SLIF: A Specification-Level Intermediate Format for System Design

Frank Vahid
Department of Computer Science
University of California, Riverside, CA 92521

Daniel D. Gajski
Department of Information and Computer Science
University of California, Irvine, CA 92717

Abstract

As methodologies and tools for chip-level design mature, design effort becomes focused on higher abstraction levels. Presently, much effort is focused on system-level design, where the key tasks include system component allocation, functional partitioning and transformation, and coarse estimation. However, commonly-used internal formats of functionality, such as the control-dataflow graph, are too fine-grained for the system level. We introduce a more abstract format, and we demonstrate its order-of-magnitude more efficient support of system design tasks and its support of practical designer interaction. The format is used by the SpecSyn system design environment, and can be extended to handle many new system design problems.

1 Introduction

The maturation of behavioral synthesis tools, which generate a register-transfer structure for a given behavior, has lead to new system-design research efforts. System design involves several highly-interdependent tasks, which must be performed before behavioral synthesis or software development take place, including: (1) Allocation of system components, such as processors, ASICs, memories and buses, to the design; (2) Partitioning of the functional specification among those components; (3) Transformation of the specification into one more suited for synthesis or compilation, such as sequentializing processes for implementation with a single processor; (4) Coarse estimation of constrained design metrics, including performance, size, pins, power, design time, modifiability, and cost. After completing these tasks, the result is an interconnection of functionally-specified system components. The functionality assigned to each ASIC or processor component can be implemented through behavioral synthesis or software design, respectively.

In current practice, system design tasks are vaguely defined, decisions are based on mental or hand-calculated estimations, and documentation of decisions is scarce. These shortcomings have led researchers to propose starting from a simulatable functional specification, which is then partitioned among system components with the aid of tools. Most such approaches first convert the specification to an internal format, on which system design tasks are then applied.

By describing the SLIF internal format in this paper, we hope to encourage comparison of various formats, to encourage new solutions to system design tasks, and to enable uniform comparison of research results. SLIF is unique from existing formats due to its orientation around accesses among computations rather than around dependencies, its representation of the assignment of functionality to structure, and its higher abstraction level. These features will be discussed throughout the paper. SLIF has proven to be an efficient format for system design tasks, as verified by several years of use in the SpecSyn system-design tool.

The paper is organized as follows. In Section 2, we provide a SLIF definition and an example. In Section 3, we demonstrate SLIF's usefulness for system design tasks. In Section 4, we discuss related research. In Section 5, we demonstrate SLIF's efficiency. In Section 6, we discuss strengths and limitations of SLIF. In Section 7, we provide conclusions and future work.

2 SLIF definition

After providing a simple example to introduce SLIF's basic organization, we provide a set of annotations for estimation, and a more formal SLIF definition.

2.1 Basic organization

As a simple example, consider the partial VHDL specification of a fuzzy-logic controller in Figure 1. Inputs *in1* and *in2* must be converted to output *out1* using fuzzy logic. The main process *FuzzyMain* first samples input values by writing them into variables *in1val* and *in2val*. It then calls procedure *EvaluateRule* twice, once for each input, and that procedure fills an array (*tmr1* or *tmr2*) based on the input and on another predefined array (*mr1* or *mr2*). After convolving the *tmr* arrays, a centroid value is computed and output. The process repeats after a time interval.

We represent this specification as the directed graph in Figure 2. Each node of the graph represents a **behavior** or a **variable** from the specification, where a behavior is a process or procedure, though for finer granularity we can consider statement blocks like loops. Each directed edge of the graph represents a communication **channel** from the specification, where a channel represents a procedure call, a variable/port read or write, or a message pass specified using send/receive constructs. For example, process *FuzzyMain*, procedure *EvaluateRule* and variable *in1val* are each represented by a node. The write of *in1val* in *FuzzyMain* translates to a single edge, while the two calls of *EvaluateRule* by *FuzzyMain* translate to another single edge. Nodes representing processes are tagged to distinguish them from procedure nodes (hence the *FuzzyMain* node is shown in bold).

We refer to the representation as the **Specification-level intermediate format**, or **SLIF**, since its granularity is that of behaviors and variables explicit in the specification. We refer to the part of SLIF shown so far as an **access graph**, or **AG**, since the relations between the functional objects are defined by the accesses among those objects. The AG is similar to a procedure call-graph commonly used for software profiling, where an edge represents an access rather than a flow of data; the AG is more general since it also includes variables. (Since there may be

```
entity FuzzyControllerE is
   port ( in1, in2 : in integer;  out1:  out integer );
end;
...
FuzzyMain: process
        variable in1val, in2val : integer;
        type mr_array is array (1 to 384) of integer;
        variable mr1, mr2: mr_array; -- membership rules
        type tmr_array is array (1 to 128) of integer;
        variable tmr1, tmr2: tmr_array; -- truncated memb. rules
        ...
begin
        in1val := in1;  in2val := in2;
        EvaluateRule(1);
        EvaluateRule(2);
        Convolve;
        out1 <= ComputeCentroid;
        wait until ...
end process;


procedure EvaluateRule(num : in integer) is
   variable trunc : integer;  -- truncated value
begin
   if (num = 1) then
      trunc := Min(mr1(in1val), mr1(128+in1val));
   elsif (num = 2) then
      trunc := Min(mr2(in2val), mr2(128+in2val));
   end if;

   for i in 1 to 128 loop
      if (num = 1) then
         tmr1(i) := Min(trunc, mr1(256+i));
      elsif (num = 2) then
         tmr2(i) := Min(trunc, mr2(256+i));
      end if;
   end loop;
end;
```

Fig. 1: Partial fuzzy-controller VHDL specification.

a large number of variables, we usually include only array variables and global variables, treating each remaining local scalar variable as part of the behavior in which it was declared).

Note a unique feature of SLIF: *SLIF is oriented around accesses rather than dependencies.* Such an orientation results in fewer nodes and more closely reflects the likely implementation style of each procedure. For example, if a procedure is called many times throughout the specification, the SLIF-AG would use only one node for the procedure; in contrast, a dependency graph would use multiple nodes, one per access, since the dependencies with other behaviors would differ for each access. Multiple nodes are useful in behavioral synthesis when the node represents an arithmetic operation, since the main design task is to map many identical operations to a single functional unit.

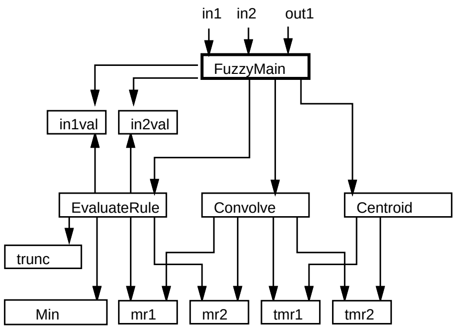

Fig. 2: Basic SLIF-AG for the example.

However, multiple nodes are often unnecessary in system design, since each node represents a more complex computation that we may implement as a single custom controller/datapath, a single custom control subroutine, a single functional unit, or a single software subroutine; there is no complicated binding stage of multiple identical nodes to a single unit. In case we do decide to replicate the procedure, it will likely be through inlining, which would still not require multiple nodes. In the cases where we actually want multiple nodes, then we can provide an automatic transformation to replicate the node for each access.

Continuing with the SLIF definition: SLIF represents structural system components and the assignment of functional objects to those components. Returning to the fuzzy-logic example, suppose we decide to implement *EvaluateRule* on an ASIC, *Convolve* on a standard processor, and *mr1* and *mr2* in their own memory. We represent this information by adding structural components and assigning graph nodes to those components. A structural component may be a custom or standard **processor**, which implements behaviors and variables; a **memory**, which implements variables; or a **bus**, which implements channels. For example, Figure 3 illustrates the assignment of several functional objects to an ASIC, a processor, two memories, and one bus.

Note a second unique feature of SLIF: *SLIF represents not only functionality, but also implementation structure, and the assignment of functionality to structure.* Therefore, we can represent system-design results in the same format, thus supporting iterative specification refinement, and we can also include structure in the initial input specification, thus handling partial implementation decisions already made by the designer.

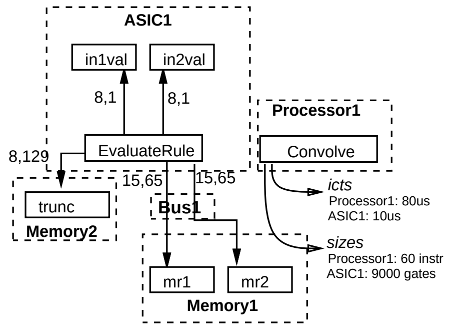

Fig. 3: SLIF after allocation, partitioning, annotation.

2.2 Annotations for estimations

We now consider annotating SLIF with preprocessed information for performance, I/O and size estimation.

We first consider performance. We annotate each edge with an access frequency, **accfreq**, indicating the number of accesses by the edge's source behavior to the edge (channel) during one start-to-finish execution of the behavior. Since execution may be data-dependent, we use an average execution, based on a (manually or automatically generated) branch probability file. Likewise, we can associate maximum and minimum frequencies; for brevity, though, we consider only averages in this paper.

We also annotate each edge with a **bits** number, indicating the number of bits transferred during an access. For access to a scalar, this is the number of bits into which the scalar would be encoded. For access to an array of scalar elements, this is the number of bits to encode an array element, plus the number of address bits needed to specify an element's address. For a more complex data item, such as a three-dimensional array, the data item is first transformed to an array of scalars. For access to another behavior, this is the number of bits needed to transfer all (if any) parameters, where the number of bits for each parameter is computed as for variables. For a message pass, this is the number of bits into which the message would be encoded. Note that an edge's bits number is independent of physical bus width; if the bus width is smaller than the bits, then multiple transfers will be required.

We annotate each behavior node with its internal computation time, or **ict**, indicating the behavior's execution time excluding communication time. Since a behavior executes at different speeds on different components, we annotate each behavior node with a set of ict's, one ict for each component type. A behavior's ict is determined during preprocessing, using synthesis for ASIC components, compilation for processor components, or designer-provided numbers. We also annotate each variable node with an ict set, representing the times to read or write various memories/processors in which the variable may be implemented.

We annotate each bus with a **ts** (time-same) number, which is the time to transfer data entirely within the same system component, and a **td** (time-different) number, which is the time to transfer data between two different system components. The td is usually much larger than the ts. (We could even use more extensive annotations with a unique ts value for each component type, and a unique td value for each pair of component types).

We now turn to I/O (input/output), which is the number of pins required on a system component for a given partition. To determine I/O, we need bus width information. Thus, we annotate each bus with a **width** number, which represents the number of physical bus wires. We can also annotate each system component with a constraint, **iocon**, on the maximum I/O that the component can implement. For an ASIC, this number might be the number of pins excluding power and ground, whereas for a standard processor, it might be the size of the bus.

Finally, we consider size information. For behaviors and variables assigned to a processor, size is measured by the number of program and data bytes after compilation, whereas on a custom processor, size is the number of gates, cells, transistors, or combinational logic blocks after synthesis. For variables assigned to a memory, size is the number of words. Because size means something different for each component type, we annotate each behavior and variable node with a **size** set, one size for each component type, obtained by synthesizing or compiling each behavior into the appropriate technology. We can also annotate each component with a size constraint, **sizecon**.

Several annotation examples are shown in Figure 3. The edge from *EvaluateRule* to *in1val* is annotated with a bits value of 8 (assuming an 8-bit integer encoding) and an access frequency of 1, while the edge to array *mr1* has 15 bits (7 address bits plus 8 data bits) and an access frequency of 65. The ict set for *Convolve* shows 80 us on the processor and 10 us on the ASIC, while the size set shows 60 processor instructions and 9000 ASIC gates.

Note a third unique feature of SLIF: *SLIF represents a specification at a high abstraction level of procedural-level objects*, rather than the detailed level of arithmetic-level operations. This means that there are far fewer functional objects (tens or hundreds rather than thousands), resulting not only in faster run-times and less memory from automated algorithms, but more importantly, in a representation that is comprehensible to designers. Hence, SLIF supports interactive design, fitting well with current system-design methodologies, which deal with procedural-level functions. In addition, it means that we can perform extensive estimation preprocessing for each procedural-level object, and then quickly combine values without too much loss of accuracy from unconsidered inter-procedural optimizations. In contrast, the arithmetic-level does not permit such preprocessing; e.g., we cannot estimate a behavior's time or size as the sum of its operations' times and sizes, since those operations will be combined to share hardware in a complex and unpredictable manner.

2.3 Formal definition

Figure 4 provides a formal definition of SLIF with annotations. Note that F_objs is the same as the SLIF-AG.

Item	Definition
SLIF	$< F\_objs, S\_objs >$
F_objs	$< IO_{all}, BV_{all}, C_{all} >$ (Functional objects)
IO_{all}	$\{io_1, io_2, ...\}$ (Input/output ports)
BV_{all}	$B_{all} \cup V_{all}$
B_{all}	$\{b_1, b_2, ...\}$ (**Behaviors**)
V_{all}	$\{v_1, v_2, ...\}$ (**Variables**)
b_i, v_i	$< icts, sizes, process >$
$icts$	$\{ict_1, ict_2, ...\}, ict_k =< cmp, val >$
$sizes$	$\{size_1, size_2, ...\}, size_k =< cmp, val >$
cmp	$\in P_{all} \cup M_{all}$
C_{all}	$\{c_1, c_2, ...\}$ (**Channels**)
c_i	$< src, dst, accfreq, bits >$
src	$\in B_{all}$ (Accessor behavior)
dst	$\in BV_{all} \cup IO_{all}$ (Accessee object)
S_objs	$< P_{all}, M_{all}, I_{all} >$ (Structural objects)
I_{all}	$\{i_1, i_2, ...\}$ (**Buses**)
i_k	$< C, width, ts, td >, C \subset C_{all}$
M_{all}	$\{m_1, m_2, ...\}$ (**Memories**)
m_k	$< V, sizecon >, V \subset V_{all}$
P_{all}	$\{p_1, p_2, ...\}$ (**Processors**, custom or standard)
p_k	$< BV, sizecon, iocon >, BV \subset BV_{all}$

Fig. 4: Formal SLIF definition

3 Using SLIF for system design

In this section, we provide examples of how SLIF efficiently supports the system design tasks of allocation, partitioning, transformation, and estimation.

3.1 Allocation, partitioning, transformation

Allocation is easily supported since system components are represented in the format. Figure 3 illustrated the SLIF representation after an ASIC, a processor, two memories, and a bus were allocated. To add a system component, we simply add a member to I_{all}, M_{all}, or P_{all}.

Partitioning of functional objects among system components is also directly represented, as shown in Figure 3. To represent assignment of functional objects among components, we simply modify $i_k.C, m_k.V$, or $p_k.BV$.

Many transformations are also easily supported. For example, inlining a procedure into a behavior would be supported by eliminating the behavior's access edge to that procedure, and recomputing the behavior's annotations after inlining the procedure. As another example, process merging would be supported by merging two process nodes into one, and once again updating the annotations. Converting a procedure to a process would be achieved by tagging the procedure node as a process, and replacing edges to that node by edges to new global variable nodes. In general, system-level transformations require a modification of the graph and a recomputation of affected annotations.

3.2 Estimation

From the earlier annotations, we can straightforwardly obtain coarse estimates for execution time, bitrate, software size, hardware size, memory size, and I/O.

We can estimate a behavior's execution time as the sum of the behavior's internal computation time (ict) and communication time. We use a procedure $GetBvComp(bv)$ that returns the component pm to which bv is assigned. Procedure $GetBvIct(bv, pm)$ finds the ict_k in the $bv.icts$ for which $ict_k.cmp$ equals the given component pm, and returns $ict_k.val$. Procedure $GetBehChans(b)$ returns all channels C accessed by behavior b (i.e., $c.src = b$). Procedure $GetChanBus(c)$ returns the bus i to which channel c is mapped. Execution time is then computed as follows:

$$Exectime(b) = GetBvIct(b, p) + Commtime(b) \qquad (1)$$

$$\begin{array}{ll}
p & = GetBvComp(b) \\
Commtime(b) & = \sum_{c_k \in GetBehChans(b)} c_k.accfreq \times \\
 & \quad (TTime(c_k, p) + Exectime(c_k.dst)) \\
TTime(c_k, p) & = \lceil bdt_time \times (c_k.bits \div \\
 & \quad GetChanBus(c_k).width) \rceil \\
bdt_time & = GetChanBus(c_k).ts \\
 & \quad \text{if } GetBvComp(c_k.dst) = p, \\
 & = GetChanBus(c_k).td \text{ otherwise.}
\end{array}$$

In other words, a behavior's execution time equals its ict on the current component ($GetBvIct(b,p)$), plus its communication time ($Commtime(b)$). The communication time equals the data-transfer time over a channel for each accessed object ($TTime(c_k, p)$), plus the execution time of each accessed object ($Exectime(c_k.dst)$), times the number of such accesses ($c_k.accfreq$). The data-transfer time over a channel is determined from the bus data transfer time (bdt_time) and the width of the channel's bus; if the data bits exceeds the bus width, then multiple transfers are used (as computed by the division). The bdt_time is less when the communication is within one component.

We now compute bitrate, or the rate that bits are transferred over a channel or bus. A channel's bitrate can be computed as the number of bits transferred during the execution time of the source behavior. Using $Exectime(b)$ defined above, we compute channel bitrate as follows:

$$ChanBitrate(c) = \frac{c.accfreq \times c.bits}{Exectime(c.src)} \qquad (2)$$

A bus bitrate is the sum of that bus' channel bitrates. For more sophisticated bitrate estimation techniques derived from SLIF, see [18].

We now focus on size estimation. Estimating software size for a standard processor component, hardware size for a custom component, and memory size for a memory component each requires adding the precomputed weights of each functional object for the given component.

Finally, we consider I/O estimation. I/O is the number of wires crossing the boundary of a system component. Usually relevant for ASICs, I/O can be computed as:

$$I\_O(p) = \sum_{i_k \in CutBuses(p)} i_k.width \qquad (3)$$

$$\begin{array}{ll}
CutBuses(p) & \subset I, i_k \in CutBuses(p) \text{ iff} \\
 & i_k \bigcap CutChans(p) \neq \emptyset, \\
CutChans(p) & \subset C, c_l \in CutChans(p) \text{ iff} \\
 & c_l.src \in p \text{ and } c_l.dst \notin p \text{ or} \\
 & c_l.dst \in p \text{ and } c_l.src \notin p
\end{array}$$

In other words, the wires crossing a component boundary equals the total width of the buses crossing that boundary ($CutBuses(p)$), which in turn are those buses that implement at least one channel crossing the boundary.

4 Related work

Many projects focus on partitioning functionality among hardware or hardware/software components. The need for coarse-grained procedural-level partitioning was stressed in [2], as well as in [3, 4]. Approaches in [5, 6, 7] partition at the finer-grained statement level (or statement sequence). Other approaches, such as those in [8, 9, 10, 11, 12, 13], partition at the fine-grained arithmetic-operation level. Additional research has focused on functional transformation, at a procedural level of granularity in [14, 15], at a loop and statement level in [16], and at an operation-level in [17]. (Details on several of these efforts can be found in [18]). SLIF is best-suited for the procedural-level partitioning and transformation approaches described above.

5 Results

SLIF serves as the internal format for the SpecSyn system design tool. SpecSyn takes VHDL as input, creates an internal SLIF representation, and then permits rapid exploration of various allocations (including processors, ASICs, memories and buses), partitions and transformations, providing rapid estimates of size, I/O, and performance metrics. The SLIF creation software consists of 12,200 lines of code, and it is integrated with hardware and software size and performance estimators; it provides each estimator with a procedure and a technology type, and the estimator returns the metric value. The estimators are distinct from the SLIF creation program, so we can replace any estimator by another estimator, or even by a synthesis or compilation tool, to gain more accuracy at the expense of longer estimation times.

We measured the SLIF creation time and estimation time for four examples: an answering machine, an ethernet coprocessor, a fuzzy-logic controller, and a volume-measuring medical instrument. To indicate the examples' complexities, we note that the examples had an average of 554 lines, 58 behavior and variable functional objects, and 68 channels. The SLIF creation time averaged only 3.35 seconds. In addition, once the SLIF was created, estimation times were on the order of milliseconds.

To demonstrate SLIF's efficiency compared with lower-granularity formats, we compared SLIF with two other formats for the fuzzy-logic controller. The SLIF-AG required 35 nodes and 56 edges. The ADD format [19], which is similar to the Value Trace format, required over 450 nodes and 400 edges. The CDFG format required over 1100 nodes and 900 edges. The difference in complexity greatly affects the design algorithms (e.g., partitioning) that can be applied. For example, if an n^2 algorithm is to be applied, then the SLIF-AG, VT or ADD, and CDFG formats would require 1225, 202500, and 1210000 computations, respectively. Clearly, the latter two are not practical for an interactive tool.

6 Strengths and limitations

We have discussed several strengths of SLIF, including its orientation around accesses, its representation of both functionality and structure, and its high level of abstraction. There are also some limitations. First, SLIF does not represent fine-grained operations, so cannot be used for fine-grained scheduling. However, we can associate a fine-grained representation with each SLIF node, such as a control/dataflow graph, to address this problem. In fact, we may associate a hierarchical transition graph [15] with each node to support process merging transformations. Second, SLIF is not simulatable. However, with the popularity of the VHDL standard, we see no need for developing a simulator based on SLIF. Third, hardware estimation for a set of procedures can not always be estimated accurately by summing the procedure sizes. For this reason, we have developed sophisticated annotations that consider hardware sharing among procedures [20]. Fourth, SLIF associates only one average (and minimum/maximum) execution time with each procedure; however, execution time often depends on the calling location, so one might want to extend SLIF to incorporate such call-dependent information. Fifth, SLIF doesn't describe forked procedures since they are not supported by VHDL. However, we can extend SLIF to represent such parallel calls by associating tags with each channel; channels with the same tag would be considered concurrent. Finally, we may wish to extend SLIF to handle multiport memories, explicit component pin definitions (e.g., interrupt pins on a standard processor), and hierarchical system components.

7 Conclusions and future work

We have presented the SLIF specification-level internal format, shown that its unique features make it well-suited for the system design tasks of allocation, partitioning, transformation and estimation, and demonstrated that its abstraction-level is appropriate for designer interaction. SLIF has proven to be an efficient internal format in the SpecSyn system design tool, which has been released to over 20 companies and universities and has been used experimentally in several industry designs.

We hope to continue to improve estimation accuracy by linking with more accurate estimators during preprocessing and by tuning our estimation equations based on SLIF for various technologies; there are currently several efforts with industry partners in this direction. We also plan to develop an environment for functional VHDL transformation, using SLIF as the core representation. Finally, if there is sufficient interest, we may create a textual version of SLIF for communication among system design tools, as well as a standalone form of our current VHDL-to-SLIF software for use by other system-design tool developers.

References

[1] S. Narayan and D. Gajski, "Synthesis of system-level bus interfaces," in *EDAC*, 1994.

[2] F. Vahid and D.Gajski, "Specification partitioning for system design," in *DAC*, pp. 219–224, 1992.

[3] D. Thomas, J. Adams, and H. Schmit, "A model and methodology for hardware/software codesign," in *IEEE Design & Test*, pp. 6–15, 1993.

[4] P. Gupta, C. Chen, J. DeSouza-Batista, and A. Parker, "Experience with image compression chip design using unified system construction tools," in *DAC*, pp. 250–256, 1994.

[5] R. Gupta and G. DeMicheli, "Hardware-software cosynthesis for digital systems," in *IEEE Design & Test*, pp. 29–41, October 1993.

[6] R. Ernst, J. Henkel, and T. Benner, "Hardware-software cosynthesis for microcontrollers," in *IEEE Design & Test*, pp. 64–75, December 1994.

[7] X. Xiong, E. Barros, and W. Rosentiel, "A method for partitioning UNITY language in hardware and software," in *EuroDAC*, 1994.

[8] R. Gupta and G. DeMicheli, "Partitioning of functional models of synchronous digital systems," in *ICCAD*, pp. 216–219, 1990.

[9] E. Lagnese and D. Thomas, "Architectural partitioning for system level synthesis of integrated circuits," *IEEE Trans. on CAD*, July 1991.

[10] K. Kucukcakar and A. Parker, "CHOP: A constraint-driven system-level partitioner," in *DAC*, 1991.

[11] Z. Peng and K. Kuchcinski, "An algorithm for partitioning of application specific systems," in *EDAC*, pp. 316–321, 1993.

[12] Y. Chen, Y. Hsu, and C. King, "MULTIPAR: Behavioral partition for synthesizing multiprocessor architectures," in *IEEE Trans. on VLSI*, pp. 21–32, 1994.

[13] C. Gebotys, "An optimization approach to the synthesis of multichip architectures," *IEEE Trans. on VLSI*, vol. 2, no. 1, pp. 11–20, 1994.

[14] T. Ismail, K. O'Brien, and A. Jerraya, "Interactive system-level partitioning with Partif," in *EDAC*, 1994.

[15] J. Hagerman and D. Thomas, "Process transformation for system level synthesis." TR CMUCAD-93-08, 1993.

[16] H. Samsom, L. Claesen, and H. DeMan, "Synguide: an environment for doing interactive correctness preserving transformations." In *VLSI Signal Processing VI*, IEEE Press, New York, 1993.

[17] R. Walker and D. Thomas, "Behavioral transformation for algorithmic level IC design," *IEEE Trans. on CAD*, October 1989.

[18] D. Gajski, F. Vahid, S. Narayan, and J. Gong, *Specification and design of embedded systems*. New Jersey: Prentice Hall, 1994.

[19] V. Chaiyakul and D. Gajski, "High-level transformations for minimizing syntactic variances," in *DAC*, 1993.

[20] F. Vahid, S. Narayan, and D. Gajski, "Constant-time cost evaluation for behavioral partitioning." UCI TR 92-29,1992.