Procedure Exlining: A New System-Level Specification Transformation

Frank Vahid
Department of Computer Science
University of California, Riverside, CA 92521
vahid@cs.ucr.edu

Abstract

We introduce a new system-level specification transformation called procedure exlining. Exlining is the problem of replacing sequences of statements by procedure calls, which is the opposite problem of inlining. Procedures are used by system synthesis and behavioral synthesis tools to guide exploration of various high-level implementations, so exlining can greatly improve the results of synthesis. We demonstrate the usefulness of exlining on several examples.

1 Introduction

The focus of system design is shifting to earlier stages of design, as automatic tools mature for the later stages. At the early stages, we focus on creating a functionally-correct specification of the entire system, with VHDL being commonly used as the specification language. Functional correctness means that we have specified correct system output for every possible sequence of inputs, without regard for how the system is eventually implemented. We can simulate the specification extensively to verify correctness, thus eliminating functional errors early and avoiding lengthy changes at later design stages. We can then divide the specification into pieces, and implement each piece as software running on a standard processor, or as a custom-hardware processor, with the aid of synthesis and compilation tools. We may evaluate many possible implementations to tradeoff performance and cost. The various design stages are illustrated in Figure 1.

Because creating a functionally-correct specification is crucial for eliminating late changes, we would like to focus only on functional-correctness when creating the initial specification. Unfortunately, the style in which one writes a functional specification can greatly affect the runtime and output of synthesis tools. For example, an encryption algorithm written in VHDL required over 10 minutes to behaviorally synthesize when written as a single process, but only 30 seconds when written as two processes – the time differences were even more dramatic for logic synthesis. In addition, the two process version synthesized into 10% fewer gates. As a result of the effect of style on synthesis, many specification writers focus on both correctness and synthesis simultaneously. Because writing correct specifications is an extremely difficult task, such simultaneous focus on two different issues usually results in a less readable specification that contains more errors. In addition, the specification will be tuned towards one particular synthesis tool, so it might not be suitable for other synthesis tools or for software compilation. Alternatively, we have observed designers rewriting the original functional specification a number of times, tuning it for synthesis. While this approach ensures a focus on correctness when writing the initial specification, rewriting can be tedious and time-consuming, meaning that functional errors may be introduced.

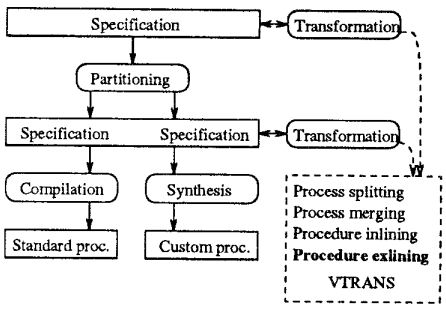


Fig. 1: Exlining as one possible specification transformation before system partitioning or behavioral synthesis.

Our solution to the above problems is to develop a tool to partially automate VHDL transformations, which we call VTRANS. Such transformations might include those that split one process into two, merge two processes into one, or inline a procedure call. A specification writer can then focus on correctness, confident that the task of rewriting will be greatly simplified by the transformation tool.

In this paper, we describe a new transformation called procedure exlining, which involves finding and replacing sequences of statements by procedure calls. This transformation can significantly affect the results of system-level partitioning and synthesis. In Section 2, we describe the purpose of the transformation. In Section 3, we briefly discuss two techniques for finding sequences of statements suitable for exlining. In Section 4, we describe how to replace such sequences by procedure calls. In Section 5, we consider a simple example. In Section 6, we highlight results of several examples. In Section 7, we provide conclusions and future work.

2 Exlining roles

Given a VHDL functional specification, exlining is defined as: (1) Finding sets of sequential statements that should be replaced by procedure calls, and (2) Replacing those sets by procedure calls. We perform exlining at the statement level, rather than decomposing the statements

0-8186-7156-4/95 $4.00

into a control/dataflow graph representation and grouping arithmetic-level nodes, in order to maintain statements similar to the original specification and thus maintain designer interaction. Before describing techniques for the two exlining steps, we must first understand the two roles of exlining: to improve system-level partitioning, and to improve synthesis.

2.1 Exlining for partitioning

Partitioning is the task of assigning pieces of the specification to various implementation parts, like hardware and software parts. Such partitioning requires that we first decompose the specification into a set of functional objects (i.e., basic computations that cannot be divided across parts), and that we then assign those objects to parts. We can decompose a specification to various levels of granularity, such as processes/procedures, basic blocks, statements, or even arithmetic operations. Decomposing to the level of arithmetic operations, as done in [1], has proven useful during synthesis, but may be too fine-grained for the system level. Decomposing to the statement level or basic block level, as done in [2, 3, 4], may yield thousands of functional objects for a VHDL specification possessing several thousand lines. Decomposing to the process/procedure level, as done in [5, 6, 7, 8, 9] yields tens or a few hundred functional objects, which is approximately the level at which designers can interact, and at which inter-object communication times don't dominate over object computation times. However, we have found that designers often write processes or procedures with a large number of statements, meaning that processes and procedures might be too coarse-grained for partitioning. For example, we might not be able to fit an entire process onto a single part, or we might not want to run synthesis on the entire process (as illustrated by the encryption example in the introduction). As another example, we might only need to implement a time-critical piece of a process in custom hardware, assigning the rest of the process to a cheaper software implementation. The decomposition granularity issue is illustrated in Figure 2, where the circles represent functional objects, and the small rectangles represent package size constraints, synthesis-tool input-size constraints, or timing constraints. Figure 2(a) shows that very coarse-grained decomposition can lead to constraint violations, while Figure 2(b) shows that very fine-grained decomposition can lead to an excessive number of possible partitions and to incomprehensible results.

Exlining provides one solution to this problem. Before decomposing to the process/procedural level, we look for processes/procedures with a large number of statements. We then replace closely-related statements by a procedure call, until no process/procedure has more than N statements, or until some other criteria is satisfied. We can then decompose to processes and procedures, yielding a granularity well-suited for functional partitioning, as illustrated in Figure 2(b).

2.2 Exlining for synthesis

Procedures provide a variety of implementation options to a synthesis tool. Specifically, synthesis tools can make

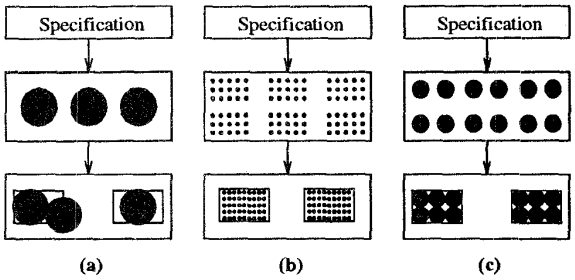


Fig. 2: Exlining for better functional decomposition: (a) procedural-level decomposition, (b) statement level, (c) procedural level with exlining.

better size/performance tradeoffs when presented with a set of procedures than when presented with just a large set of sequential statements without procedures [10]. First, a procedure can represent a complex functional unit, which should be used for some system computations [11, 12], thus enabling reuse of predesigned complex units. Second, a procedure can be used to represent a separate controller in order to simpify the synthesized control logic [13, 14], or to achieve more concurrent execution [15]. Third, a procedure can be implemented as a control subroutine [10]. Fourth, a procedure can be inlined. These options yield implementations with different size and performance characteristics.

Exlining thus permits better exploration during synthesis, without incurring any penalty, since synthesis can always just inline the procedures to obtain the same results as would have been obtained without exlining.

3 Exlining techniques

There are two distinct procedure exlining problems. In the first problem, we find redundant sequences of statements, and replace those sequences by calls to a single procedure; we call this **redundancy exlining**. In the second problem, 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; we call this **isolation exlining**. Each problem requires a different solution technique.

3.1 Redundancy exlining

Instead of trying to solve the very hard problem of finding sequences of statements with identical functionality, we solve the problem of finding sequences with similar statement types, letting the user determine when such sequences should actually be replaced by a procedure call. At the heart of the solution is the use of an existing, powerful pattern matching tool.

First, we encode the VHDL statements into an *encoded string*, by replacing each VHDL statement with a single character based on the statement type. One possible encoding character table is given in Figure 3. For example, the first seven statements in Figure 5 might be encoded

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

Fig. 3: Statement-type encoding characters for VHDL

as "fswvswm," where 'f' represents the beginning of a for loop, 's' represents a signal assignment, and so on.

Second, we search for a designer-provided *pattern* in the encoded string, using the *agrep* pattern matching tool [16], and we display all the matching statements, or *candidates*.

Third, we reduce the number of candidates by requiring target and/or source consistency. A *target* is the identifier being updated in an assignment statement. Target consistency means that if the pattern writes an identifier v with its i'th and j'th statements, then a candidate must also write some identifier u (possibly different than v) with its i'th and j'th statements. Likewise, a *source* of a statement is any identifier that is read by that statement. To extend our encoding for target and source consistency, we first replace all targets and sources in the candidates by symbolic identifiers. Each statement is encoded as $cTtxSls$, where c is the statement type, tx is the target's symbolic identifier, and ls is a list of source identifiers 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 string. We use *agrep* to find the candidates that still match the pattern.

The use of pattern matching for redundancy exlining has several advantages. First, pattern matching tools are widely used and researched, and hence extremely fast. Second, recent research has led to fast search time for **approximate matching**; *agrep* uses sub-linear time. Approximate matching allows us to find matches with minor differences from the pattern, where those differences would be accounted for by procedure parameters. Third, using pattern matching gives us the ability to form patterns using **regular expressions**. Regular expressions permit a powerful, concise method for describing possible variations in the pattern; for example, we might want to search for all *for* loops containing any number of variable and signal assignments, followed by either a wait statement or procedure call. Regular expressions are heavily-used by the Unix community and hence are familiar to many designers.

3.2 Isolation exlining

For isolation exlining, we try to group very close statements into a procedure. "Close" statements are those statements that share much data among them, that are executed in the same thread of control, and that can execute using the same hardware.

We first create a 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 of such a tree represents a process. We extend the representation by adding ***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. 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. By merging at the appropriate level of hierarchy, we can include entire if-then-else statements or case statements into a procedure.

Our problem is to insert procedure nodes into this tree in a manner that results in the best decomposition of the initial statements. The best decomposition is determined using a cost function that is a weighted sum of several terms. *Procedure size* is the variation from a designer-specified number of statements per procedure, summed over all procedures. *Control transfer* 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. *Data interconnect* 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* 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* is the total synthesized hardware size assuming each procedure is synthesized independently. This term encourages groupings in which large hardware items, such as multipliers, appear in the same procedure so can be shared.

We have defined three techniques to solve the above problem, spanning the spectrum of fast heuristics to computationally intensive heuristics. The first technique, the *naive* hueristic, simply inserts a procedure node for every hierarchical statement. 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. We prohibit merges that would exceed the maximum procedure size, which in turn provides a condition for terminating the clustering. Alternatively, we can terminate clustering based on a closeness threshold; if no nodes are closer than some minimum threshold value, then we terminate clustering. The third technique uses *simulated annealing*. Given an initial tree with procedure nodes, we attempt to improve the cost function value through a series of changes.

4 Creating the new procedures

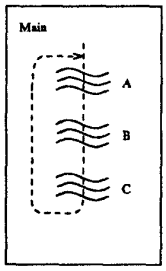
Creating a new procedure for a given sequence of statements requires attention in two areas. First, in the case that a single procedure will replace several redundant sequences, we must add parameters and provide conditionally executed statements within the procedure to account for variations among the sequences. Presently, such details

are handled manually.

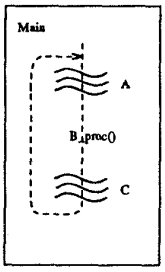
Second, in the case there is just a single sequence being replaced, we must determine the procedure's parameters. We now briefly discuss such parameters. Consider the example of Figure 4(a). A process called *Main* is shown consisting of three sequences of statements, labeled A, B and C. Suppose that B is to be exlined into a new procedure, *B_Proc*. To determine the parameters and local variables of *B_proc*, we must examine the use of all symbols accessed in B. We can distinguish two regions of *Main*: B, and C,A. Note that C,A both precedes and follows B, since the process loops back to its beginning. We can distinguish four types of accesses by either region to a given symbol s:

- *w_only* means that the region only writes s.
- *r_only* means that the region only reads s.
- *rbw* means that the region might read s before writing to it.
- *wbr* means that the region always writes s before reading it.

At first glance, the last two possibilities don't seem to cover all cases where a region both reads and writes s, but they do. The opposite of *rbw* "might read before writing" is "never reads before writing", which is the same as "always writes before reading," or *wbr*. Likewise, the opposite of *wbr* is "sometimes writes before reading", which is the same as *rbw*.



(a)



(b)

A,C	B	Action
r_only	r_only	Invalid -- report error
r_only	w_only	Move to B_Proc, out parm
r_only	rbw	Move to B_Proc, out parm
r_only	wbr	Move to B_Proc, out parm
w_only	r_only	In parm
w_only	w_only	Dead variable -- delete
w_only	rbw	In parm
w_only	wbr	Move to B_proc
rbw	r_only	In parm
rbw	w_only	Out parm
rbw	rbw	Inout parm
rbw	wbr	Out parm
wbr	r_only	In parm
wbr	w_only	Dead variable in B -- delete from B
wbr	rbw	In parm
wbr	wbr	Duplicate in B_proc

(c)

Fig. 4: Handling symbols during procedure creation: (a) original Main process, (b) new process with B extracted, (c) determining data parameters between B and Main.

Given those four access types, there are 16 possible combinations of accesses of a given symbol s by regions A,C and B, as illustrated in Figure 4(c). The first row corresponds to the situation where both regions only read the symbol; such a situation is a specification error. The next three rows correspond to B writing before A,C reads. In this case, we prefer to move the symbol's declaration into *B_proc*, returning the written value as an output parameter. The fifth and seventh rows correspond to A,C writing and B reading the symbol, so the symbol becomes an input parameter. The sixth row corresponds to a dead symbol that is written but never read, so we can just delete all occurrences of it. The eighth row corresponds to the symbol being dead in A,C, so we move it to *B_proc*. The remaining rows follow similarly. A particularly interesting row is the last one, in which the symbol is written before being read in both regions; in this case, we duplicate the symbol declaration within *B_proc*.

```
-- Collect data for this x,y location
for i in 1 to n loop
   data_strobe <= '1'; wait until rdy='1'
   M(i) := data;
   data_strobe <= '0'; wait until rdy='0';
end loop;
flat_level := (M(n-2) + M(n-1))*8 + 10;
-- Find anterior wall
for i in 1 to n loop
   if (M(i) < flat_level) then
      anterior_wall := i;
      exit loop;
   end if;
end loop;
-- Find posterior wall
for i in anterior_wall to n loop
   if (M(i) >= flat_level) then
      posterior_wall := i;
      exit loop;
   end if;
end loop;
-- Write data to disk for later analysis
for i in 1 to n loop
   diskport <= M(i);
   write_disk <= '1'; wait for 1 us;
   write_disk <= '0'; wait for 1 us;
end loop;
...
```

Fig. 5: Example VHDL specification.

5 Example

We now consider a simple example, which is trivially small and is used simply to demonstrate the exlining techniques. The example is a small portion of a VHDL behavioral description of a volume-measuring medical instrument, shown in Figure 5.

We choose to initially perform redundancy exlining. The exlining tool would encode the statements, using the encoding characters given in Figure 3, as: *fswvswmvfivbjmfivbjmfsswswm* . Noting that the wall-search statements are similar, we search for *fivbjm* – the exlining tool finds two matches, corresponding to the statements that find the anterior wall and the posterior wall. We use the exlining tool to see if these two matches are target consistent. The first match would be encoded as: *fTt1 i vTt2 b j m*,

```
-- Collect data for this x,y location
for i in 1 to n loop
   data_strobe <= '1'; wait until rdy='1'
   M(i) := data;
   data_strobe <= '0'; wait until rdy='0';
end loop;
flat_level := (M(n-2) + M(n-1))*8 + 10;
FindWall
   (M,1,n,flat_level,ant,ant_wall);
FindWall
   (M,ant_wall,n,flat_level,post,post_wall);
-- Write data to disk for later analysis
for i in 1 to n loop
   diskport <= M(i);
   write_disk <= '1';  wait for 1 us;
   write_disk <= '0';  wait for 1 us;
end loop;
...
```

Fig. 6: After redundancy exlining.

```
CollectData(M,n);
flat_level := (M(n-2)+M(n-1))*8 + 10;
FindWall
   (M,1,n,flat_level,ant,ant_wall);
FindWall
   (M,ant_wall,n,flat_level,post,post_wall);
WriteData(M,n);
```

Fig. 7: After isolation exlining.

where $t1$ and $t2$ correspond to i and *anterior_wall*. The second match would be encoded identically, meaning the two matches are target consistent. We then decide to replace the two matches by a procedure call, as shown in Figure 6.

Next, we perform isolation exlining. We might instruct the exlining tool to create procedures with no more than 7 statements. The exlining tool would build a tree representation and apply one of the partitioning heuristics. The result would be a suggested procedure for the first *for* loop and another procedure for the last *for* loop. These procedures yield the fewest control transfers. If we had specified a smaller maximum size of 5 statements, then the tool would have grouped each loop's inner statements into its own procedure also; likewise for the loop inside the *FindWall* procedure. The final result is shown in Figure 7.

6 Results

We have implemented a prototype procedure exlining tool, and incorporated it with a VHDL transformation tool (VTRANS) consisting of 26,000 lines of C code. VTRANS reads VHDL and displays a graphical tree, where each node represents a process, procedure or data declaration from the VHDL, and edges represent the declaration hierarchy. The user can quickly view the contents of a node by clicking on that node. Another command highlights the accessor or accessee nodes of any given node. Commands exist to convert a procedure to a process, and to inline a procedure. The exlining command can be performed on a particular process/procedure or on the entire specification. The exlining tool displays candidates, and prompts the designer to exclude candidates or change matching criteria or cost functions. Presently, new-procedure creation is manual.

We have applied redundancy exlining on an image processing example and the i8251 high-level synthesis benchmark, both written by outside sources. In the former, we introduced 4 new procedures and replaced statement sequences by 18 procedure calls (in turn reducing 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 VTRANS. In the latter example, we introduced five procedures and eleven procedure calls (and reduced the code size by 40 statements). In addition, the process of exlining caused us to find an error in which two redundant sequences had a minor unintentional variation.

We applied isolation exlining on a number of procedures from an image processing example and an MPEG decoding example, both obtained from outside sources. The naive heuristic averaged one second, clustering averaged twenty seconds, and simulated annealing averaged eight minutes. Simulated annealing yielded the lowest-cost results, and the naive heuristic yielded the highest-cost results.

In the image processing example, two exlined procedures were converted to forked processes, leading to a reduction from 87 clock cycles down to 31 cycles for a part of the specification. Such coarse parallelism would have been difficult to find by a synthesis tool alone. In the MPEG 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). The latter makes designer interaction 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 (with even greater savings in logic synthesis), and reduced the resulting hardware size from 79,000 gates down to 65,000 gates.

Several other uses of exlining have been suggested. One is to search a specification for statement sequences that could be replaced by a predesigned complex functional unit of the type described in [12]. Another is to reduce software size when there is limited program memory. A third is to

search large numbers of software files to determine if a procedure has already been written, without knowing the name of that procedure.

7 Conclusion

We have shown that exlining is an important part of a VHDL system-level design tool. Exlining is necessary to decompose large processes and procedures into a suitable size for functional partitioning, and to provide behavioral synthesis tools with the ability to explore various high-level implementation options. Given an exlining tool, the writer of a system's initial specification can focus on describing the system in the most natural manner possible, using procedures only when necessary for readability. The designer need not consider how procedures will be used for functional partitioning or in behavioral synthesis; instead, the exlining tool can be used to help easily convert an initial specification written for readability into one suitable for partitioning or synthesis. Tools that provide such a transformation ability, and those that provide unique views of the specification, will likely become increasingly important as design effort shifts to the specification level.

References

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

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

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

[4] X. Xiong, E. Barros, and W. Rosentiel, "A method for partitioning UNITY language in hardware and software," in *Proceedings of the European Design Automation Conference (EuroDAC)*, 1994.

[5] 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.

[6] 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.

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

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

[9] P. Eles, Z. Peng, and A. Doboli, "VHDL system-level specification and partitioning in a hardware/software co-synthesis environment," in *International Workshop on Hardware-Software Co-Design*, pp. 49–55, 1992.

[10] 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.

[11] 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.

[12] 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.

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

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

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

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