm;
                f_mat[1][0] = inv_rel/norm;
                f_mat[1][1] = rel/norm;


                final_mat[0][0] = cw*c_mat[0][0] + bw*b_mat[0][0] + fw*f_mat[0][0];
                final_mat[0][1] = cw*c_mat[0][1] + bw*b_mat[0][1] + fw*f_mat[0][1];
                final_mat[1][0] = cw*c_mat[1][0] + bw*b_mat[1][0] + fw*f_mat[1][0];
                final_mat[1][1] = cw*c_mat[1][1] + bw*b_mat[1][1] + fw*f_mat[1][1];

                norm = final_mat[0][0] + final_mat[1][0];
                l->mat[0][0] = final_mat[0][0]/norm;
                l->mat[1][0] = final_mat[1][0]/norm;
                norm = final_mat[0][1] + final_mat[1][1];
                l->mat[0][1] = final_mat[0][1]/norm;
                l->mat[1][1] = final_mat[1][1]/norm;

                l = l->next;
        }
        i++;
    }
}
```

```
void main(void)

{
        double c_weight=0;
        double b_weight=0;
        double f_weight=0;

        get_nodes();
        get_links();
        get_weights(&c_weight, &b_weight, &f_weight);
        calc_matrices(c_weight, b_weight, f_weight);
        print_mat();
}
```

REFERENCES

[1]. Alpert, Charles J.; Kahng, Andrew B., "Recent Directions In Netlist Partitioning: a Survey," *Integration, the VLSI Journal*, Vol. 19, No. 1-2, August 1995, pp. 1-81.

[2]. Catania, V.; Malgeri, M.; Russo, M., "Applying fuzzy logic to codesign partitioning," *IEEE Micro*, Vol. 17, No. 3, May-June 1997, pp. 62 –70.

[3]. Charniak, E., "Bayesian networks without tears," *AI Magazine*, Vol. 12, No. 4, (winter 1991) pp. 50-63.

[4]. Chiodo, M.; Giusto, P.; Jurecska, A.; Hsieh, H.C.; Sangiovanni-Vincentelli, A.; Lavagno, L., "Hardware-Software Codesign of Embedded Systems," *IEEE Micro*, 1994, Vol. 14, No. 4, pp. 26-36.

[5]. Cunning, S.J.; Ewing, T.C.; Olson, J.T.; Rozenblit, J.W.; Schulz, S., "Towards an integrated, model-based codesign environment," Proceedings ECBS '99, IEEE Conference and Workshop on Engineering of Computer-Based Systems, 1999, pp. 136 –143.

[6]. De Micheli, G.; Gupta, R.K., "Hardware/Software Co-Design," *Proceedings of the IEEE*, Vol. 85, No. 3, March 1997, pp. 349-365.

[7]. Eles, P.; Zebo, Peng; Kuchcinski, K.; Doboli, A., "Hardware/software partitioning with iterative improvement heuristics," 1996 Proceedings 9th International Symposium on System Synthesis, 1996, pp. 71 –76.

[8]. Gajski, D.; Narayan, S.; Vahid, F.; Gong, J., *Specification and Design of Embedded Systems*, Englewood Cliffs, NJ: Prentice-Hall, 1994.

[9]. Grode, J.; Knudsen, P.V.; Madsen, J., "Hardware resource allocation for hardware/software partitioning in the LYCOS system," Proceedings 1998 Design, Automation and Test in Europe, 1998, pp. 22 –27.

[10]. Gupta, R.K.; De Micheli, G., "System-level synthesis using re-programmable components," Proceedings of the [3rd] European Conference on Design Automation, 1992, pp. 2 –7.

[11]. Harel, D., "STATEMATE: A Working Environment for the Development of Complex Reactive Systems", *IEEE Transactions on Software Engineering*, Vol. 16, No. 4, pp. 403-414, 1990.

[12]. Hendry, D.C.; Sananikone, D.S., "Hardware/software partitioning of embedded systems with multiple hardware processes," IEE Proceedings-Computers and Digital Techniques, Vol. 144, No. 5, Sept. 1997, pp. 285 –294.

[13]. Henkel, J.; Ernst, R., "High-level estimation techniques for usage in hardware/software co-design," Proceedings of the ASP-DAC '98 Asia and South Pacific Design Automation Conference 1998, pp. 353 –360.

[14]. Jensen, Finn V., *An Introduction To Bayesian Networks*, UCL Press, London, UK, 1998.

[15]. Kalavade, A.; Subrahmanyam, P.A., "Hardware/software partitioning for multifunction systems," IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, Vol. 17, No. 9, Sept. 1998, pp. 819 –837.

[16]. Knudsen, P.V.; Madsen, J., "Integrating communication protocol selection with partitioning in hardware/software codesign," Proceedings 11th International Symposium on System Synthesis, 1998, pp. 111 –116.

[17]. Lopez, M.L.; Iglesias, C.A.; Lopez, J.C., A knowledge-based system for hardware-software partitioning," Proceedings 1998 Design, Automation and Test in Europe, 1998, pp. 914 –915.

[18]. Maciel, P., Barros, E.; Rosenstiel, W., "A Petri net based approach for performing the initial allocation in hardware/software codesign," IEEE Proceedings of the International Conference on Systems, Man, and Cybernetics, Vol. 1, October 1998, pp. 505 –510.

[19]. Olson, John T.; Rozenblit, Jerzy W., "Framework For Hardware/Software Partitioning Utilizing Bayesian Belief Networks," IEEE Proceedings of the International Conference on Systems, Man and Cybernetics, Vol. 4, October 1998, pp. 3983-3988.

[20]. Pearl, J., *Probabilistic Reasoning in Intelligent Systems : Networks of Plausible Inference*, Morgan Kaufmann Publishers, San Mateo, CA, 1988.

[21]. Rozenblit, J.W.; Buchenrieder, K., (Eds.), *Codesign: Computer-Aided Software/Hardware Engineering*, IEEE Press, 1994.

[22]. Russell, Stuart J.; Norvig, Peter, *Artificial Intelligence: A Modern Approach*, Prentice Hall Publishers, Upper Saddle River, NJ, 1995.

[23]. Saha, D.; Mitra, R.S.; Basu, A., "Hardware software partitioning using genetic algorithm," Proceedings Tenth International Conference on VLSI Design, 1997, pp. 155 –160.

[24]. Schulz, S.; Rozenblit, J.W.; Mrva, M.; Buchenrieder, K., "Model-Based Codesign: the Framework and its Application, *IEEE Computer*, August 1998.

[25]. Srinivasan, V.; Radhakrishnan, S.; Vemuri, R., "Hardware software partitioning with integrated hardware design space exploration," Proceedings 1998 Design, Automation and Test in Europe, 1998, pp. 28 –35.

[26]. Thomas, Donald E.; Adams, Jay K.; Schmit, Herman, "A Model and Methodology for Hardware-Software Codesign," *IEEE Design and Test of Computers*, Vol. 10, No. 3, Sept. 1993, pp. 6-15.

[27]. Vahid, Frank, "Modifying Min-Cut for Hardware and Software Functional Partitioning," Proceedings of the Fifth International Workshop on Hardware/Software Codesign, CODES/CASHE '97, pp. 43-48.

[28]. Van der Gaag, L.C., "Bayesian Belief Networks: Odds and Ends," *Computer Journal*, Vol. 39, No. 2, 1996, pp. 97-113.

[29]. Wolf, Wayne, "Object-Oriented Cosynthesis of Distributed Embedded Systems," *ACM Transactions on Design Automation of Electronic Systems*, Vol. 1, No. 3, July 1996, pp. 301-331.