A Generalized Control-Flow-Aware Pattern Recognition Algorithm for Behavioral Synthesis

Jason Cong, Hui Huang and Wei Jiang
Department of Computer Science, University of California
Los Angeles, USA
Email: {cong, huihuang, wjang}@cs.ucla.edu

ABSTRACT— **Pattern recognition has many applications in design automation. A generalized pattern recognition algorithm is presented in this paper which can efficiently extract similar patterns in programs. Compared to previous pattern-based techniques, our approach overcomes their limitation in handling control-flow-aware patterns, and leads to more opportunities for optimization. Our algorithm uses a feature-based filtering approach for fast pruning, and an elegant graph similarity metric called the *generalized edit distance* for measuring variations in CDFGs. Furthermore, our pattern recognition algorithm is applied to solve the area optimization problem in behavioral synthesis. Our experimental results show up to a 40% area reduction on a set of real-world benchmarks with a moderate 9% latency overhead, compared to synthesis results without pattern extractions; and up to a 30% area reduction, compared to the results using only data-flow patterns.**

Keywords: Behavioral Synthesis, control flow, pattern, feature

I. INTRODUCTION

Regularity plays a very important role in circuit design, which leads to fewer multiplexing logics, better resource sharing and better physical layout. We believe that efficiently handling regularity is one of the advantages of designers in manual RTL coding. However, the tight time-to-market requirements have made it increasingly difficult for designers to thoroughly analyze every design in a short time; therefore, *pattern recognition*, which helps designers automatically extract and utilize regularities in applications of design optimizations, has drawn wide interest at every level of circuit designs, from layout to behavioral synthesis.

Among all the pattern recognition algorithms, the graph-matching-based approach provides the best trade-off between efficacy and scalability. Recently, these approaches have been applied to solve problems in various domains, such as data mining, biochemistry, and behavioral synthesis [4, 8, 9, 10, 11, 13]. The work in [10] generates the candidate templates with a clustering algorithm based on occurrence frequency; the work in [9] proposed a polynomial time algorithm with respect to the input/output port number; another approach in [11] constructs subgraphs by combining single input cones. All three are designed for customized instruction generation for ASIP synthesis. Recently, a pattern-based behavioral synthesis framework was proposed in [4]. It uses the concept of *graph edit distance* to capture graph similarity. However, all the work mentioned above are constricted to extracting patterns only in data flow graphs, without considering higher level control flow structures. In [17] both control and data flow level differences have been considered for program partition, but their method merely compares signatures between two partitions without accurate similarity evaluation.

In fact, control flows, such as loops and branches, have big impacts on the final QoR in many cases and introduce more opportunities for optimizations. The challenges for pattern extraction in CDFGs are quite different from that in DFGs: first, CDFG graphs have two kinds of edges — control flow edges and data flow edges. Second, the matching between two collections of graphs needed in the related control-flow-aware pattern recognition process can not be directly solved by known pattern matching techniques which are applicable to only two graphs.

In this paper we develop a generalized control data flow pattern recognition algorithm to identify regularity in the behavioral specification and help users to reduce the resource usage of a certain design. To our best knowledge, this is the first automated approach to considering control-flow patterns for behavioral synthesis. In particular, the contributions of our approach include:

(1) A general hierarchical approach which automatically recognizes control data flow patterns from programs.
(2) A graph similarity evaluation metric to estimate *general edit distance* between CDFG patterns.
(3) A two-level feature-based filtering scheme to reduce the amount of expensive similarity evaluations effectively.
(4) An efficient and accurate pattern selection strategy which helps to select optimal pattern combinations from discovered patterns.

The remainder of the paper is organized as follows. Section II discusses preliminaries knowledge; Section III presents our CDFG pattern recognition algorithm and related techniques; pattern selection strategy is discussed in Section IV; Section V shows the experimental results, and Section VI is the conclusion part.

II. PRELIMINARIES

DEFINITION 1. *A **control data flow graph (CDFG)** is a directed graph $G(V_G,E_G)$ where $V_G = V_{bb} \cup V_{op}$ and $E_G = E_c \cup E_d$. V_{bb} is a set of basic blocks. V_{op} is the entire set of operation nodes in G. Data edges in E_d denote the data dependencies between operation nodes. Control edges in E_c represent control dependencies between basic blocks.*

In the control-flow-aware pattern mining problem, a CDFG basic block normally consists of multiple data flow graphs, as shown in Figure 1, therefore the original *edit distance* [1] which measures the similarity between two connected graphs is no longer applicable. To solve this problem we introduce a new concept called *generalized edit distance*:

DEFINITION 2. *The **generalized edit distance (GED)** between two sets of labeled graphs GS_1 and GS_2 is the minimal number of edit operations (insert/delete/relabel a node/edge) to transform GS_1 to GS_2.*

In our approach, the concept of *generalized edit distance* is exploited to describe the similarity degree of CDFGs. For example, the *generalized edit distance* between CDFGs in Figure 1(a) and 1(b) is three, since relabeling the node "+" with "*" at three places in 1(a) can transform it to the graph in 1(b). *GED* provides an effective metric to capture the similarity of CDFG subgraphs with more than one data flow graphs inside, and also gives the minimum number of edit operations needed to share common structures among similar subgraphs.

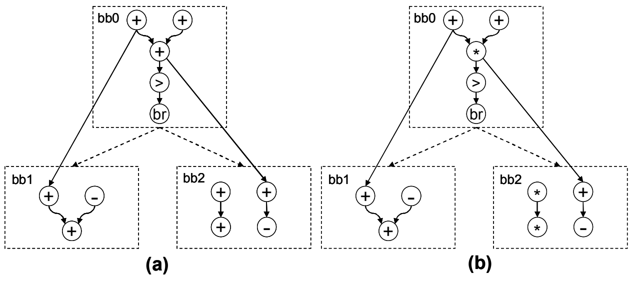


Figure 1: Two sample CDFG subgraphs.

With the concept of *generalized edit distance*, *CDFG pattern* and *pattern instances* are defined as follow:

DEFINITION 3. *Given a set of CDFGs $\{G_i \mid i = 1, 2 \ldots N\}$, a threshold l_{dist} as an upper bound for the generalized edit distance, and a significance threshold l_{sig}, a labeled graph $\mathbb{P}$ is called a* ***CDFG pattern*** *if there exists set $\mathbb{S}(\mathbb{P}) = \{ SG_j \}$ which satisfies: (1) for all $SG_j \in \mathbb{S}$, $GED(SG_j, \mathbb{P}) \leq l_{dist}$; (2) for any $SG_j \in \mathbb{S}$, there exists G_i such that SG_j is a subgraph of G_i; (3) $|S(P)| * |P| \geq l_{sig}$. In this case, each subgraph in set $\mathbb{S}$ is called a* ***CDFG pattern instance*** *of the pattern $\mathbb{P}$, and correspondingly set $\mathbb{S}$ is called a* ***CDFG pattern group****.*

III. PATTERN RECOGNITION

This section introduces our generalized pattern recognition algorithm, which is used to discover patterns with similar control flow, as well as data flow structures. The similarity between two CDFG patterns is measured in terms of *generalized edit distance*, and a feature-based filtering technique is applied to reduce total number of GED computations.

A. CDFG Subgraph Labeling

In our approach each pattern is represented by a labeled graph. Notice that a DFG node can be labeled by the operation it represents, while each supernode in CFG is a basic block and it is not straightforward to find a label. Thus further similarity evaluation technique discussed in Section III(C) is used to decide whether two supernodes in CFG can be assigned the same label or not. Once the label of a basic block has been obtained, we will append it to the original DFG label of the nodes it contains. Therefore the label of a node in a given CDFG graph is a combination of its own DFG label and the label for the supernode (basic block) it belongs to. And distinct labels are assigned to data dependence edges according to their commutability.

B. Two-level Feature-based Filter

The computation of *generalized edit distance* is expensive, considering currently there is no polynomial-time algorithm for solving the graph isomorphism problem. Therefore filtering techniques have been adopted in the pattern recognition algorithm to reduce the number of *GED* calculations. In our approach, a signature called two-level *feature vector* is introduced for each CDFG subgraph, based on the work in [4]. However, the features discussed in [4] are constricted to data flow graph. In our control-flow-aware approach the feature information needs to be collected at both levels.

DEFINITION 4: *In a CDFG graph $G = (V_G, E_G)$, a* ***DFG-feature*** *is a subgraph $S = \{u, l, r \mid u, l, r \in V_{op}\}$, such that (l, u) and $(r, u) \in E_d$; a* ***CFG-feature*** *is a subgraph $S = \{u, l_1, \ldots, l_m \mid u, l_i \in V_{bb}\}$, such that $(u, l_i) \in E_c$ (m equals the number of outputs for supernode u). For a given universe of distinct features $U = \{F_1, F_2, \ldots, F_{|\tau|}\}$, the* ***feature vector*** *is a vector $(f_1, f_2, \ldots f_{|\tau|})$ with each element f_i representing the number of occurrences of the i^{th} feature in U. F_i could either appear in the DFG or CFG graph, and the corresponding feature vector is called* ***DFV*** *and* ***CFV*** *respectively.*

Figure 2 shows the features for a DFG graph and a CFG graph, along with their feature vectors. The symbol ε denotes an empty node if the input or output of a node is not in this graph. As discussed in [4], DFG nodes with multiple outputs will also be counted as ε node. Notice that we assume supernode 1 and 2 are similar, therefore the number of the second feature in (c) is two instead of one.

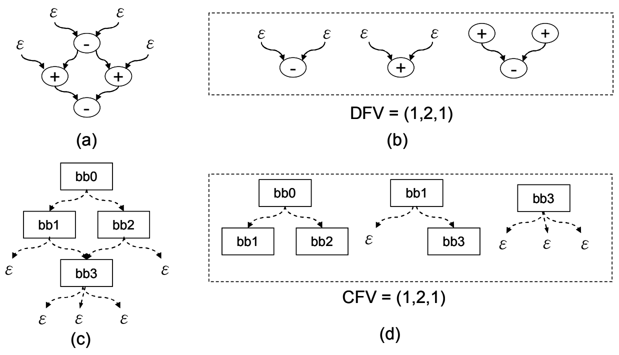
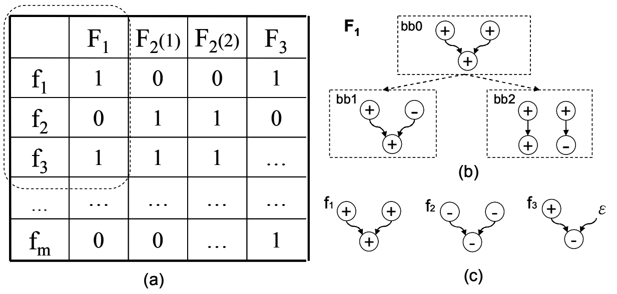


Figure 2: (a) A DFG. (b) Data flow features and the DFV of the given DFG. (c) A CFG. (d) Control flow features and the CFV of the given CFG, assuming bb1 and bb2 are similar supernodes.

The combination of DFV and CFV is used in our approach to capture structural properties of the original CDFG graph. The L_1 distance of two feature vectors, which is easy to compute, serves as an indicator of the similarity degree between the corresponding two graphs.

Theorem 1 [4]**.** *Let $d(G_1; G_2)$ be the edit distance between two data flow graphs G_1 and G_2, and $DFV(G_i)$ be the data flow feature vector of G_i, $||DFV(G_1)-DFV(G_2)||_1 \leq 4 * d(G_1; G_2)$.*

Theorem 1 from [4] tells us that given an edit distance limit l_{dist}, the maximal number of possible DFG feature misses between two CDFG subgraphs is $4 * l_{dist}$. However, this reveals no information for the similarity degree in their control flow graphs.



	F_1	$F_2(1)$	$F_2(2)$	F_3
f_1	1	0	0	1
f_2	0	1	1	0
f_3	1	1	1	…
…	…	…	…	…
f_m	0	0	…	1

Figure 3: (a) a sample *feature map matrix* (b) a sample CFG feature F_1 (c) three sample DFG features

In order to develop an upper bound for the number of possible CFG feature misses, we propose a data structure called *feature map matrix*. Each row of the *feature map matrix* corresponds to a DFG feature, while each column corresponds to a CFG feature. Each entry records whether a DFG feature appears in a target CFG feature. Figure 3 shows the *feature map matrix* built for the CDFG in Figure 2(c), with its CFG/DFG features. F_i in Figure 3(a) corresponds to the i^{th} CFG feature in Figure 2(d) and all the occurrences of a CFG feature will be recorded in the matrix. For instance, columns 3 and 4 are two occurrences of F_2 in the original CDFG. In Figure 3(b) and 3(c), we can observe that DFG features f_1 and f_3 appear once in CFG feature F_1, therefore the corresponding entries are set to one.

Now, with the newly built *feature map matrix M* and the maximal number of allowed DFG feature misses $4*l_{dist}$, we can find out the upper bound for possible CFG feature misses l_{missUb}. If a DFG feature f_i is missing, all the CFG features containing f_i in their inside data flow structures will be destroyed correspondingly, namely any column j satisfying $M(i,j) = 1$ needs to be removed from the matrix. Since the total number of missing DFG features is no more than $4*l_{dist}$, any set of missing CFG features can not cover more than $4*l_{dist}$ rows. Therefore l_{missUb} equals the maximal number of columns hit by $4*l_{dist}$ rows in M — here "column *j* is *hit* by row *i*" means $M(i,j)$ equals one.

Theorem 2. *Let l_{dist} be the given edit distance limit, and $CFV(G_i)$ be the CFG feature vector of G_i. If the generalized edit distance between G_1 and G_2 does not exceed l_{dist}, we have: $||CFV(G_1)-CFV(G_2)||_1 \le l_{missUb}$*
Proof: Assume the generalized edit distance between G_1 and G_2 is less than l_{dist}, Theorem 1 tells us that the DFV distance between G_1 and G_2 is no more than $4*l_{dist}$, therefore their CFV distance $||CFV(G_1)-CFV(G_2)||_1$ can not exceed l_{missUb}, which is the maximal CFG feature misses brought by $4*l_{dist}$ DFG feature misses.

To find the maximal number of columns hit by *k* rows is a classic max-cover problem, which has been proved to be NP-complete[18]. For the runtime trade-off, we can approximate the optimal solution by using a greedy algorithm. The greedy algorithm first selects a row that hits the largest number of columns, then removes this row and the columns covering it. This operation will continue until $4*l_{dist}$ rows have been selected. $l_{missGreed}$ equals the number of columns removed by this algorithm.

Theorem 3. [14] *Let $l_{missGreed}$ and l_{missUB} be the greedy and optimal solution for the max-cover problem, we have:*

$$l_{missUB} \le (1-(1-\frac{1}{k})^k)^{-1} \cdot l_{missGreed} \le \frac{e}{e-1} \cdot l_{missGreed} \approx 1.59 l_{missGreed}$$

Theorem 3 from [14] shows the optimal solution is approximated by the greedy solution within a ratio of $e/(e-1)$, therefore $1.59* l_{missGreed}$ can be used as an estimation of l_{missUb}. If the actual CFG feature misses between two graph candidates are greater than l_{missUb}, we can safely assert that the difference in their control flow structure will result in a generalized edit distance larger than l_{dist}, which means that further GED computation is not necessary. Experimental results show that on average only about 3 to 7 GED calculations are needed with hundreds of pattern candidates, which brings dramatic runtime speedup.

C. *Similarity Evaluation*

Given two CDFG subgraphs G_1 and G_2 which have passed the two-level feature-based filter, similarity evaluation will first be performed between their control flow structures, in which each basic block is treated as a supernode. After that, we will look into the data flow graphs inside to do further comparison.

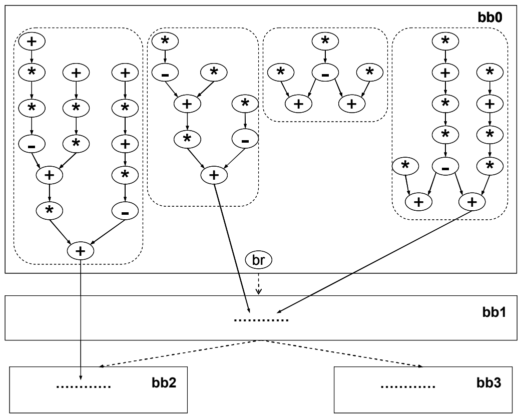


Figure 4: Four subgraph fragments in CDFG of BH

Figure 4 shows the data flow structure inside a supernode in one of our test cases – BH. We can observe that the data flow structure inside each supernode is not connected and consists of several separate subgraphs which we call *subgraph fragments*, marked with dashed rectangle in Figure 4. This situation is very common. Here we do not allow edge insertion between two separate subgraph fragments, since it will introduce data dependency between two originally parallel data flow graphs, and make it hard to measure the latency of a pattern. Under this constraint we can approximate *generalized edit distance* as below:

Given two sets of subgraph fragments SF_1 and SF_2, in which $SF_1 = \{ fg_{11}, fg_{12}, ... fg_{1N}\}$ and $SF_2 = \{ fg_{21}, fg_{22}, ..., fg_{2N}\}$, and fg_{ij} is the j^{th} subgraph fragment in set i. If the edit distance between fg_{1a} and fg_{2b} is $dist(fg_{1a}, fg_{2b})$, the ***generalized edit distance (GED)*** *between SF_1 and SF_2, denoted by $DIST(SF_1, SF_2)$, is approximated as:*

$$\min \{ \sum_{i=1}^{N} dist(fg_{1i}, fg_{2p_i}) \}$$

where $(p_1,p_2,..,p_N)$ is a permutation of $(1,2, .., N)$.

If the numbers of *subgraph fragments* of the two sets are different, we can introduce dummy graphs with 0 nodes and 0 edges to make it equal. To calculate *GED* between two sets of subgraph fragments, we construct a *fragment-edit-distance* matrix as follow:

DIST	fg_{21}	fg_{22}	…	fg_{2n}
fg_{11}	9	7	…	∞
fg_{12}	2	10	…	5
…	…	…	…	…
fg_{1n}	7	∞		1

Figure 5: Fragment-Edit-Distance Matrix M

Entry $M(i,j)$ in Figure 5 records the edit distance between the i^{th} subgraph fragment in set 1 and the j^{th} fragment in set 2. When we build M, the feature vector distance constraint $||DFV(fg_{1i}) - DFV(fg_{2j})||_1 \le 4*l_{dist}$ must be satisfied, otherwise, an infinite value will be assigned to the corresponding entry. In our experiments, we find that for most cases, the number of fragments with more than ten nodes is less than five; therefore, even though we need to compute edit distance between every two fragments, the cost is still acceptable.

With the *fragment edit-distance matrix*, our problem is to find an optimal index permutation $(p_1,p_2,\ldots,p_N)$ of $(1,2,\ldots,N)$, so that the sum of edit distance between the i^{th} fragment in the first set and p_i^{th} fragment in the second set is minimal, for i = 1 to N. This problem is similar to the *assignment problem*, and can be formulated as below:

$$
\begin{aligned}
&\textit{minimize} && \sum_{i=1}^{N}\sum_{j=1}^{N}\mathrm{dist}(i,j)\cdot X_{ij} \\
&\textit{s.t.} && \sum_{j=1}^{N} X_{ij} = 1 \quad \text{for i=1...N} \\
& && \sum_{i=1}^{N} X_{ij} = 1 \quad \text{for j=1...N} \\
& && X_{ij} \geq 0 \qquad \text{for i,j=1...N}
\end{aligned}
$$

In the optimal solution, if the m^{th} subgraph fragment in the first set is matched to the n^{th} fragment in the second set, X_{mn} will be set to 1, otherwise 0. Hopfield network has been developed to solve this problem efficiently in polynomial time [2].

D. *Pattern Recognition Algorithm*

With all the important techniques discussed, our pattern recognition algorithm is presented in this section. The algorithm does pattern extraction in both data flow graphs and control flow graphs, and the two-level recognition process is performed in an interweaving way.

Our CDFG pattern recognition process discovers patterns in a breadth-first order. All the subgraphs of size k in a given CDFG are enumerated, among which the non-similar subgraphs will be directly removed by our proposed two-level feature-based filter. Then the remaining subgraph pairs will perform *GED* calculation to measure their similarity accurately. The pseudo code of the algorithm is shown in Algorithm 1.

Our algorithm iteratively finds patterns of size k starting from k = 1. At step k + 1, all the size k CDFG pattern instances are extended by one supernode using the subgraph enumeration techniques discussed in Section III(A). If a subgraph s_k is not a pattern instance of a certain pattern P at step k, it is impossible to be a subgraph of another pattern instance larger than k, which means we don't need to further extend it.

Lines 15 to 25 show the pruning process in Algorithm 1. After a new subgraph s_{k+1} is generated, it will be compared to the existing patterns by calculating the CFG level edit distance between itself and existing patterns. First, the CFV of a subgraph is calculated and used as a signature to find the patterns which have similar control flow structures. After getting the list of possible pattern candidates, the generalized edit distances are calculated by the techniques discussed in Section III(C). If s_{k+1} matches a pattern P, it will be added to the pattern instance list of P, otherwise a new pattern will be generated based on s_{k+1}. When all size k + 1 subgraphs are processed, each newly generated pattern will be examined to see whether it satisfies the significance limit, which is measured by the pattern size times the number of its instances. If not, the pattern itself and its instances will be removed together.

Algorithm 1 CDFG Pattern Recognition

```
1: $\mathbb{P}$          → set of discovered patterns
2: $\mathbb{S}_k$        → set of size k CDFG subgraphs
3: INST($P$) → instances of a pattern $P$
4: $l_{dist}$     → generalized edit distance limit
5:$l_{missUb}$   → estimated upper bound for CFG feature misses
6: $l_{sig}$      → significance limit
7: $l_{min}$      → minimal DFG edit distance increase by one
                 CFG edit operation.
8:
9: Travel all supernodes, add size 1 CFG patterns and
     instances to $\mathbb{P}$ and $\mathbb{S}_1$
10: Compute all DFG and CFG features, create feature map
     matrix, and compute $l_{missUb}$
11: for k = 1 to N-1 do
12:     for all $s_k \in \mathbb{S}_K$ do
13:         Add a neighbor expand $s_k$ to $s_{k+1}$;
14:         Calculate CFV($s_{k+1}$);
15:         Get a list of patterns $\{P_i\}$ which satisfies
                $||CFV(P_i)-CFV(s_{k+1})|| \leq l_{missUb}$;
16:         Calculate CFG-level edit distance $l_{ed}$ of $s_{k+1}$
            with each $P_i$;
17:         if $l_{min} * l_{ed} \leq l_{dist}$ then
18:             Build fragment ed. matrix $M_{fg}$;
19:             Compute the generalized edit distance
                    $dist(P_i,s_{k+1})$ from $M_{fg}$;
20:             if $dist(P_i,s_{k+1}) < l_{dist}$ then
21:                 Add $s_{k+1}$ to INST($P_i$);
22:         if $s_{k+1}$ matches no current patterns
23:             Create a new pattern based on $s_{k+1}$, add to
                $\mathbb{P}$;
24:             Add $s_{k+1}$ to $\mathbb{S}_{k+1}$;
25: for all new pattern $P_i \in \mathbb{P}$ do
26:     if $|INST(P_i)|*|P_i| < l_{sig}$ then
27:         Remove $P_i$ from $\mathbb{P}$;
28:         Remove INST($P_i$) from $\mathbb{S}_{k+1}$;
```

IV. PATTERN SELECTION

Given a set of patterns discovered by the pattern recognition process, pattern selection algorithms attempt to find an appropriate set of patterns with the largest gain.

$$gain(P) = area(P) * (\#inst(P) - 1) - area_{overhead} + \alpha \cdot \frac{|P|}{latency(P)}$$

#inst(P) is the number of instances of P and $area_{overhead}$ is the area overhead brought by newly introduced multiplexors when resources are shared among similar patterns. *The second term measures the "flatness" of P — "flat" means the critical path of P is small compared to the number of nodes in P. The variable* α *is a user-given parameter to trade off between resource reduction and latency overhead.* Now, with a given set of pattern candidates, the problem is how to find a subset of non-conflicting patterns to maximize total gain. Here "non-conflicting" means non-overlapping, and no loop will be in control flow graph formed after selecting a certain set of patterns.

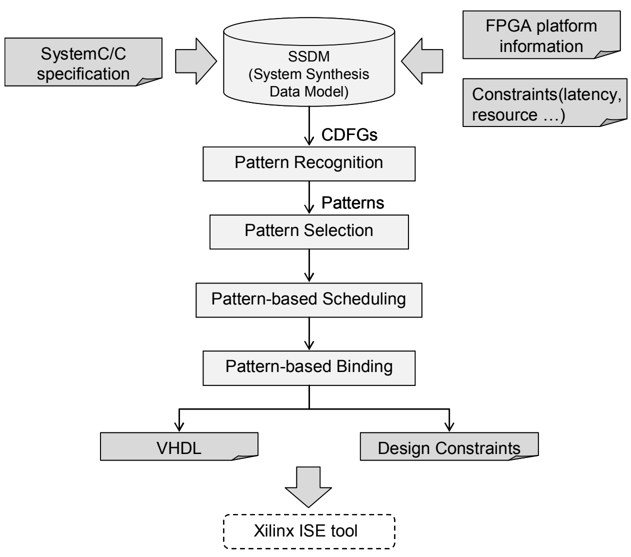
For example, given a CDFG graph G which consists of seven supernodes, indexed from 0 to 6, assume our pattern recognition algorithm finds three patterns P_0, P_1 and P_2 in G. The corresponding pattern groups are denoted by { P_0 | 1 ; 2 }, { P_1 | 0 1 ; 1 3 ; 4 5 } and { P_2 | 3 4 ; 5 6 }. That is, pattern P_0 has two 1-node instances, and the node index for each instance is 1 and 2; P_1 has three 2-node instances, and the node indices for each instance is 0 and 1, 1 and 3, 4 and 5; P_2 has two 2-node instances with node indices 3,4 and 5,6.

Assume no data flow loops exist among the three patterns. To exclude the overlapping case, each pattern group first will be partitioned to eliminate intrinsic instance overlapping. In the example above, the first and second instance of pattern P_1 have a common node 1, thus the corresponding pattern group needs to be partitioned. After that we get four new pattern groups: $\{ P_0 | 1 ; 2 \}$, $\{ P_1 | 0\ 1 ; 4\ 5 \}$, $\{ P_2 | 1\ 3 ; 4\ 5 \}$, $\{ P_3 | 3\ 4 ; 5\ 6 \}$. .

A CNF representation $F(p_0 , \ldots , p_3)$ is used in our approach to describe the *non-overlapping* constraint among pattern group candidates. If p_0 is set to 1, the corresponding pattern P_0 will be selected; therefore those pattern groups having instances overlapping with the instances of P_0 will be invalidated. For example, pattern group 0 has two one-node instances; these nodes also appear in pattern group 1 and 2, then its *non-overlapping* constraint can be represented by setting $f_0 = (\neg p_0 + \neg p_1) \bullet (\neg p_0 + \neg p_2)$ to 1. Similarly, for pattern group 1, $f_1 = (\neg p_1 + \neg p_2) \bullet (\neg p_1 + \neg p_3) = 1$, etc.

Based on the discussion above, the final CNF constraint is $F(p_0, p_1, p_2, p_3) = f_0 \bullet f_1 \bullet f_2 \bullet f_3 = 1$, and our objective is to maximize total area reduction. With this formulation, our problem can be reduced to a *Binate Covering problem*, and the bounding technique in [16] can be used to compute the optimal solution.

V. EXPERIMENTAL RESULTS



F

igure 6: Pattern-based behavioral synthesis flow

The pattern recognition framework can be applied to many practical problems, including ASIP custom instruction set selection, and resource sharing in high-level synthesis, etc. Here, we evaluate the performance of our approach in one of its applications – FPGA area reduction in high level synthesis.

A. *Experimental Setup*

The pattern-based synthesis flow is implemented in our behavioral synthesis system [5]. The whole design flow is shown in Figure 6. Our synthesis tool takes high-level languages and parses them into control data flow graphs. The GMT toolkit [3] is used for graph edit distance calculation. The synthesis engine will perform the pattern-based synthesis flow to reduce resource usage with certain design constraints. Xilinx Virtex-4 FPGA and ISE 9.1 tool [15] are used in our experiments.

Our test cases include a set of real-life programs: IDCT, SYNFLIT, BH, BLKSORT, HEAP, and LEXTREE. All test cases contain common control flows, and the last three cases are from the SPEC2006 benchmark.

B. *Pattern Recognition Results*

In this section we discuss the pattern recognition results. Figure 7 shows the two selected pattern instances in one of our test cases IDCT. There are two similar *for* loops in IDCT, and the corresponding pattern instances include basic blocks 1,2 and basic blocks 3,4 respectively.

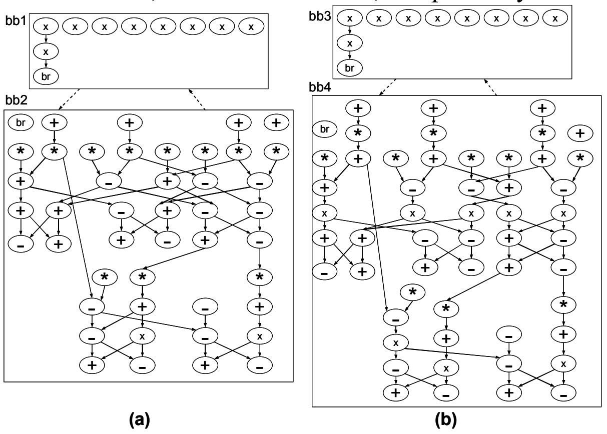


Figure 7: Two selected pattern instances of IDCT

The effectiveness of our proposed pruning techniques using FV is shown in Table I and we can observe a substantial decrease in the number of average GED computations. In Table I, #Line is the size of each test case, #Pattern and #Inst is the number of patterns and total instances respectively, #Avg. Calc is the average number of GED calculations needed before a subgraph matches with a certain pattern, and #MAX is the maximal size of patterns in terms of DFG nodes.

TABLE I. PATTERN RECOGNITION RESULTS

	#Line	#Pattern	#Inst	#Avg.Calc	#MAX
IDCT	215	5	10	1.02	64
SYNFILT	1051	3	7	0.94	24
BH	301	6	14	1.33	37
BLKSORT	289	3	6	0.82	15
HEAP	217	11	23	1.49	10
LEXTREE	696	6	12	1.31	24

C. *Resource Reduction Results*

This section shows the pattern-based FPGA resource reduction results on the test cases mentioned before. Our work is compared to a traditional behavioral synthesis flow [6, 7], and the DFG pattern-based synthesis result in [4].

Table II shows the QoR of our proposed pattern-based synthesis algorithm compared to the other two approaches. In Table II, the second, third and fifth columns are the synthesis results for the number of registers used without pattern-based technique [6, 7], with a DFG pattern-based technique [4], and with CDFG pattern-based technique, respectively. Similarly, columns 7 to 11 list the amount and comparison of logic elements usage.

Overall, our pattern-based synthesis flow can achieve a 24% resource reduction on average over the traditional behavioral synthesis flow. For most test cases, CDFG pattern extraction outperforms work in [4], which can not efficiently deal with sharing at the basic block level. The performance improvement is especially substantial in test cases BLKSORT, where similar patterns are distributed in separate basic blocks instead of the same one. The average runtime overhead is less than one minute.

We also use a heuristic method to balance latency trade-off. If the latency increases beyond a user-given limit with the current pattern selection result, we will recursively "relax" the area reduction by partially recovering shared resources until the latency falls into the acceptable region. However, in our experiments, latency doesn't show drastic changes after our pattern algorithm is applied. There is about 9% latency overhead and 3.5% clock period increase on average.

VI. CONCLUSIONS

In this paper we present a generalized control-flow aware pattern recognition algorithm which can efficiently extract patterns from behavioral specifications. To our knowledge, this is the first published work that can identify approximate patterns in control data flow graphs. Furthermore, the pattern recognition framework is evaluated in resource reduction problem and shows a 24% improvement on average in our experiments.

VII. ACKNOWLEDGEMENT

This research is partially supported by MARCO Gigascale Systems Research Center (GSRC), and the Center for Domain-Specific Computing (CDSC) funded by the NSF Expedition in Computing Award CCF-0926127.

VIII. REFERENCES

[1] B. T. Messmer and H. Bunke. A new algorithm for error-tolerant subgraph isomorphism detection. *IEEE Trans. Pattern Anal. Mach. Intell.*, 20(5): pages 493–504, 1998.

[2] C. Douligeris, G. Feng: Using hopfield networks to solve assignment problem and n-Queen problem: an application of guided trial and error technique. In *SETN '02*: *Proceedings of the Second Hellenic Conference on AI*, pages 325-336, 2002

[3] GMT toolkit. http://www.cs.sunysb.edu/algorithm/implement/gmt /implement.shtml.

[4] J. Cong, W. Jiang: Pattern-based behavior synthesis for FPGA resource reduction. In *FPGA'08*: pages 107-116, 2008.

[5] J. Cong, Y. Fan, G. Han, W. Jiang, and Z. Zhang. Platform-based behavior-level and system-level synthesis. *In Proceedings of IEEE SOCC, 2006.*

[6] J. Cong, Y. Fan, and W. Jiang. Platform-based resource binding using a distributed register-file micro-architecture. In *ICCAD '06: Proceedings of the 2006 IEEE/ACM International Conference on Computer-aided Design,* pages 709–715, 2006.

[7] J. Cong and Z. Zhang. An efficient and versatile scheduling algorithm based on SDC formulation. In *Proceedings of Design Automation Conference*, July 2006.

[8] M. R. Corazao, M. A. Khalaf, L. M. Guerra, M. Potkonjak, and J. M. Rabaey. Performance optimization using template mapping for datapath-intensive high-level synthesis. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 15(8), pages 877–888, 1996.

[9] P. Bonzini and L. Pozzi. Polynomial-time subgraph enumeration for automated instruction set extension. In *DATE '07: Proceedings of the conference on Design, automation and test in Europe*, pages 1331–1336, 2007.

[10] P. Brisk, A. Kaplan, R. Kastner, and M. Sarrafzadeh. Instruction generation and regularity extraction for reconfigurable processors. In *Proceedings of CASES*, 2002.

[11] P. Yu and T. Mitra. Scalable custom instructions identification for instruction-set extensible processors. In *CASES '04: Proceedings of the 2004 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, pages 69–78, 2004.

[12] R. Yang, P. Kalnis, and A. K. H. Tung. Similarity evaluation on tree-structured data. In *SIGMOD '05: Proceedings of the 2005 ACM SIGMOD International Conference on Management of Data*, pages 754–765, 2005.

[13] T. Ly, D. Knapp, R. Miller, and D. MacMillen. Scheduling using behavioral templates. In *DAC '95: Proceedings of the 32nd ACM/IEEE conference on Design automation*, pages 101–106, 1995.

[14] U. Feige. A threshold of ln n for approximating set cover. *Journal of the ACM*, 45: pages 634–652, 1998

[15] Xilinx website. http://www.xilinx.com

[16] X. Li, M. F. M. Stallmann, F. Brglez. Effective bounding techniques for solving unate and binate covering problems. In *DAC '05: Proceedings of the 42nd ACM/IEEE conference on Design automation*, pages 385–390, 2005.

[17] W. Chen, B. Li, R. Gupta. Code Compaction of Matching Single-Entry Multiple-Exit Regions. In *SAS'03: Proceedings of the 10th Annual International Static Analysis Symposium,* pages 401–417, 2003.

[18] R. M. Karp. Reducibility Among Combinatorial Problems. In Complexity of Computer Computations, Plenum Press, 1972.

TABLE II. RESOURCE REDUCTION ON ALL TEST CASES

	FF_{nP}	**FF_{DFG}**	**CMP**	**FF_{CDFG}**	**CMP**	**LE_{nP}**	**LE_{DFG}**	**CMP**	**LE_{CDFG}**	**CMP**
ICDT	1514	1601	5.75%	1193	-21.20%	5071	3998	-21.15%	2870	-43.40%
SYNFILT	476	394	-17.22%	325	-31.72%	1578	1193	-24.40%	1070	-32.19%
BLKSORT	295	237	-6.10%	235	-20.34%	1565	1455	-7.03%	1147	-26.71%
BH	850	701	-17.52%	640	-24.71%	3288	2692	-18.13%	2659	-19.13%
HEAP	1023	945	-7.62%	934	-8.61%	6743	6291	-6.70%	5995	-11.09%
LEXTREE	824	714	-13.35%	695	-15.66%	4211	3740	-11.18%	3543	-15.86%
average			-9.34%		-20.37%			-14.76%		-24.73%