Hardware/Software Partitioning Using Bayesian Belief Networks

John T. Olson, Jerzy W. Rozenblit, Claudio Talarico, and Witold Jacak

***Abstract*—In heterogeneous system design, partitioning of the functional specifications into hardware (HW) and software (SW) components is an important procedure. Often, an HW platform is chosen, and the SW 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 HW and SW classifications. The BBN's ability to propagate evidence permits the effects of a classification decision that is 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. A methodology for automatically generating the qualitative structural portion of BBN and the quantitative link matrices is given. 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, which is called model-based codesign.**

***Index Terms*—Hardware/software partitioning, heterogenous system design, model-based codesign.**

I. Introduction

IN THIS, we present a new approach to the hardware (HW)/software (SW) partitioning problem [4]–[6], [8], [21], [24] that uses Bayesian belief networks (BBNs) for functional component classification into HW and SW. Design of heterogeneous systems entails choosing which functional components should be implemented in HW and which should be implemented in SW. Typically, an HW platform is chosen, and the SW is written to make the HW meet the specified requirements. The problem with this approach, however, is that, during system integration, interface and incompatibility problems may arise very late in the design cycle. HW/SW partitioning is used to push the implementation decisions into the early design phases, so that the decision of whether to use HW or SW is not made in isolation for each functional component.

In our previous work, we have established a systematic approach to the 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 of the system model components into HW or SW must be made.

The partitioning approach presented here uses the BBN concept [3], [14], [20], [28]. The reasons for using the BBN framework are the following: 1) its ability 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 toward an HW or SW 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 in 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 formally. We describe the principles of BBNs and then address the existing body of HW/SW partitioning work. Then, we present our formal methodology, followed by an illustrative design example.

II. Motivation

Extensive research has been conducted in the area of HW/SW partitioning. There are three main partitioning approaches: 1) methods in which the partitioning is driven toward a preexisting HW platform; 2) flexible methods that use no fixed preexisting HW platform; and 3) methods that are based on traditional very large scale integration (VLSI) partitioning techniques.

In the methods that drive partitioning onto a preexisting HW platform, the configuration usually consists of a single microprocessor that runs the SW component and an application-specific integrated circuit (ASIC) to implement the HW portion. The work presented in [7], [9], [12], [16], [17], [31], [32], and [34] is among the most notable in this area of HW/SW partitioning. The major limitation to this body of work is the inability to utilize a different HW configuration, particularly, in the cases where the design and fabrication of an ASIC is not feasible because of the cost. Therefore, the need for HW/SW partitioning systems that are flexible with regard to the type and number of components comprising the HW platform is warranted.

Several works have attempted to address the problem of partitioning onto a flexible HW/SW platform. Among them are [2], [10], [13], [15], [23], [25], [29], [30], and [33] that

Manuscript received April 26, 2002; revised June 3, 2005. This work was recommended by Associate Editor K. C. Chang.

J. T. Olson is with IBM Inc., Armonk, NY 10504 USA.

J. W. Rozenblit is with the Department of Electrical and Computer Engineering, University of Arizona, Tucson, AZ 85721 USA (e-mail: jr@ece.arizona.edu).

C. Talarico is with the Department of Electrical Engineering, Eastern Washington University, Cheney, WA 99044 USA.

W. Jacak is with the Department of Software Engineering, Upper Austria University of Applied Science, 4232 Hagenberg, Austria.

Color versions of one or more of the figures in this paper are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/TSMCA.2007.902623

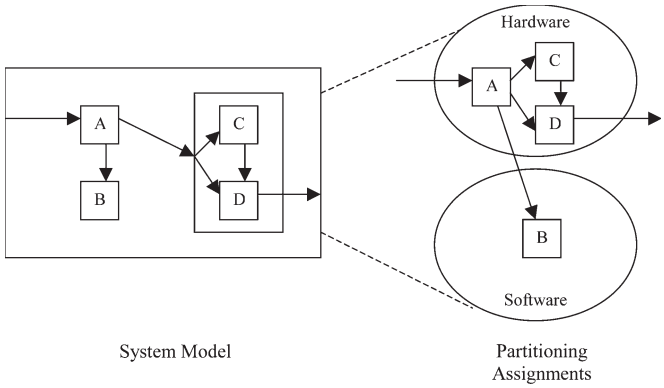


Fig. 1. Illustration of the partitioning problem. The labels A, B, C, and D represent functional elements.

use criticality, dynamic cost functions, HW 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 HW 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 is a useful method to determine the cause of problems during system integration.

Another approach uses VLSI circuit partitioning algorithms with some modifications. Alpert and Kahng [1] provide an excellent review of such partitioning algorithms. Vahid [27] and Maciel *et al.* [35] 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 to the greatest extent. Again, there is no direct way to measure the effect that a partitioning choice has on the other components individually, just the effect on the overall cost.

III. 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, an HW platform was often chosen, and then, SW was written for correcting the inadequacies of the HW. Currently, however, research has progressed from the idea of partitioning HW elements within a VLSI design to that of partitioning a high-level functional model of a system. Fig. 1 shows an example partitioning into HW and SW, with the system model containing four functional components (A, B, C, and D) that are partitioned into HW (A, C, and D) and SW (B).

The HW/SW partitioning methodology that 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 is 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.

The partitioning methodology that is presented here takes the high-level functional description along with parameters that are obtained from simulating the functional model (with a design tool such as StateMate [11]—the primary design tool used in our laboratory) to create the desired BBN representation. The results that are obtained from the simulation experiments of the components that have been classified thus far are then used as evidence and propagated throughout the BBN. The HW and SW functional classifications chosen by the BBN framework are mapped into specific HW and SW components. At this point, the abstract model is mapped onto a collection of interconnected real-world components. This HW/SW partitioning methodology is capable of the following: 1) partitioning onto an HW platform that is determined dynamically and not fixed prior to partitioning; 2) extending partitioning decisions to apply to the entire system; and 3) generation of partitioning representation based on BBNs.

IV. Background

In this section, we provide a brief overview of the BBN approach and the various existing partitioning methods.

A. BBNs

In order to fully understand the complexities of a 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 preexisting 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?"

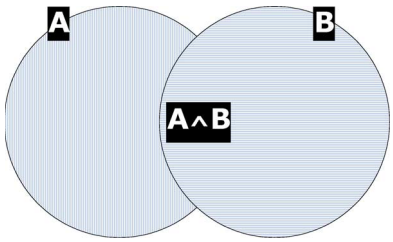
The purpose of BBNs, on the other hand, is to look at the world from a causal point of view. The key role of the tool has changed from that of inferring new knowledge (as is the case with rule-based systems) to that of cause and effect. The causal nature of the domain is captured in the BBN, and the 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?"

The theoretical foundation of probabilistic reasoning lies on the concepts of joint probability, conditional probability, Bayes' theorem, the product rule, and the chain rule. Given statement A, we denote the probability or likelihood that it is true as $P(\mathrm{A})$. A union probability is written as $P(\mathrm{A} \vee \mathrm{B})$ and represent the "probability that either A is true or B is true." A joint probability is written as $P(\mathrm{A} \wedge \mathrm{B})$ or $P(\mathrm{A,B})$ and means

the "probability that both A and B are true." The relationship between A $\wedge$ B and A $\vee$ B is given by the following rule:

$$P(\mathrm{A} \vee \mathrm{B}) = P(\mathrm{A}) + P(\mathrm{B}) - P(\mathrm{A} \wedge \mathrm{B})$$

and it is illustrated graphically by the following diagram.



The notation $P(\mathrm{B}|\mathrm{A})$ is known as conditional probability, and it states the probability of B, given that we already know about A. $P(\mathrm{B}|\mathrm{A})$ is defined by the following rule:

$$P(\mathrm{B}|\mathrm{A}) = \frac{P(\mathrm{B} \wedge \mathrm{A})}{P(\mathrm{A})}.$$

Of course, this rule cannot be used in cases where $P(\mathrm{A}) = 0$. Due to the commutativity of $\wedge$, we can also write

$$P(\mathrm{B} \wedge \mathrm{A}) = P(\mathrm{A} \wedge \mathrm{B}) = P(\mathrm{B}|\mathrm{A}) \cdot P(\mathrm{A}) = P(\mathrm{A}|\mathrm{B}) \cdot P(\mathrm{B}).$$

When written in this form, the former expression is called the multiplication rule. Bayes' theorem is a direct consequence of the multiplication rule and provides the mean to calculate the probability that a certain proposition is true, given that we already know related information. The theorem is stated as follows:

$$P(\mathrm{B}|\mathrm{A}) = \frac{P(\mathrm{A}|\mathrm{B}) \cdot P(\mathrm{B})}{P(\mathrm{A})}.$$

$P(\mathrm{B})$ is called the *prior probability* of B, and $P(\mathrm{B}|\mathrm{A})$, aside from being called the conditional probability, is also known as the *posterior probability* of B. In general, joint statements (beliefs) can be calculated by using the definition of conditional probability and recursively applying Bayes' theorem. The resulting expression is known as the chain rule and is stated as follows:

$$\begin{aligned} &P(E_1, E_2, \ldots, E_{n-1}, E_n) \\ &\quad = P(E_n|E_1, E_2, \ldots, E_{n-1}) \cdot P(E_1, E_2, \ldots, E_{n-1}) \\ &\quad = P(E_n|E_1, E_2, \ldots, E_{n-1}) \cdot P(E_{n-1}|E_1, E_2, \ldots, E_{n-2}) \\ &\qquad \cdot, \ldots, \cdot P(E_3|E_2, E_1) \cdot P(E_2|E_1) \cdot P(E_1). \end{aligned}$$

BBN simply represents an efficient way of storing a joint probability distribution. Formally, a BBN is defined as a directed acyclic graph (DAG), 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 link matrices (also known as conditional probability tables) that are associated with each edge of the DAG and the equations that govern the propagation of evidence (the quantitative part). Together, these two parts provide a system that is capable of translating evidence pertaining to measured results into the probability that a given hypothesis is true.

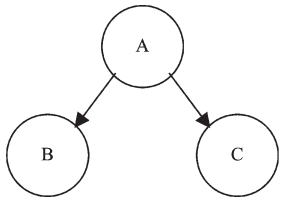


Fig. 2. Simple BBN showing that A has a causal influence over B and C.

$$\mathrm{A} \rightarrow \mathrm{B}$$

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

Fig. 3. Definition for the entries in a link matrix.

In the qualitative portion of a BBN, a directed edge from any node A to another node B (denoted $A \rightarrow B$) represents the causal influence of A over B. Each node within a BBN represents a statistical random variable, which may comprise several hypotheses. Therefore, evidence related to the likelihood 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 probabilistic reasoning can easily understand the causal relationships among the nodes. In Fig. 2, it is easy to see that node A has a causal influence over nodes 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 probabilistic evidence within a BBN. Evidence is a probabilistic 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. 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 in 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, is used to determine the beliefs that are 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 Fig. 2, 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 in Fig. 3 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 HW, given that A is implemented in HW.

All of the messages that interact with the link matrices and the directions in which they flow are shown in Fig. 4.

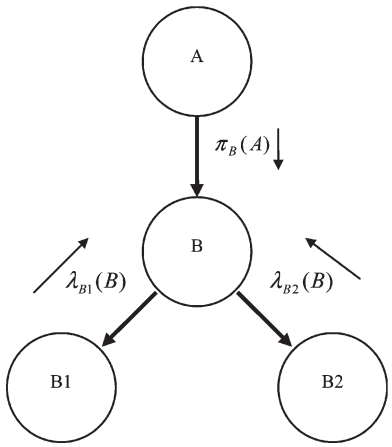


Fig. 4. Small BBN with the messages passed to node B illustrated.

The final step in understanding BBNs consists of knowing how the belief of each node is calculated. The belief of a node is actually the belief in each hypothesis within the node. Fig. 5 shows the simplest kind of BBN structure, i.e., 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."

B. Partitioning Methods

Many previous works present partitioning algorithms that assume a base architecture and then partition according to that architecture. The work of Hendry and Sananikone [12] partitions a solution into an SW component that runs on a single processor and an HW component that consists of either an ASIC or a field-programmable gate array in a system called COSYN. Partitioning begins with a completely SW solution and then iteratively moves nodes into an HW solution utilizing *activation frequency and token residence time* along with HW and SW estimators to determine the amount of speedup from moving a given functional block from SW to HW.

The work of Eles *et al.* [7] is similar, in that partitioning occurs to a single-microprocessor-based SW component and an HW coprocessor. 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 HW implementation, and interprocess 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] uses a single microprocessor paired with an ASIC utilizing a shared memory for a target architecture. Knudsen and Madsen [16] provide 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 HW and SW.

In [15], Kalavade and Subrahmanyam assume the input 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. The 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 HW. 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 HW, then this node is biased toward HW. If the preceding similar node was placed in SW, then this node is biased toward SW. If there is no preceding similar node, then the performance/area ratio is used to carry out the partition.

Henkel and Ernst [13] present a unique method for high-level estimation for HW/SW partitioning. In this paper, the effort is placed on two high-level estimation techniques: 1) HW effort and 2) HW/SW 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, multiplexers, and the control unit. The overall HW effort is then calculated by adding the HW efforts of each component listed previously. The approach used in the HW/SW communication estimation assumes that SW is running on a single processor core and that it must stop and send information to the HW components when needed. Therefore, the overhead in cost comes from data being sent from SW to shared memory and then from shared memory to HW. All of the estimation techniques are combined into a dynamically weighted cost function where the constant attached to the HW area increases as the difference between the system time and the constraint time decreases. In this way, the HW area component becomes more important as the system time approaches its maximum allowable value.

Gupta and De Micheli [10] compile an HardwareC description to produce graphs that are similar to data flow graphs where vertices represent operations and edges represent either data or sequencing dependence between vertices. The first step of partitioning is to partition according to points of nondeterminism; external points of nondeterminism (caused by external input/output operations) are assigned to HW, and internal points of nondeterminism (caused by internal data-dependent operations) are assigned to SW. Following the initial partitioning (assuming feasibility), operations are migrated from HW to SW in search of a lower cost partition.

The work of Wolf [29] uses object-oriented techniques for partitioning. It begins with a method data flow graph, execution rates, and a library of available components. Initially, all methods and variables that are associated with an object are greedily assigned to their own processing elements that are as fast as the fastest method that is 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

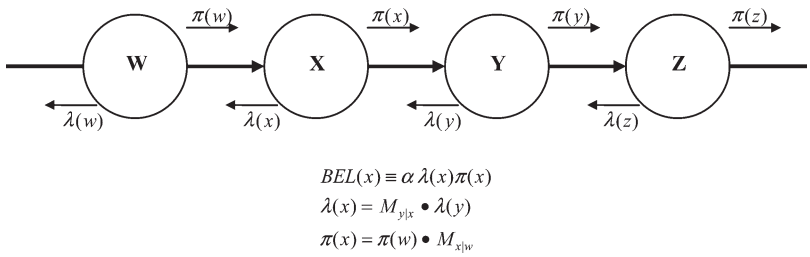


Fig. 5. Actual calculations that must be made to determine the belief of X.

thus removing the need for it, or by moving methods 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.

Alpert and Kahng [1] provide 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 intergroup communication must be kept to a minimum. Among these classes of partitioning algorithms are the following: 1) move-based approaches such as greedy and iterative exchange algorithms; 2) geometric approaches such as vector partitioning; 3) combinatorial approaches such as max-flow min-cut; and 4) clustering-based approaches [1].

V. BBN-Based Partitioning Methodology

As we have stated in Section I, the HW/SW partitioning methodology presented here is part of a larger design context that is known as model-based codesign [21], [24].

In traditional HW and SW design, decisions regarding partitioning the HW and SW occur at the beginning of the design process. SW and HW components are designed separately and are later integrated. The difficulty with this type of design scenario is that there are often compatibility and timing problems that are 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 model-based 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 Fig. 6). Because of this, there are fewer integration problems, and the ways in which HW and SW interact in the final solution are better known. By knowing SW/HW interactions, the design space can be better optimized in terms of HW area, which in turn helps with space-constrained embedded systems.

The HW/SW partitioning methodology presented here takes an executable functional model (including design characteristics of the functional components) and produces a partitioned model, as shown in Fig. 7.

There are four basic steps of the partitioning methodology: 1) generation of the BBN; 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 described in [20]); and 4) classification of each functional component into HW or SW, if possible. The decision whether to classify a functional component into HW or SW can be made based on the degree of belief for each assignment, as given in

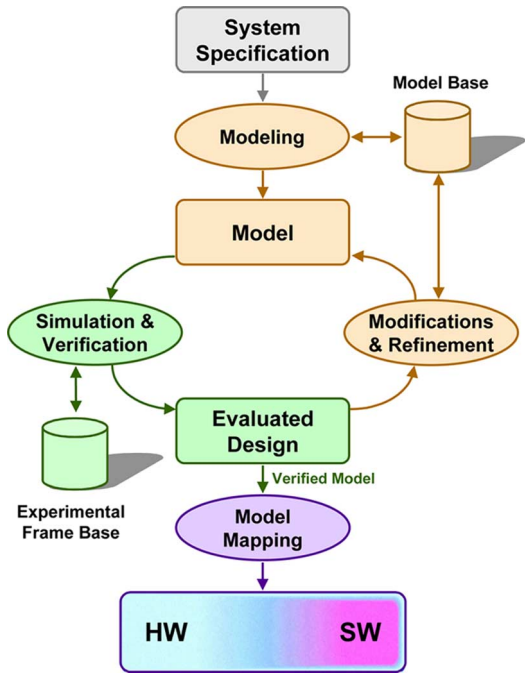


Fig. 6. HW/SW codesign methodology.

the belief probabilities that are associated with each node of the BBN.

In the first step of the methodology, i.e., 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 a "flat" BBN structure. The values for complexity, bandwidth, and frequency are combined to construct the probability matrix that is associated with each causal link.

Once the representative BBN has been constructed, evidence that is output from the simulation of the model is propagated throughout the network. Given a specific implementation of a component (HW or SW), we use simulation to estimate the performance indexes of interest (e.g., response time and throughput) of the component. The information collected is transformed into evidence by comparing the estimated performances. Let us say that the performance of interest is response time, and the response time estimated through simulation is $\mathrm{RT_{sw}}$ for the SW implementation and $\mathrm{RT_{hw}}$ for the HW implementation; then, the evidence for the given component is computed as follows:

$$\lambda = \begin{bmatrix} \mathrm{RT_{hw}}/(\mathrm{RT_{sw}} + \mathrm{RT_{hw}}) \\ \mathrm{RT_{sw}}/(\mathrm{RT_{sw}} + \mathrm{RT_{hw}}) \end{bmatrix}.$$

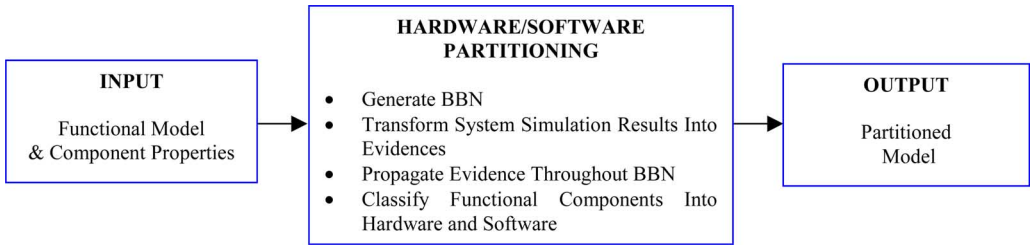


Fig. 7. BBN-driven HW/SW partitioning methodology.

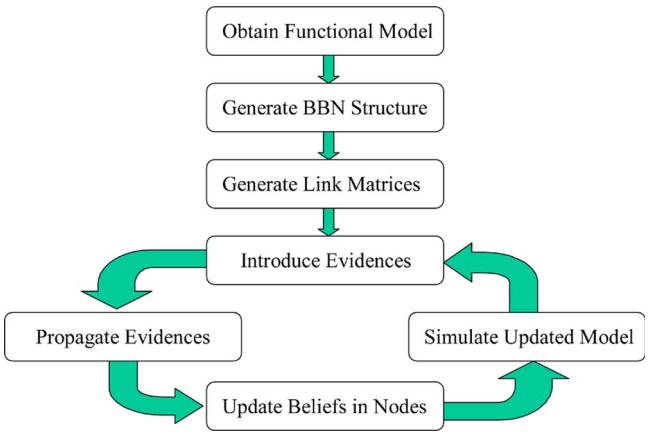


Fig. 8. BNN-based partitioning revisited.

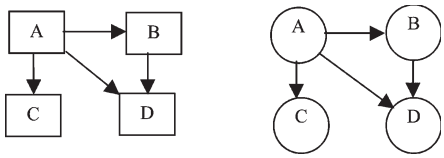


Fig. 9. Typical nonhierarchical functional model with corresponding BBN.

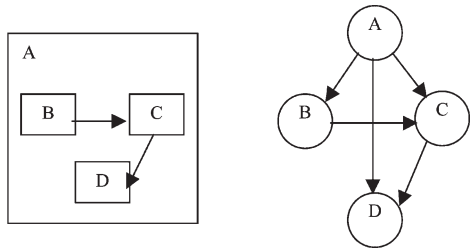


Fig. 10. Typical layer of abstraction with corresponding BBN representation.

The model used during simulation is updated according to the latest belief values, e.g., if the belief that a functional component is 75% in agreement with an HW solution, then that component is modeled as HW in the simulation. The initial values for the beliefs that are associated with each node of the BBN are set to 50% HW and 50% SW. As evidence is introduced, the beliefs change according to the rules of evidence propagation, as set forth by [20]. This process in continued in an iterative cycle, as illustrated in Fig. 8.

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 that are associated with each node are examined. If there is a clear indication that either HW or SW should be used for the component (a belief having a value of 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 HW belief and SW belief is < 0.2), then it does not matter which partition is chosen. In this case, the designer should intervene and make the decision based on other criteria, for instance, the area of the circuit.

A. BBN Generation

Fig. 9 shows a typical nonhierarchical functional model with its corresponding BBN representation. The functional model representation that 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. The arrows in the functional model of Fig. 9 represent coupling constraints. In the BBN of Fig. 9, 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.

In choosing the BBN representation of an abstract level within a functional model, the meaning of how the components within the level are affected by their interactions was very important. Fig. 10 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 subcomponents. 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 subcomponents.

Fig. 11 shows how components that are outside of a level of abstraction can influence both the abstract level as a whole and the individual subcomponents. 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). Fig. 11 also shows how an outside component H can have causal influence over an entire abstract level A and over an individual subcomponent G.

The generation of a BBN structure involves taking the functional model in the form of a set of functional nodes and converting it first into a hierarchy tree. After conversion into the hierarchy tree, the set of vertices and edges of the BBN is then generated. The graph-based algorithm for carrying out the conversion is presented in the Appendix.

The functional model and its corresponding BBN are shown in Fig. 11. Fig. 12 portrays the same functional model and its hierarchy tree.

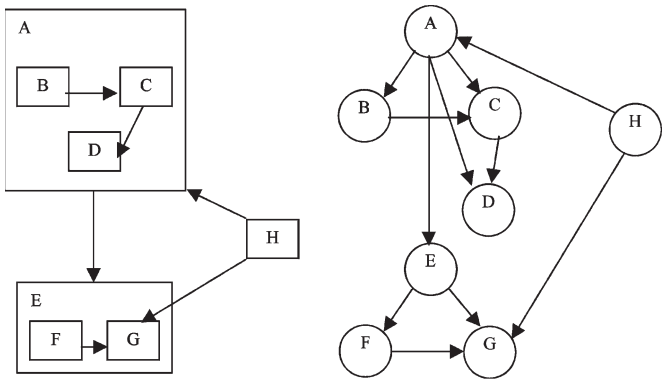


Fig. 11. (Left) Example functional model containing causal links crossing abstraction boundaries and (right) corresponding BBN representation.

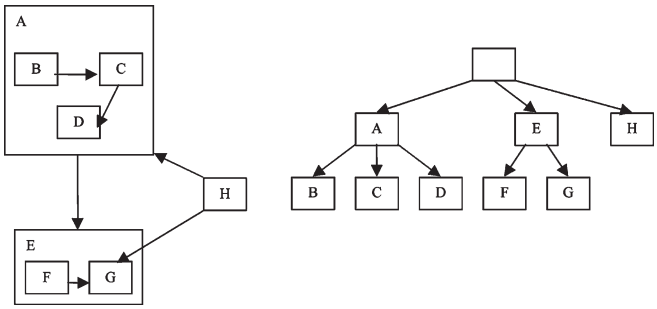


Fig. 12. (Left) Functional model with (right) associated hierarchy tree.

B. Generation of BBN Link Matrices

In order to calculate quantifiable values such as those contained within the link matrices of a BBN, we must be given some quantifiable input that is related to the operation of each functional component. Therefore, we have chosen to characterize each functional component with three types of measurement: 1) the *complexity* of the functional component; 2) the total *bandwidth* that is associated with the individual functional component (in bits per second); and 3) the *frequency of execution* of the functional component (in executions per second). Each of these measurements is assumed to be available as output from the functional simulation of the design model.

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] (recall that we use StateChart as the modeling representation and StateMate as the design tool) 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 measures of costs such as the ones shown in the following to weigh 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 arithmetic operation: Cost = 3. (Add, subtract, etc.)
- Execute a medium complexity calculation: Cost = 6. (Multiply, divide, etc.)
- 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 that are needed to complete the given operation on a basic microprocessor such as the Motorola 68000.

The next two measurements (bandwidth within a single functional component and frequency of execution) are obtainable from a simulation of the functional model (in our case, using StateMate).

Although we have chosen to use complexity, bandwidth, and frequency as our metrics of choice, there are obviously several different measures that could have been chosen. Gajski *et al.* [8] show that some of the most popular types of metrics include HW area, delay, and power consumption, just to name a few. They also show that closeness metrics are useful when no partition yet exists. Relative complexity, relative bandwidth, and relative frequency act as closeness metrics that we use to construct the BBN representation before any partitioning takes place.

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 that are relative to the influenced functional components that are of importance. Thus, we use the following equation as a measurement for how "similar" the complexities of two functional components are (which we call the relative complexity):

$$rel\_comp(a,b) = \frac{1}{\log_{10}\left(\max\left(\frac{comp(a)}{comp(b)}, \frac{comp(b)}{comp(a)}\right)\right) + 0.1}. \quad (1)$$

What the preceding equation means is that the relative complexity between two functional components is inversely proportional 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 the following:

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

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

Now that we know how each functional component relates to one another, the final step is to find a suitable matrix representation. In order for the link matrix to properly represent

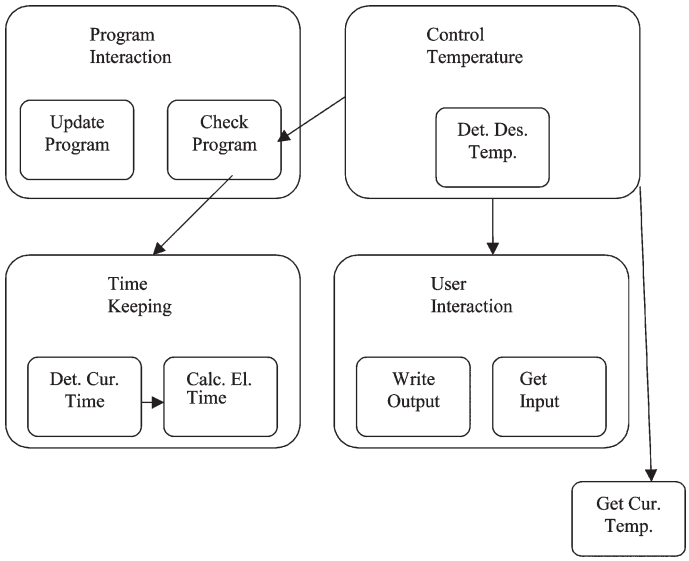


Fig. 13. Functional model for programmable thermostat.

beliefs, it must be guaranteed that the summation of each column in the matrix equals 1. To that purpose, the link matrix is constructed as follows:

$$M_{b|a} = \frac{\alpha}{N_\alpha}\begin{bmatrix} rel\_comp(a,b) & \frac{1}{rel\_comp(a,b)} \\ \frac{1}{rel\_comp(a,b)} & rel\_comp(a,b) \end{bmatrix} + \frac{\beta}{N_\beta}\begin{bmatrix} rel\_band(a,b) & \frac{1}{rel\_band(a,b)} \\ \frac{1}{rel\_band(a,b)} & rel\_band(a,b) \end{bmatrix} + \frac{\gamma}{N_\gamma}\begin{bmatrix} rel\_freq(a,b) & \frac{1}{rel\_freq(a,b)} \\ \frac{1}{rel\_freq(a,b)} & rel\_freq(a,b) \end{bmatrix} \quad (4)$$

where N_α, N_β, and N_γ are normalization factors, while the coefficients α, β, and γ provide system designers the chance to express that one or more of the three measurements are more critical than the other(s). The normalization factors are given by

$$N_\alpha = (\alpha+\beta+\gamma)\cdot\left(rel\_comp(a,b) + \frac{1}{rel\_comp(a,b)}\right)$$
$$N_\beta = (\alpha+\beta+\gamma)\cdot\left(rel\_band(a,b) + \frac{1}{rel\_band(a,b)}\right)$$
$$N_\gamma = (\alpha+\beta+\gamma)\cdot\left(rel\_freq(a,b) + \frac{1}{rel\_freq(a,b)}\right). \quad (5)$$

VI. Partitioning Example

In this section, we present a programmable thermostat design example. Fig. 13 shows the functional model used in this example. Fig. 14 gives a block representation of the functional model that will make the generation of the structural portion of the BBN easier to follow.

Fig. 15(a) shows the two levels of abstraction of the functional model that include five components: one that contains no subcomponents (L) and four components that act as abstraction modules. All links between the five top-level components are shown in Fig. 15(b), 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.

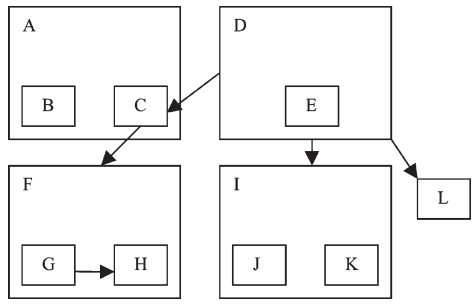


Fig. 14. Block representation of the functional model.

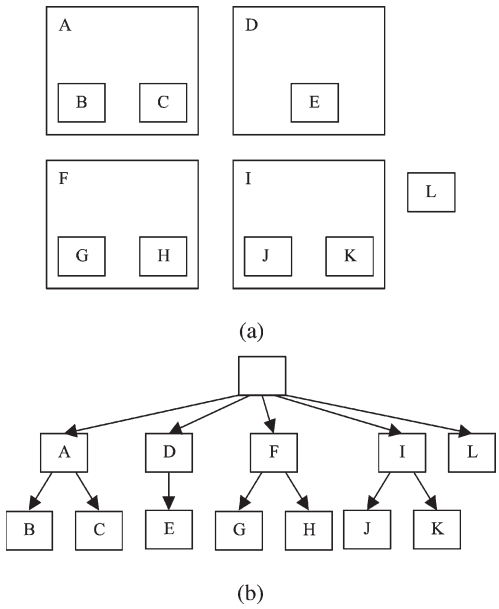


Fig. 15. Hierarchical elements from (a) functional model and (b) corresponding hierarchy tree.

The second iteration of the BBN structural generation algorithm brings about many new nodes corresponding to the subcomponents of "A," "D," "F," and "I." Fig. 16 shows the system model at the first two levels of abstraction and the matching

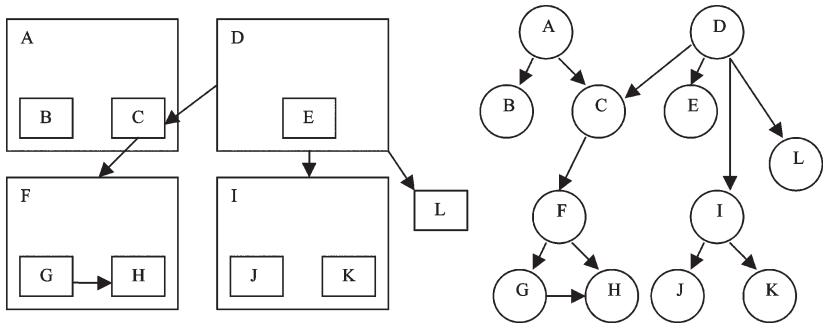


Fig. 16. After the second iteration of the algorithm, both levels of the (left) functional model with the (right) corresponding BBN.

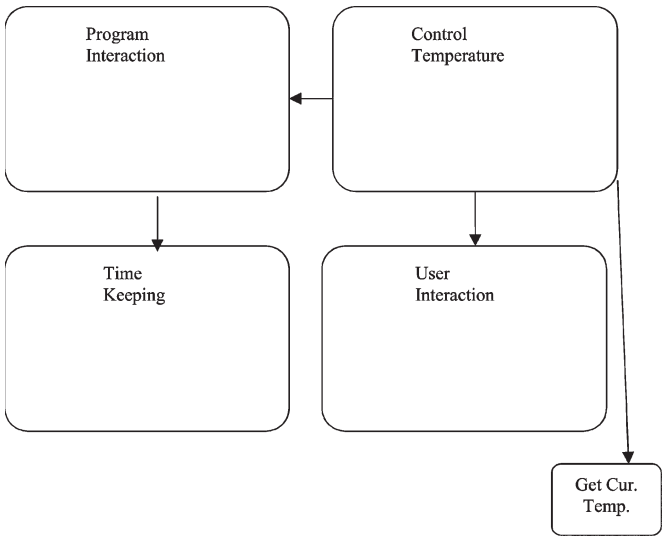


Fig. 17. Reduced functional model for programmable thermostat.

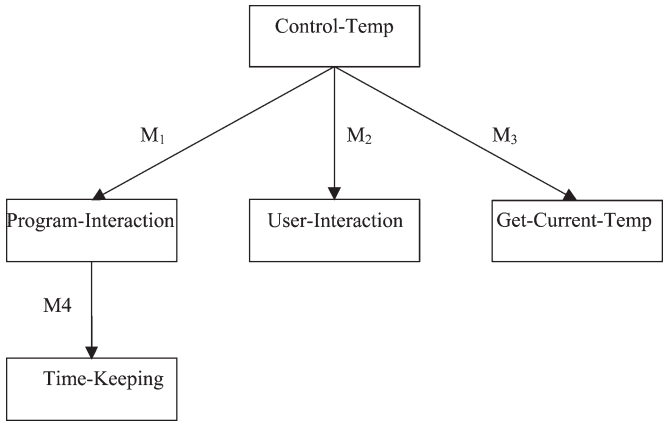


Fig. 18. BBN representation of reduced functional model.

BBN. Note that links have been added for both the internode connections between nodes of the same abstraction level and the links from the abstract parent nodes to the subordinate nodes representing their subcomponents. Since this functional model only contains two levels of abstraction, the algorithm stops at this point.

For the generation of the link matrices and the introduction of evidences, we have chosen to use a reduced functional model shown in Fig. 17. This will allow the reader to better understand the effects of evidence propagation.

The final result of the BBN structural generation algorithm for the reduced node model is shown in Fig. 18.

In order to find the complexity, bandwidth, and frequency measures of each functional component, a StateMate design representation was created and simulated. To demonstrate how the measures were calculated, consider the Control-Temperature functional component. First, we calculate the complexity. We have taken the embedded code in the Control-Temperature StateChart and placed the associated complexity values for each line next to the line in Fig. 19 (this code is generated by StateMate). From adding the numbers in Fig. 19 and multiplying by the number of states (four states in the StateChart representation), we obtain 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, multiply each by the size of the data in bits, and multiply each by the number of times that data pass through the particular data or control line per second to obtain a value with units of bits per 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. In our design models, there are eight control lines connected to Control-Temperature, each with an event occurring about once a second for a total of 256 bits/s. The total bandwidth associated with Control-Temperature is 768 bits/s, as calculated from the StateMate model.

To determine the frequency of execution, we find the fastest executing portion of the StateChart associated with the functional model. For Control-Temperature, this corresponds to the time that it takes to take in and transmit a program, i.e., fives times a second. All of the values just calculated are given in Fig. 20, along with the values for the other functional components.

For this example, we have chosen the following weights: $\alpha = 3$, $\beta = 1$, and $\gamma = 3$ (these are the designer's preferences). We obtain the set of link matrices shown in Fig. 21.

In the formulation of these matrices, the emphasis (designer's preference) was placed on complexity and frequency. This can be seen particularly 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 thermostat design example. We use a BBN calculation tool to perform the propagation of evidence and update of beliefs

```
/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
```

Fig. 19. 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

Fig. 20. Table of complexity, bandwidth, and frequency of execution for functional components in programmable thermostat example.

called Hugin Lite, which is a shareware version of the Hugin System by Hugin Expert A/S. Initially, all beliefs are equal, but let us say that we introduce evidence to "Get-Current-Temp" that it is 83% likely that it should be implemented in HW. Fig. 22 illustrates how this evidence affects the entire BBN.

Because of the vast differences in the link matrix values between the "Get-Current-Temp" and "Control-Temp" functional components, the introduction of this evidence actually drives "Control-Temp" toward an SW implementation. Next, we introduce evidence that "Program-Interaction" has a 65% chance that it should be implemented in SW, and the result is shown in Fig. 23.

Here, we see that, although "Program-Interaction" has been driven toward an SW implementation, "Time-Keeping" has been driven slightly toward HW. This difference can be directly attributed to the discrepancies between the two functional components, as illustrated in Fig. 21, where the link matrix shows that any influence from "Program-Interaction" drives "Time-Keeping" in the opposite direction by a 0.52/0.48 ratio.

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 each function should be implemented in HW or SW. This decision is based on the amount of belief that is 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 HW or SW is reflected in the system model, and the process continues until all functions can be classified into either HW or SW.

VII. Summary

In this paper, we have introduced a new methodology for HW and SW partitioning. We have shown how BBNs can be used to

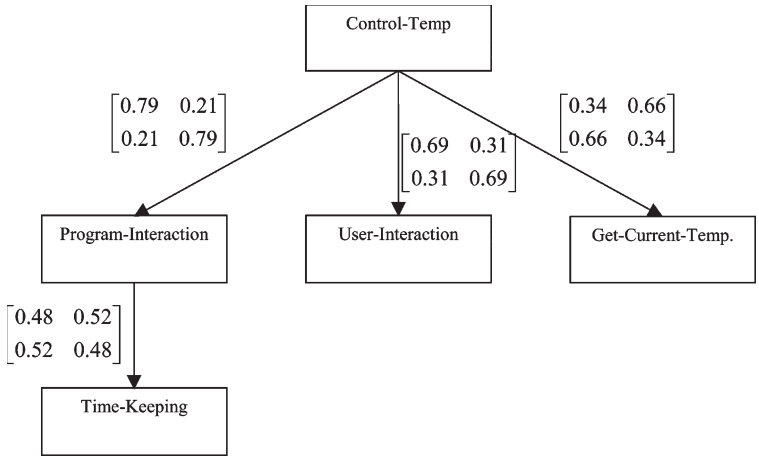


Fig. 21. BBN for programmable thermostat after the generation of the link matrices.

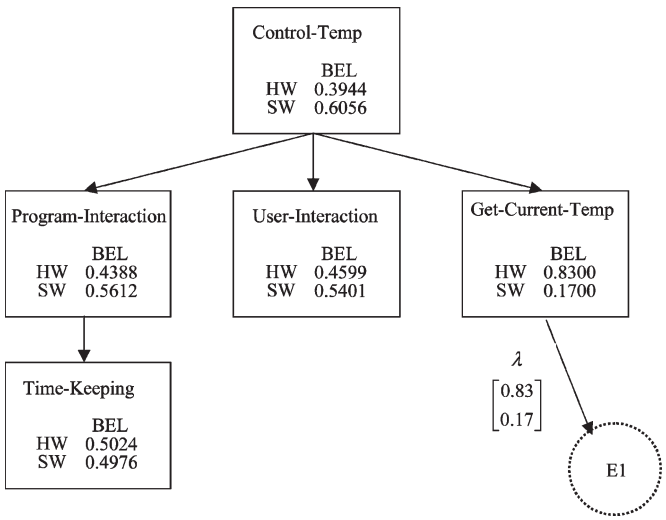


Fig. 22. Beliefs associated with each functional component after the introduction of the first piece of evidence.

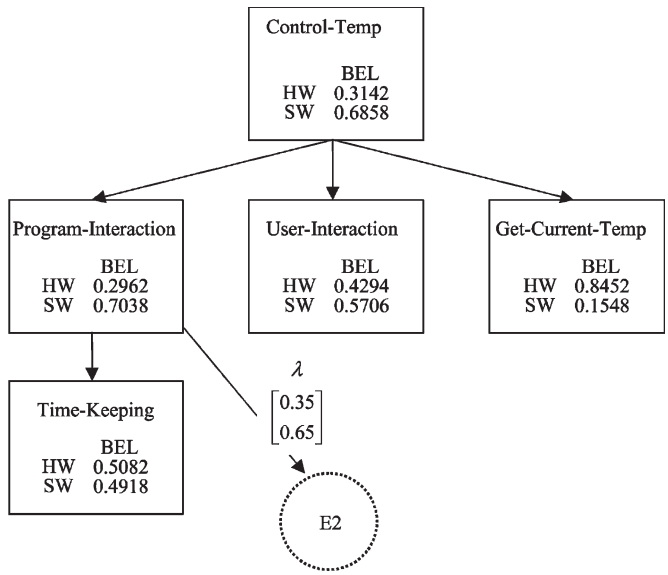


Fig. 23. Beliefs for the functional components after the introduction of a second piece of evidence.

propagate evidence regarding the classification of functions into HW or SW realizations. This propagation permits the effects of a classification decision that is 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 HW or SW. Knowing that a function with, for instance, 75% HW belief should be implemented 75% of the time in HW allows the user to have a measure of the appropriateness of their solution.

This paper also introduced a methodology for the generation of both the qualitative and quantitative portions of a BBN in the HW/SW 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 that are connected by causal links.

The limitations of our work are given as follows: 1) the accuracy and convergence of the method are bounded by the precision and execution time of the simulations that are run to determine the relative properties of the various components, and thus, the fidelity of partitioning decisions is dependent on the validity of the simulation models; 2) the generation of the BBN representation may not be trivial to manage by design engineers who are not familiar with the tenets of belief networks; and 3) other metrics for link matrix generation should be explored.

Future work should include testing other metrics and combinations of other metrics for the link matrix generation. We would also like the ability to incorporate the BBN partitioning methodology into an integrated computer-aided design system for HW/SW codesign.

APPENDIX

This appendix illustrates the algorithm for generating the BBN. The following definitions will be of use in the presentation of the algorithm that follows:

(BBN) a directed acyclic graph G consisting of a set of vertices $V[G]$ and a set of edges $E[G]$, where $E[G]$ is defined as $E[G] \subseteq V[G] \times V[G]$;

(Functional model) a hierarchical system representation consisting of an array FM of *functional nodes*, where each

member can be accessed by an integer between 1 and the number of elements in the array;

(Functional node) a node within a *functional model* consisting 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 *functional 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 *functional node* that is influenced by the given *functional node* identified by *name*, where each member can be accessed by an integer between 1 and *num_children*. The children of a functional node can be only the children on the same or higher level of abstraction.
- *num_children:* an integer giving the number of entries in the *children* array.
- *abs_children:* an array of *names* of the *functional nodes* that are contained in the abstraction level below the current *functional 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.

(Hierarchy tree) a general tree T consisting of a set of vertices $V[T]$ and a set of edges $E[T]$, where $E[T]$ is defined as $E[T] \subseteq V[T] \times V[T]$. The root vertex represents level 0, with the name equal to the string ("root"), and is only meant as an anchor for the *functional nodes* of the *functional model*.

The algorithm that is responsible for generating the set of vertices V and the set of directed edges E for the BBN graph G is shown here.

BBN-GENERATE(FM)

```
V[G] ← ∅; //The set of vertices for the BBN is
          initialized to empty
E[G] ← ∅; //The set of directed edges for the BBN
          is initialized to empty
V[T] ← ∅; //The set of vertices for the hierarchy
          tree is initialized
// to empty
E[T] ← ∅; //The set of directed edges for the
          hierarchy tree is
// initialized to empty
// Find the deepest level of abstraction
deepest_level ← 0;
for i ← 1 to length[FM] do
  if (FM[i].level > deepest_level) then
    deepest_level ← FM[i].level;
// Create the root vertex and add it to hierarchy
  tree T
V[T] ← V[T] ∪ CREATE − VERTEX(root);
// For each functional node of level 1 add a vertex v
  and a directed
// edge e = (v, root) between vertex and the root of
          the hierarchy tree T
for i ← 1 to length[FM] do
  if (FM[i].level == 1) then
  begin
    v ← FM[i].name;
    V[T] ← V[T] ∪ CREATE − VERTEX(v);
    E[T] ← E[T] ∪ CREATE − EDGE(root, v);
  end if;
// Traverse the functional model FM, level by level,
  until all children
// of all nodes have been added to V[T] and the
  associated edges to E[T].
i ← 1;
while (i < deepest_level) // Try all levels but
deepest
begin
  for j ← 1 to length[FM; //step through FM
    if (fm[j].level == i) then
    begin
      u ← fm[j].name;
      for k ← 1 to fm[j].num_abs_children do
      begin
        v ← fm[j].abs_children[k];
        V[T] ← V[T] ∪ CREATE − VERTEX(v);
        E[T] ← E[T] ∪ CREATE − EDGE(u, v);
      end for;
    end if;
  i ← i + 1;
end while;
// At this point the hierarchy tree has beenconstructed.
// Copy the vertices and edges for the hierarchy
  tree to the set of
// vertices and edges for the BBN
V[G] ← V[T];
E[G] ← E[T];
// Remove the root node and the associated edges
  from the BBN graph
V[G] ← V[G] − root;
  for i ← 1 to length(FM) do
    if (FM[x].level == 1) then
      v ← FM[i].name;
      E ← E − (root, v);
// Now, the edges not associated with hierarchy are
  added by
// stepping through the functional model, until all
  influence related
// edges have been added
for i ← 1 to length(FM) do
begin//Step through FM
  u ← fm[i].name;
    for j ← 1 to fm[i].num_children do
    begin//Step through children
    v ← fm[i].children[j];
    E[G] ← E[G] ∪ CREATE − EDGE(u, v); //Add edge
  end for;
end for;
return (V[G], E[G]);
```

REFERENCES

[1] C. J. Alpert and A. B. Kahng, "Recent directions in netlist partitioning: A survey," *Integr. VLSI J.*, vol. 19, no. 1/2, pp. 1–81, Aug. 1995.

[2] V. Catania, M. Malgeri, and M. Russo, "Applying fuzzy logic to codesign partitioning," *IEEE Micro*, vol. 17, no. 3, pp. 62–70, May/Jun. 1997.

[3] E. Charniak, "Bayesian networks without tears," *AI Mag.*, vol. 12, no. 4, pp. 50–63, Winter 1991.

[4] M. Chiodo, P. Giusto, A. Jurecska, H. C. Hsieh, A. Sangiovanni-Vincentelli, and L. Lavagno, "Hardware-software codesign of embedded systems," *IEEE Micro*, vol. 14, no. 4, pp. 26–36, Aug. 1994.

[5] S. J. Cunning, T. C. Ewing, J. T. Olson, J. W. Rozenblit, and S. Schulz, "Towards an integrated, model-based codesign environment," in *Proc. IEEE Conf. and Workshop ECBS*, 1999, pp. 136–143.

[6] G. De Micheli and R. K. Gupta, "Hardware/software co-design," *Proc. IEEE*, vol. 85, no. 3, pp. 349–365, Mar. 1997.

[7] P. Eles, Z. Peng, K. Kuchcinski, and A. Doboli, "Hardware/software partitioning with iterative improvement heuristics," in *Proc. 9th Int. Symp. Syst. Synthesis*, 1996, pp. 71–76.

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

[9] J. Grode, P. V. Knudsen, and J. Madsen, "Hardware resource allocation for hardware/software partitioning in the LYCOS system," in *Proc. DATE*, 1998, pp. 22–27.

[10] R. K. Gupta and G. De Micheli, "System-level synthesis using re-programmable components," in *Proc. Eur. Conf. Des. Autom.*, 1992, pp. 2–7.

[11] D. Harel, "STATEMATE: A working environment for the development of complex reactive systems," *IEEE Trans. Softw. Eng.*, vol. 16, no. 4, pp. 403–414, Apr. 1990.

[12] D. C. Hendry and D. S. Sananikone, "Hardware/software partitioning of embedded systems with multiple hardware processes," *Proc. Inst. Electr. Eng.—Computers Digital Techniques*, vol. 144, no. 5, pp. 285–294, Sep. 1997.

[13] J. Henkel and R. Ernst, "High-level estimation techniques for usage in hardware/software co-design," in *Proc. ASP-DAC*, 1998, pp. 353–360.

[14] F. V. Jensen, *An Introduction to Bayesian Networks*. London, U.K.: UCL Press, 1998.

[15] A. Kalavade and P. A. Subrahmanyam, "Hardware/software partitioning for multifunction systems," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 17, no. 9, pp. 819–837, Sep. 1998.

[16] P. V. Knudsen and J. Madsen, "Integrating communication protocol selection with partitioning in hardware/software codesign," in *Proc. 11th Int. Symp. Syst. Synthesis*, 1998, pp. 111–116.

[17] M. L. Lopez, C. A. Iglesias, and J. C. Lopez, "A knowledge-based system for hardware-software partitioning," in *Proc. DATE*, 1998, pp. 914–915.

[18] M. P. Barros and W. Rosenstiel, "A Petri net based approach for performing the initial allocation in hardware/software codesign," in *Proc. IEEE Int. Conf. Syst., Man, Cybern.*, Oct. 1998, vol. 1, pp. 505–510.

[19] J. T. Olson and J. W. Rozenblit, "Framework for hardware/software partitioning utilizing Bayesian belief networks," in *Proc. IEEE Int. Conf. Syst., Man, Cybern.*, Oct. 1998, vol. 4, pp. 3983–3988.

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

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

[22] J. S. Russell and P. Norvig, *Artificial Intelligence: A Modern Approach*. Upper Saddle River, NJ: Prentice-Hall, 1995.

[23] D. Saha, R. S. Mitra, and A. Basu, "Hardware software partitioning using genetic algorithm," in *Proc. 10th Int. Conf. VLSI Des.*, 1997, pp. 155–160.

[24] S. Schulz, J. W. Rozenblit, M. Mrva, and K. Buchenrieder, "Model-based codesign: The framework and its application," *Computer*, vol. 3, no. 8, pp. 60–67, Aug. 1998.

[25] V. Srinivasan, S. Radhakrishnan, and R. Vemuri, "Hardware software partitioning with integrated hardware design space exploration," in *Proc. DATE*, 1998, pp. 28–35.

[26] D. E. Thomas, J. K. Adams, and H. Schmit, "A model and methodology for hardware-software codesign," *IEEE Des. Test Comput.*, vol. 10, no. 3, pp. 6–15, Sep. 1993.

[27] F. Vahid, "Modifying min-cut for hardware and software functional partitioning," in *Proc. 5th Int. Workshop CODES/CASHE*, 1997, pp. 43–48.

[28] L. C. Van der Gaag, "Bayesian belief networks: Odds and ends," *Comput. J.*, vol. 39, no. 2, pp. 97–113, 1996.

[29] W. Wolf, "Object-oriented cosynthesis of distributed embedded systems," *ACM Trans. Des. Autom. Electron. Syst.*, vol. 1, no. 3, pp. 301–331, Jul. 1996.

[30] G. Stitt, R. Lysecky, and F. Vahid, "Dynamic hardware/software partitioning: A first approach," in *Proc. IEEE DAC*, 2003, pp. 250–255.

[31] G. N. Khan and M. Jin, "A new graph structure for hardware-software partitioning of heterogeneous systems," in *Proc. IEEE Can. Conf. Electr. and Comput. Eng.*, 2004, pp. 229–232.

[32] R. Lysecky and F. Vahid, "A configurable logic architecture for dynamic hardware/software partitioning," in *Proc. DATE*, 2004, pp. 480–485.

[33] P. Waldeck and N. Bergmann, "Dynamic hardware-software partitioning on reconfigurable system-on-chip," in *Proc. IEEE Syst.-on-Chip Real Time Appl.*, 2003, pp. 102–105.

[34] P. Arato, S. Juhasz, Z. A. Mann, A. Orban, and D. Papp, "Hardware–Software partitioning in embedded system design," in *Proc. IEEE Symp. Intell. Signal Process.*, 2003, pp. 197–202.

[35] F. C. Filho, P. Maciel, and E. Barros, "A Petri net approach for hardware/software partitioning," in *Proc. Symp. Integr. Circuits and Syst. Des.*, 2001, pp. 72–77.

John T. Olson received the Ph.D. degree in electrical and computer engineering from the University of Arizona, Tucson, in 2000.

Since then, he has been with IBM Inc., Armonk, NY. He is currently the Lead Software Engineer for IBM's Centralized Virtual Tape Products. He has filed several patents in the areas of automated error recovery, configurable error generation, and error notification in computer systems.

Jerzy W. Rozenblit received the Ph.D. and M.S. degrees in computer science from Wayne State University, Detroit, MI, and the M.Sc. degree in computer engineering from the Technical University of Wroclaw, Wroclaw, Poland.

He is currently a Professor and the Head of the Department of Electrical and Computer Engineering, University of Arizona, Tucson. He was a Fulbright Senior Scholar and a Visiting Professor at the Institute of Systems Science, Johannes Kepler University, Linz, Austria, and a Fulbright Senior Specialist at the Cracow University of Technology, Cracow, Poland. He has held visiting professorship appointments at the Technical University of Munich, Munich, Germany; Central Research Laboratories of Siemens AG; and Infineon Technologies AG, Munich. Among many sponsors, his research in design has been supported by the National Science Foundation, the U.S. Army, Siemens AG, and NASA. His research and teaching interests include complex systems design and simulation modeling. He serves as an Associate Editor of the *ACM Transactions on Modeling and Computer Simulation*, an Area Editor of the *Transactions of the Society of Modeling and Simulation International*, and an Executive Board Member of the IEEE Technical Committee on Engineering of Computer-Based Systems.

Claudio Talarico received the M.S. degree from Universitá degli Studi di Genova, Genova, Italy, and the Ph.D. degree from the University of Hawaii at Manoa, Honolulu, both in electrical engineering.

He is currently an Assistant Professor of electrical engineering in the Department of Electrical Engineering, Eastern Washington University, Cheney. Before joining Eastern Washington University, he held various management and technical positions in the area of digital integrated systems design at Infineon Technologies, Sophia Antipolis, France; Ikos Systems, Cupertino, CA (now part of Mentor Graphics); and Marconi Telecommunication, Genova. His research interests include design methodologies for integrated circuits and systems with emphasis on system-level design; embedded computing systems; hardware–software codesign; low-power design; system specification languages; and early design assessment, analysis, and refinement of complex systems on chips.

Witold Jacak received the M.S.E. degree in electronics and the Ph.D. degree in control and system engineering from the Technical University of Wroclaw, Wroclaw, Poland, in 1973 and 1978, respectively, and the Habilitation degree in intelligent multiagent systems from the Technical University of Warsaw, Warsaw, Poland, in 1992.

He was with the Institute of Technical Cybernetics, Technical University of Wroclaw, as a Professor Assistant from 1977 to 1989 and as a Professor from 1989 to 1995. From 1991 to 1998, he was a Guest Professor at the University of Linz, Linz, Austria. Since 1994, he has been the Head of the Information Technology Faculty, Department of Software Engineering, Upper Austria University of Applied Science, Hagenberg. His research interests include artificial and computational and distributed intelligence, multiagent autonomous systems, cognitive modeling and simulation, and knowledge engineering.

Dr. Jacak is a member of editorial boards and a Council Member of several national and international committees.