Procedure Exlining:
A Transformation for Improved System and Behavioral Synthesis

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

Abstract

We present techniques for solving the inverse problem of procedure inlining, namely the problem of replacing sequences of statements with procedure calls. Two techniques are provided, one for finding redundant sequences of statements that can be replaced by calls to one procedure, and another for dividing a large set of statements into several procedures, where each procedure performs a distinct computation. Such procedure exlining can transform a behavioral specification, originally written for readability, into a specification that can be implemented efficiently, because procedures can greatly improve the results of synthesis tools. We demonstrate the usefulness of the techniques on several examples.

1 Introduction

A functional specification serves the purpose of precisely defining a system's intended behavior. Such a specification usually will be read by humans as well as input to synthesis tools. Unfortunately, a specification written for readability may not directly lead to the best synthesized design. Designers often try to juggle such synthesis considerations with readability considerations while writing the initial functional specification. Such juggling usually leads to lower readability, less portability, and more functional errors. Alternatively, we have observed specification writers rewriting their specifications to explore different implementation options, e.g., creating one specification that is sequential, another one concurrent, another one pipelined, etc. Such rewriting is tedious and time-consuming, and hence prone to human error. We can reduce the juggling, or the rewriting effort, by providing an interactive transformation tool. With such a tool, the designer may focus on readability of the initial specification, confident that the tool will enable easy conversion of the specification for synthesis. Well-known transformations that such a tool might support include process splitting, process merging, procedure inlining, and loop unrolling.

Procedure exlining is a new transformation that can be particularly useful in such a tool. We define procedure exlining as the inverse of procedure inlining (hence its name), namely, finding and replacing sequences of statements by procedure calls. For example, Figure 1(a) illustrates an initial specification consisting of several statements, one set of which is already part of a procedure. Figure 1(b) demonstrates the exlining of several sequences of statements into procedures. There are many benefits of introducing more procedures into a specification when we plan to subsequently use system or behavioral synthesis tools. First, a procedure can represent a basic indivisible computation, thus defining the granularity of functional partitioning [1, 2, 3, 4, 5]. Second, a procedure can be converted to a process to represent a separate controller, in order to simpify the synthesized control logic [6, 7], or to achieve more concurrent execution [8]. Third, procedures can be input separately to synthesis tools, in order to reduce the memory and runtime requirements of a synthesis tool (which are typically polynomial in the size of the specification), as successfully done at the logic level in [7]. Fourth, procedures provide a variety of implementation options to a behavioral synthesis tool [9], including a complex functional unit [10, 11], a control subroutine, a concurrent controller, or an inlined behavior. Figure 1(c) shows how the original specification can be decomposed into a set of functional objects, where each object represents a procedure. Those objects can then be partitioned among parts, as illustrated in Figure 1(d), where each part represents a system component, a processor, or an input to a synthesis tool.

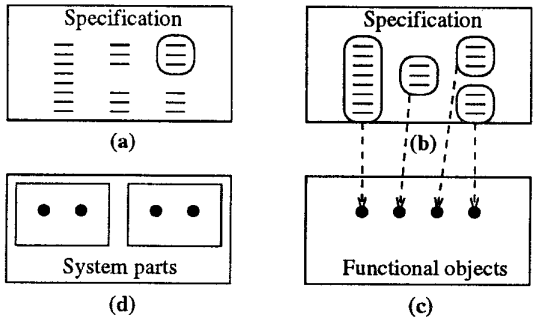

Fig. 1: Procedure exlining during system partitioning: (a) Initial specification, (b) after exlining procedures, (c) after decomposition based on procedures, (d) after functional partitioning.

Procedure exlining transformations create procedures for one of two purposes:

- **Redundancy elimination**: we find redundant sequences of statements, and replace those sequences by calls to a single procedure.
- **Distinct-computation isolation**: we divide a large sequence of statements into several subsequences, where each subsequence performs a distinct computation, and we replace each subsequence by a call to a distinct procedure.

Each problem requires a different solution technique.

This paper is organized as follows. Section 2 describes our technique for redundancy elimination, and Section 3 highlights results of two examples. Section 4 describes our technique for distinct-computation isolation, and Section 5 highlights several experiments. Section 6 provides conclusions.

2 Redundancy elimination

We often wish to replace redundant sequences of statements by a call to a single procedure. Such a procedure may have been initially omitted if it wasn't necessary for readability or if the redundancy simply wasn't noticed; adding the procedure will likely improve synthesis results. Unfortunately, humans have a hard time finding redundant sequences in thousands of lines of code. Current tools, such as text searching tools, graph matching tools, state-machine factorization tools, and program equivalence tools, are not ideally suited for finding such sequences. For example, suppose we wish to find all if-then-else statements that compute the maximum of two integers, such as statements 2-6 of Figure 3(a). Text-searching tools aren't by themselves sufficient because we can't pick a pattern to search for: statement keywords (such as *if*) are too general, while symbol names (such as *x*) are too specific. Graph-matching tools, which first convert the specification to a graph form and then search for a particular subgraph in the graph [12, 13], might prove useful except for the fact that they would require decomposing the specification to a granularity finer than statements, making designer interaction nearly impossible. State-machine factorization tools [14] also eliminate designer interaction due to their granularity, and they consider only the control, not the data computations. Finally, program equivalence tools don't help because we don't have two programs that we wish to compare; instead, we have one "program" (the statement sequence), and a large specification in which we are searching for equivalent subprograms (and to make matters worse, we don't even want exact equivalence, but instead must allow for variation).

The technique that we have developed to solve this problem operates at the statement level, since this level enables designer interaction. We decided that the problem of finding sequences having identical computations was too constrained, because we wanted to allow for some variation in the way the statements were written (as is often the case), and because that problem seemed intractable. Instead, we focused on the problem of finding sequences with similar *form*, which makes the problem more tractable. Such a problem requires designer input to decide when such sequences are actually redundant, but this requirement is acceptable since we assume interactivity.

In our approach, the designer provides a statement **pattern**, which is a sequence of statements, and we then search for **candidate** sequences throughout the specification. Alternatively, the pattern can also be obtained automatically by tabulating the number of occurences of all patterns of a certain length, and picking the pattern with the most occurrences. There are three criteria that can be used when searching for candidates, as will now be discussed.

Statement type	Encoding
assert	a
case	c
else	e
elsif	g
end case	d
end if	j
end loop	m
exit	b
for loop	f
if	i
loop	l
null	n
procedure call	p
return	r
signal assignment	s
variable assignment	v
wait	w
when	x
while loop	y

Fig. 2: Statement-type encoding characters for VHDL

2.1 Initial criteria: statement types

Here, we simply look for a candidate sequence whose types are similar to the pattern's types. We encode each statement of the pattern into a single character representing that statement's type, and concatenate those characters into a string called the **encoded pattern**. The characters used to encode VHDL sequential statements are shown in Figure 2. We encode the specification's statements similarly; the resulting characters are concatenated into one string called the **encoded string**. We then use the *agrep* pattern matching tool [15] to search for matches of the encoded pattern in the encoded string. We then display all candidate matches to the designer.

For example, consider the example specification of Figure 3(a), which is trivially small and is used only to demonstrate the technique. We wish to search this specification for a sequence of statements that computes the maximum of two numbers; the pattern appears in lines 2-6. The encoded pattern is therefore the following:

$$ivevj$$

while the encoded string is:

$$vivevjivevjwfivevjm$$

Applying a pattern-matching tool yields matches at the second, seventh, and thirteenth characters of the encoded string, meaning there are matches at those line numbers, as shown in column A of Figure 3(c).

2.2 Refined criteria: target/source consistency

Matching based solely on statement types will result in many candidates that perform very different computations from the pattern. We can reduce the number of candidates by requiring target consistency. A **target** is the variable identifier being updated in a variable assignment statement (likewise for VHDL signals). Target consistency means that if the pattern writes a variable v with its i'th and j'th statements, then a candidate must also write some variable u (possibly different than v) with its i'th and j'th statements. For example, in Figure 3, we note that although all three candidates possess two variable assignments, only the first two candidates use the same target

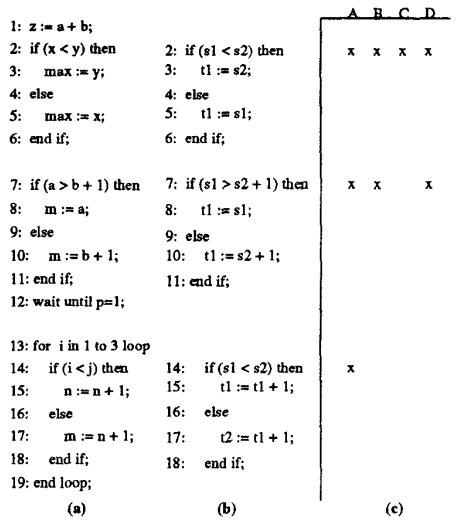

Fig. 3: Exline example: (a) specification, (b) target and source symbolic replacements, (c) matches with various criteria.

for both assignments, as does the pattern; the third candidate, however, uses two different targets. Likewise, we can require source consistency. A **source** of a statement is any identifier that is read by that statement.

To extend our encoding with target consistency, we first replace all targets by symbolic identifiers. We then create an encoded pattern from the pattern's statements. Each statement is encoded as $cTtx$, where c is the statement type, and tx is the target's symbolic identifier. For source consistency, symbols have the form sx instead of tx, and the encoded pattern is $cTtxSls$, where ls is a list of symbols of the form $sx\_sy\_...sz$. If a statement does not have a target or any sources, then the corresponding target or source part is omitted from the encoding. As before, we concatenate the code for each statement into an encoded pattern.

After encoding the pattern, we encode each candidate. Finally, we search for "occurences" of the encoded pattern string in each encoded candidate string using agrep. If the encoded pattern string matches the encoded candidate string, then one occurence will be found, otherwise no occurence will be found and the candidate is discarded.

For example, Figure 3(b) shows the initial candidates of Figure 3(a), where targets and sources have been replaced by symbols. For target consistency, the pattern (lines 2-6) would be encoded as: *i vTt1 e vTt1 j*. Likewise, the candidates corresponding to lines 7-11 and lines 14-18 would be encoded as *i vTt1 e vTt1 j* and *i vTt1 e vTt2 j*, respectively. Thus, note that lines 14-18 no longer match the pattern, as indicated in column B of the figure, because of the appearance of $t2$ rather than $t1$ at the end of the encoded candidate string. For source consistency, lines 7-11 would no longer match either, as indicated by column C of the figure, because the symbols $s1$ and $s2$ are used in different statements than in the pattern.

Note that we can not initially search the specification for matches based on target (or source) consistency. The reason is that consistency is based on a target's order of appearance in the pattern and in the candidate. If we didn't already select candidates based on statement types, then we wouldn't be able to determine the x values of the targets in the specification's statements.

2.3 Approximate and regular expr. matching

A powerful advantage of using string matching is that we can perform efficient approximate-matching. For example, statements may appear in a slightly different order, a statement may have additional sources (which usually means an expression would be passed to the procedure), or a sequence may have a few extra statements (again requiring a parameter to indicate when those statements should be executed). Several fast approximate matching techniques have evolved recently. The approximate matching problem is to find all substrings that are no more than a certain distance from the pattern. A string S is of distance d from a string T if we can transform S to be the same as T with a sequence of d insertions of single characters in T, deletions of single characters from T, or substitutions of characters. A fast (sub-linear) approach to approximate text searching is described in [15]. We use their text searching tool, *agrep*, rather than using the standard Unix *grep*, because of the tool's ability to find approximate matches. Therefore, for any of the matching criteria listed above, we can request a distance d to obtain approximate matches. For example, if we allow a distance of two when performing source consistency matching in Figure 3, then lines 7-11 will match the pattern, even though there are two differences from the pattern.

A second very powerful advantage of using string matching is the ability to use regular expressions. String matching tools typically allow a pattern to be a regular expression. Such expressions enable one to pose advanced matching criteria, such as in the following example: find all sequences of statements of a wait statement followed by a for loop followed by either a wait statement or a signal assignment, where the loop body contains any number of variable or signal assignments. Also, because most Unix users are familiar with regular expressions, little additional learning time is needed to take advantage of this powerful feature.

3 Redundancy elimination examples

We have implemented the procedure exlining technique as part of a VHDL transformation tool. The tool reads VHDL, after which the designer can interactively apply several transformations, including procedure inlining, procedure exlining, and conversion of a procedure to a process. The tool consists of 26,000 lines of C code, which includes the code for VHDL parsing and internal representation.

To demonstrate the usefulness of our procedure exlining technique, we obtained a 780-line VHDL image-processing example written by an outside source for simulation purposes. We began looking for potential statement patterns, and found the following twice:

```
for i in 0 to (c_Pheight - 1) loop
   c_temp := c_temp or Pimage(c_PC);
   c_PC := c_PC + c_Width;
end loop;
```

The exlining tool created the encoded pattern "f v v m" for those statements. The tool then found 14 initial candidates. Interactively, we refined the matching criteria to target consistency; all candidates still matched. We refined the criteria to target and source consistency, looking for candidates encoded as "fSs1 vTt1Ss2_s3_s4 vTt2Ss2_u5 m", which eliminated all but three candidates. We then determined that those three represented the same computation; so we replaced all three occurences by a single procedure call.

We used one of the 11 remaining candidates as a pattern. Following the same steps as above led to replacing 9 of those candidates by a procedure call. The last two were then replaced by yet another procedure call.

Further examination of the specification led to finding another potential pattern, which yielded 4 target and source consistent candidates. However, a particular feature of the exline tool enabled us to detect something we might not have otherwise seen: the exline tool optionally prints a few statements that precede and follow each candidate, which led us to note that our initial pattern was too small, as all of the candidates were enclosed in a while loop. After iterating with a new, larger pattern, we replaced all four candidates by a procedure, with three parameters to account for variations. Overall, we introduced 4 new procedures, replaced statement sequences by 18 procedure calls, and reduced the overall code size by 57 lines. The entire process of scanning the specification, selecting patterns, and searching for candidates took only 10 minutes with the assistance of the exlining tool.

We also applied the exline tool to a high-level synthesis benchmark example: the i8251 serial IO device. Taking a similar approach to that of the earlier example, we introduced five procedures and eleven procedure calls (which in turn reduced the code size by 40 statements). Also significant was an error that was detected in the example with the aid of the exlining tool. In particular, the pattern "i s j s" appeared to be a promising exline candidate. The exline tool found two initial matches, and upon brief examination of the surrounding statements, we extended our pattern to "i s j s w s w s s". However, this new pattern resulted in just one match. Looking more closely, we noticed that the other sequence of statements was *missing* a wait statement. Specifically, one sequence of statements was as follows:

```
if rrdy = '1' then
    oeset <= '1';
end if;
rdata <= ldata;
wait for 1fs;
rrdyset <= '1';
wait for 1fs;
rrdyset <= '0';
oeset <= '0';
```

The other sequence was identical, except it was missing the first wait statement. Clearly, one or the other sequence was incorrect. (Since we are not the authors of the example, we do not know which is the correct sequence). We decided to add the wait statement, and we replaced each sequence by a procedure call. The exlining tool was able to help us find the error because it provides a unique "slicing" of the specification, bringing distant parts of the specification together for viewing based on potential redundancy.

4 Distinct-computation isolation

Some procedures are used primarily to break a large computation into smaller computations. Even though each procedure is only called once, those procedures still prove very useful for synthesis as they can serve as the granularity for functional partitioning, can be converted to concurrent processes to improve parallelism, can be mapped to a custom functional unit, or can reduce the runtime or memory requirements of the behavioral synthesis task. Such procedures are often omitted from the original specification for several reasons. First, we may be able to describe a computation with one page of code, using comments to describe the distinct sub-computations. Such code is often easier to write and more readable than code broken into procedures; the latter not only yields more lines of code resulting from the overhead of declaring each procedure and its parameters, but also requires a reader to jump around the specification file or flip between pages to comprehend the entire computation. Second, there is often no good identifier that describes a particular complex computation, making the procedure more of a hindrance than help. Third, procedures in the original specification may not correspond to good procedures for synthesis; in this case, we may inline those procedures, and then exline new procedures.

Once again, we chose to solve this problem at the statement level to maintain designer interaction and to provide readable output; techniques at the arithmetic-operation level can be found in [16]. We first convert the statements to a representation that indicates the valid groupings of statements. Such a representation is necessary so that we don't group half the statements of a loop with statements preceding the loop, for example. We use a straightforward tree representation, where each node represents a statement, and non-leaf nodes represent hierarchical statements such as loops, if-then-else, cases, and procedure calls; the root node represents the process itself. For example, Figure 4(a) shows the tree representation of the example of Figure 3(a). We extend the representation by adding special edges, called **sibling edges**, between sibling nodes in the tree if and only if the corresponding statements always occur in the same execution thread through the process; several sibling edges are shown in the figure. In general, the branches of an if-then-else or a case statement will never have sibling edges between them. The significance of the sibling edge is as follows: only nodes with a sibling edge between them can be merged into a procedure. Only by merging at the appropriate level of hierarchy can we include entire if-then-else statements or case statements into a procedure.

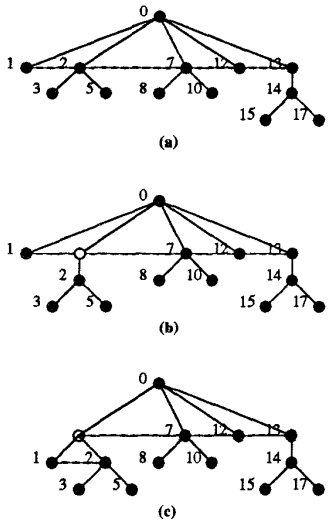

Fig. 4: Tree representation: (a) initial tree, (b) after procedure insertion, (c) after change.

Figure 4(b) shows how the tree is updated when a procedure is introduced. Our problem is to insert such procedure nodes in a manner that results in the best decomposition of the initial statements. To determine the best decomposition, we use a cost function that is a weighted sum of the following terms:

- *Procedure size* – This number is the variation (smaller or larger) from a designer-specified number of statements per procedure, summed over all procedures. The designer specifies the desired number of statements per procedure depending on his intended use of the procedures.
- *Control transfer* – This is the number of transfers of control to procedures over the entire tree. Control transfer is computed from an execution frequency associated with each node (obtained through profiling). An introduced procedure node inherits the frequency of its children, and this frequency corresponds to the number of control transfers to this procedure.
- *Data interconnect* – This is the size of the data that must be transferred to or from each procedure. Size is measured by encoding each data item into bits. Bits for arrays are the address plus the data word bits.
- *Data transfer* – This is the amount of data that must be transferred to each procedure. It is equal to the data interconnect size multiplied with the control transfer frequency.
- *Hardware size* – This is the total synthesized hardware size assuming each procedure is synthesized independently. This term encourages groupings in which large hardware items (e.g., multipliers) appear in the same procedure so can be shared.

Note that the control transfer, data interconnect, and hardware size terms are similar to those terms proposed in [16], but defined on a higher-level of abstraction. The designer may weigh each term as heavily as desired, depending on his design goals.

We have defined three solution techniques for the above problem, spanning the spectrum of fast heuristics to computationally intensive heuristics. The first technique, which we call the *naive* hueristic, simply inserts a procedure node for every hierarchical statement. The complexity of this heuristic is $O(n)$, where n is the number of nodes.

The second technique is a *clustering* heuristic, where we compute a closeness for all pairs of nodes connected by a sibling edge, merge the closest into a new procedure, and repeat. The closest nodes are those whose resulting procedure's control transfer value would be the smallest over all pairs. Alternatively, closeness can be defined as a weighted sum of the control transfers, data interconnect, data transfers, and hardware size terms. A merge of two nodes has the following effect: (1) If the two nodes were not procedure nodes, then a new procedure node is created as the parent of the two nodes, as in Figure 4(b); (2) If one node is a procedure node and the other is not, then the subtree rooted at the other node is made a child of the procedure node, as shown in Figure 4(c); (3) If both nodes are procedure nodes, then one node's children are made children of the other node, and the first node is deleted. We prohibit merges that would exceed the maximum procedure size, which in turn provides a condition for terminating the clustering. The complexity of the clustering heuristic is $O(e^2)$, where e is the number of sibling edges, which we note is less than n.

The third technique uses *simulated annealing*. Given an initial tree with procedure nodes, perhaps created with one of the above heuristics, we attempt to improve the cost function value through a series of changes. The set of possible changes is determined by looking at each sibling edge; for each edge, we can merge the edge's nodes (the merge operation was described above), or, if the nodes' parent is a procedure node, we can move one of the nodes outside of the procedure. An example merge is shown in Figure 4(c); an unmerge is obtained by returning to Figure 4(b). The complexity of simulated annealing is difficult to determine, but usually requires long runtimes.

5 Distinct-computation experiments

We applied the above three algorithms on several examples taken from an image processing VHDL file and an MPEG decoder VHDL file, both of which came from outside sources. Each example was a 100+ line procedure taken from one of the above two files. Table 1 summarizes the results. The resulting cost function value is given before and after applying each heuristic.

The naive algorithm displayed runtimes on the order of one second, clustering averaged twenty seconds, and simulated annealing averaged eight minutes. The simulated annealing heuristic resulted in an average of 16 procedures with an average size of 14 statements each (based on the designer-specified procedure size of 15 statements).

Example	Original	Naive	Cluster	Sim. Ann.
Ex1	2019	1169	1144	828
Ex2	2543	1999	1068	721
Ex3	2040	695	455	434
Ex4	2002	1096	330	221
Ex5	1965	1482	1411	1192

Table 1: Results of three exlining heuristics

We plan to perform experiments that demonstrate the usefulness of the newly introduced procedures on several examples, though we have performed several preliminary experiments to date. In one example, the exlined procedures were converted to forked processes, leading to a reduction from 87 clock cycles to 31 cycles for a part of the specification. Such coarse parallelism would have been difficult to find by a synthesis tool alone. In another example, the introduced procedures increased the granularity of functional partitioning from 8 procedures to 48 procedures, leading to a hardware/software functional partitioning with 30% less hardware (27000 gates reduced to 16250 gates) and 10 less pins that still satisfied performance constraints. It is interesting to note that if we had chosen an even finer granularity of statements, as in many other hardware/software codesign environments, the number of partitions examined by an $n^2log(n)$ algorithm would have increased from 13824 (for the 48 procedures) to 6,400,000 (for 800 statements). Clearly, the latter makes any interactive approach almost impossible.

We also experimented with a 700-line encryption example in VHDL. We exlined eight additional procedures in a process that originally used only four procedures, partitioned the twelve procedures among two parts, and applied a behavioral synthesis tool to each part. The improvements over synthesizing the original 700-line example were substantial: exlining and partitioning reduced the total runtime of a particular behavioral synthesis tool from 1166 seconds down to 230 seconds, and reduced the resulting hardware size from 79,000 gates down to 65,000 gates.

Other works, including [9, 17], also demonstrate the usefulness of procedures in functional partitioning and in behavioral synthesis.

6 Conclusion

We have introduced techniques for procedure exlining. Our encoded string matching solution to finding redundant sequences of statements enabled us to solve the problem with designer interaction and with possible variations in the sequences. Our tree representation and heuristics for finding distinct computations permit a range of solutions based on the quality of results desired by the designer. The procedures created with our technique can have substantial impact on the quality of synthesis tool results, permitting greater potential for better functional partitioning, use of complex functional units, use of multiple controllers, and reduced synthesis memory and runtime requirements.

In summary, the technique serves as a useful part of a system-level transformation tool, helping a specification writer to focus on writing readable specification by providing the means to easily convert such a specification to one suited for synthesis. We plan to develop a tool possessing a suite of system-level transformations (called VTRANS – VHDL Transformations) to assist the designer in modifying the specification before invoking synthesis tools.

References

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

[2] P. Gupta, C. Chen, J. DeSouza-Batista, and A. Parker, "Experience with image compression chip design using unified system construction tools," in *Proceedings of the Design Automation Conference*, pp. 250–256, 1994.

[3] F. Vahid and D.Gajski, "Specification partitioning for system design," in *Proceedings of the Design Automation Conference*, pp. 219–224, 1992.

[4] D. Gajski, F. Vahid, and S. Narayan, "A system-design methodology: Executable-specification refinement," in *Proceedings of the European Conference on Design Automation (EDAC)*, pp. 458–463, 1994.

[5] N. Kumar, R. Vemuri, and R. Vemuri, "Partitioning for multicomponent synthesis from VHDL specifications," in *VHDL International Users' Forum*, pp. 19–28, 1993.

[6] R. Camposano, L. Saunders, and R. Tabet, "VHDL as input for high level synthesis," *IEEE Design & Test of Computers*, pp. 43–49, March 1991.

[7] R. Camposano and R. Brayton, "Partitioning before logic synthesis," in *Proceedings of the International Conference on Computer-Aided Design*, 1987.

[8] R. Walker and D. Thomas, "Behavioral transformation for algorithmic level IC design," *IEEE Transactions on Computer-Aided Design*, pp. 1115–1128, October 1989.

[9] L. Ramachandran, S. Narayan, F. Vahid, and D. Gajski, "Synthesis of functions and procedures in behavioral VHDL," in *Proceedings of the European Design Automation Conference (EuroVHDL)*, 1993.

[10] P. Gutberlet and W. Rosentiel, "Specification of interface components for synchronous data paths," in *Proceedings of the International Workshop on High-Level Synthesis*, pp. 134–139, 1993.

[11] A. Jerraya, I. Park, and K. O'Brien, "Amical: An interactive high-level synthesis environment," in *Proceedings of the European Conference on Design Automation (EDAC)*, pp. 58–62, 1993.

[12] D. Rao and F. Kurdahi, "Partitioning by regularity extraction," in *Proceedings of the Design Automation Conference*, pp. 235–238, 1992.

[13] K. Keutzer, "DAGON: Technology binding and local optimization by DAG matching," in *Proceedings of the Design Automation Conference*, pp. 617–623, 1987.

[14] S. Devadas and A. Newton, "Decomposition and factorization of sequential FSM's," in *Proceedings of the International Conference on Computer-Aided Design*, 1988.

[15] S. Wu and U. Manber, "Fast text searching allowing errors," *Communications of the ACM*, vol. 35, no. 10, pp. 83–91, 1992.

[16] E. Lagnese and D. Thomas, "Architectural partitioning for system level synthesis of integrated circuits," *IEEE Transactions on Computer-Aided Design*, July 1991.

[17] F. Vahid and D. Gajski, "Clustering for improved system-level functional partitioning," in *International Symposium on System Synthesis*, 1995.