Engineering Change: Methodology and Applications to Behavioral and System Synthesis

Darko Kirovski and Miodrag Potkonjak
Computer Science Department, University of California, Los Angeles

Abstract

Due to the unavoidable need for system debugging, performance tuning, and adaptation to new standards, the engineering change (EC) methodology has emerged as one of the crucial components in synthesis of systems-on-chip. We introduce a novel design methodology which facilitates design-for-EC and post-processing to enable EC with minimal perturbation. Initially, as a synthesis pre-processing step, the original design specification is augmented with additional design constraints which ensure flexibility for future correction. Upon alteration of the initial design, a novel post-processing technique achieves the desired functionality with a near-minimal perturbation of the initially optimized design. The key contribution we introduce is a constraint manipulation technique which enables reduction of an arbitrary EC problem into its corresponding classical synthesis problem. As a result, in both pre- and post-processing for EC, classical synthesis algorithms can be used to enable flexibility and perform the correction process. We demonstrate the developed EC methodology on a set of behavioral and system synthesis tasks.

1 Introduction

Due to the increasing complexities of modern systems-on-chip and more segmented design flows [Sha95], **engineering change (EC)** has recently emerged as the key enabling technology for shortening the time-to-market. The applicability of EC ranges from system debugging to performance tuning [Buc97] and adaptation to new standards. The fundamental goal for any set of EC tools is to provide the designer with the ability to easily perform functional or timing changes on the design, while minimally altering its specification throughout all levels of abstraction. In the case of RTL or logic network descriptions, a small change in the specification may result in significant perturbations of the underlying optimized structures (e.g. layout) [Fan97].

There are two fundamental approaches to EC: design-for-EC, where a certain amount of logic or programmable interconnects with no effect on the functionality and timing constraints is augmented into the design before compilation; and post-processing, where, knowing the correct functionality of the design, the optimized design is minimally altered such that the error is corrected. While the goal of the first technique is to anticipate which hardware might be useful in the case of an alteration, the second one has a difficult task of using a limited amount of resources to update the optimized design with minimal hassle.

We introduce a new design methodology which facilitates design-for-EC and post-processing to enable EC with near-minimal perturbation. Initially, as a synthesis pre-processing step, the original design specification is augmented with additional design constraints that ensure flexibility for future alteration. After the optimization algorithm is applied to the modified input, the added constraints impose a set of additional functionalities that the design can also perform. Upon diagnosis of an alteration in the initial design, a novel post-processing technique, which also facilitates constraint manipulation, achieves the desired functionality with a near-minimal perturbation of the optimized design. The key contribution introduced in this work, is a generalized constraint manipulation technique which enables reduction of an arbitrary EC problem into its corresponding classical synthesis problem. Consequently, in both design-for-EC and post-processing, classical synthesis algorithms can be used to address the optimization requirements for both design-for-EC and the correction process. That is in opposition to the currently adopted research model for EC problems which seeks for new synthesis solutions. We demonstrate the developed EC methodology on a set of behavioral and system synthesis tasks. It is important to stress that the developed EC techniques can be applied to synthesis problems at all levels of design abstraction.

1.1 Motivational Example

We demonstrate how constraint manipulation can be used to enable design flexibility for EC as well as aid in performing the EC process solely on the updated portion of the design using an off-the-shelf synthesis tool. To present the design-for-EC paradigm, we use an example CDFG shown in Figure 2(a). The CDFG has been allocated two different hardware setups: one with 3 adders and 1 subtracter; and the other one with 2 adders and 2 subtracters. Both setups satisfy the requirement of executing all operations of the CDFG within 5 control steps. Scheduling solutions for both setups are presented in Figures 2(b) and 2(c). Assuming an error model, where an addition is mistaken for subtraction or vice versa, the two allocation solutions present different resilience to errors. The solution in Figure 2(b) cannot be changed to support any error where a subtraction is mistaken for addition nor when subtraction v is an addition in the corrected spec. Allocation presented in Figure 2(c) can sustain any single operation error, as well as majority of double errors. Only when both subtractions u and v are corrected to additions, there does not exist any scheduling solution.

The synthesis solution can be optimized during synthesis for near-minimal hassle for EC. Consider the two scheduling solutions presented in Figure 2(d). Both of them correspond to the allocation in Figure 2(c). If addition v is mistaken for subtraction, then in the left scheduling solution there are three operations d, g, and v that have to be rescheduled. However, the scheduling solution on the right requires only operations g and v to be rescheduled. Therefore, in this case, the goal of the design-for-EC process is to ensure that one addition unit is idle at control step 4.

The advantage of using constraint manipulation in per-

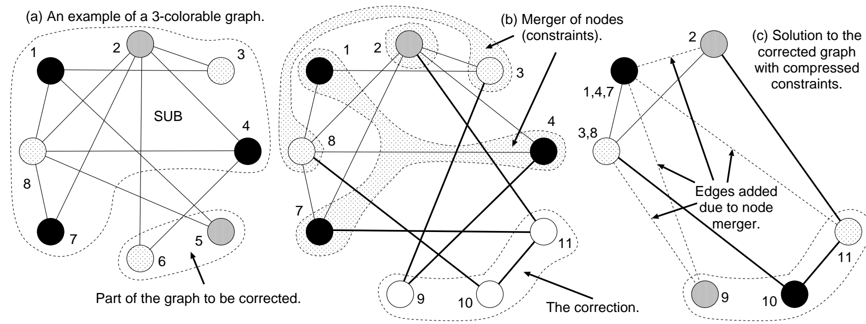

Figure 1: EC: performing graph coloring of the corrected specification only on the updated subgraph.

forming EC is demonstrated on coloring graphs. This task corresponds to many resource allocation problems. An example graph is presented in Figure 1(a). Suppose the designer wants to replace nodes 5 and 6 with nodes 9, 10, and 11, while preserving the coloring of the remaining set of nodes SUB. Note that applying an off-the-shelf coloring algorithm to the corrected specification is not guaranteed to retrieve such solution. Instead of developing a new algorithm for this problem, we manipulate the constraints of SUB in such a way that all nodes in SUB colored with one color are merged into one node. This node inherits the edges of all included nodes. For example, as illustrated in Figures 1(b-c), nodes 4 and 7 are merged with node 1. The resulting graph in Figure 1(c) can be colored with a traditional graph coloring routine, resulting in a correct global coloring of the updated specification where nodes in SUB are colored as in the initial coloring.

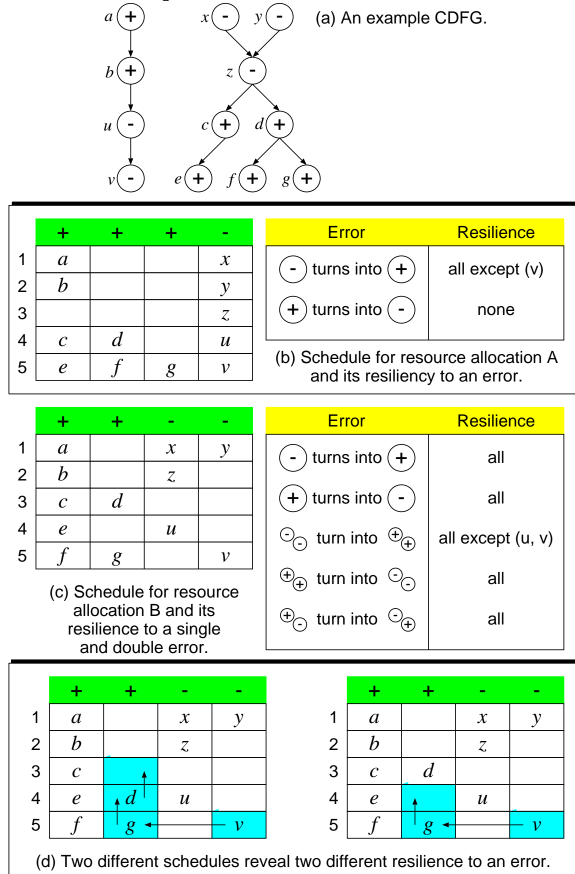

Figure 2: Design-for-EC: two resource allocation and scheduling solutions with different resilience to errors.

2 Related Work

One of the first systems that facilitates EC has been developed to interact with a formal verifier in order to construct a tight error detect-correct engine [Mad89]. A number of developed techniques for EC synthesize rectifying logic networks that, by using and altering the input and output to the existing network, implement the desired functionality [Wat91, Kha96]. Techniques for minimal alteration of an existing logic network focus on developing estimation-based iterative search techniques for minimal logic resynthesis [Swa97] or on reusing gates from the initial implementation and restricting synthesis only to the modified portions [Bra94]. Alternative wires, interconnects that can replace a target wire without changing circuit's functionality, have been shown to aid postlayout logic restructuring [Cha97]. Fang et al. have developed an RT-level EC method which establishes data relationships between design stages and localizes the circuit affected by the change. Their Quick_ECO system has been developed to support on-line debugging of FPGA-based logic emulators [Fan97]. Buch et al. have shown that a common EC technique such as rewiring can be used for power optimization of logic networks [Buc97].

3 Preliminaries

We have selected as a computational model the synchronous data flow (SDF) model [Lee87]. The SDF is a special case of data flow in which the number of data samples produced or consumed by each node on each invocation is specified a priori. Nodes can be scheduled statically at compile time onto programmable processors. We restrict our attention to homogeneous SDF (HSDF), where each node consumes and produces exactly one sample on every execution. The HSDF model is well suited for specification of single task computations in numerous application domains such as communications and multimedia. The syntax of a targeted computation is defined as a hierarchical control-data flow graph (CDFG) [Rab91]. The CDFG represents the computation as a flow graph, with nodes, data edges, and control edges. The semantics underlying the syntax of the CDFG format is that of the SDF model. The HSDF model was mainly selected because of availability of synthesis tools. All developed EC techniques can be applied successfully to other computation models such as the discrete event, communicating FSMs, synchronous/reactive, dataflow process network, and Petri net model [Edw97].

3.1 Targeted Behavioral Synthesis Tasks

Behavioral synthesis transforms a given behavioral specification into a RTL description that can implement a given behavior. An overview of existing synthesis techniques can be found in [DeM94]. For the sake of brevity, we demonstrate the developed EC methodology only for two synthesis tasks: operation scheduling and resource allocation and assignment.

Allocation determines the type and quantity of resources such as storage units, functional units, and interconnect units used in a data path. Assignment is a process of binding each operation to a functional unit, each variable to a storage unit, and each data transportation to an interconnect unit. Optimization goals may vary for various allocation problems. Many register allocation algorithms for CDFGs that contain no loops, focus on either unconditional register sharing [Pau89] or conditional register sharing [Kur87]. For CDFGs with loops, Stok and van den Born [Sto89] proposed a method to break the loops at their boundaries such that variables whose lifetimes cross a loop boundary are split and treated as two separate variables.

Scheduling is a process of partitioning a set of operations in a CDFG into groups of operations such that operations in the same group can be executed concurrently in one control step, while taking into consideration possible trade-offs between total execution time and hardware cost. In the scheduling step, the total number of control steps needed to execute all operations in the CDFG, the minimum number of functional modules, and the lifetimes of variables are determined. There are two basic approaches to scheduling: heuristics [Pau89, Lak98] and integer linear programming [Hwa91].

4 The New EC Methodology

The complexity of modern application-specific systems has resulted in design flows which consist of a number of stages. The two most widely accepted design flows are the golden model and the waterfall model. The golden model is a copy of the design specification at some level of abstraction (usually RTL) at which most of the changes are performed. The underlying concept behind the waterfall design process is a progression through various levels of abstraction with the intent of fully characterizing each level before moving to the next level [Sha95]. As the complexities of behavioral specifications increase, both design flows are becoming more vulnerable to the EC process due to the demand for updating designs throughout many stages.

Flexibility for EC is achieved in a synthesis pre-processing step as shown in Figure 3. The initial behavioral design description BD is augmented with additional design constraints (BD_a). The additional constraints reflect the demand for flexibility. For example, for register allocation, i.e. graph coloring, in order to impose that two variables which may be stored in the same register are assigned to different ones, an edge has to be added between these variables in a preprocessing step to graph coloring. The application of the optimization algorithm to BD_a provides a solution $OptD_a$ that can satisfy both the original and EC-targeted constraints. The additional design constraints can be focused towards a particular type of an error or augmented to provide a guaranteed flexibility for EC after an arbitrary error is diagnosed. The trade-off of having significant design flexibility with respect to a small hardware overhead can be tuned according to the designer's needs.

The error correction post-processing is performed on the augmented design specification BD_a with a desire to alter as few as possible design components and create an optimized design with a given functionality $cOptD_a$. The error correction process is conducted iteratively in a loop with three steps. In the first step, the correction process is restricted to a partition $OBD_a \in BD_a$. OBD_a contains a set of corrections and its closest neighborhood. The optimization process is applied only to this portion of the design, while the optimization solution for the remainder of the graph is left intact. In the second step, the constraints of the remainder of the design $rBD_a = BD_a - OBD_a$ are manipulated. Although the manipulated part of the design, $rcBD_a$, presents a problem of smaller cardinalities, its constraints have the same impact on OBD_a. The constraint manipulation algorithm is heavily dependent upon the actual optimization problem. Details of several such algorithms are presented in the Section 5. In the last step, the off-the-shelf optimization algorithm is applied to the merger of parts $MBD_a = rcBD_a \cup OBD_a$. Portion of the solution to this problem which corresponds to OBD_a is then replaced in the initial optimized solution $OptD_a$ resulting in a corrected optimized solution $cOptD_a$. The increased flexibility for EC on the initial design specification BD_a enables more efficient search for the update that satisfies the correction. The described loop is repeated in a search for the smallest subdomain OBD_a of the original specification where the error correction is performed.

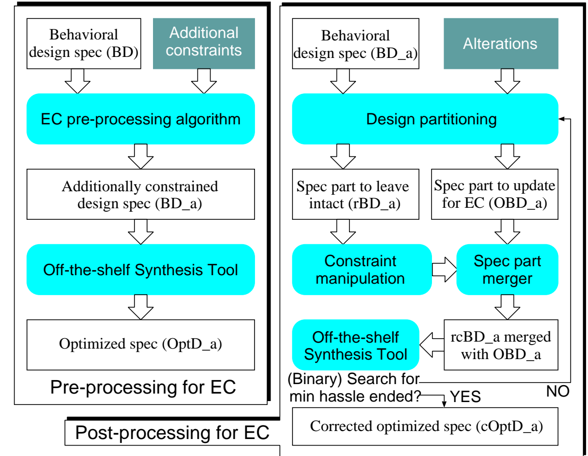

Figure 3: The design flows for design-for-EC and post-processing for EC.

5 The EC Algorithms for Behavioral Synthesis

We applied the proposed EC methodology to two behavioral synthesis tasks: operations scheduling and register allocation. For each of these tasks, we have defined the corresponding design-for-EC and post-processing algorithms for EC, outlined effective algorithms for constraint manipulation, and demonstrated the approach using a second order Gray-Markel ladder filter as an explanatory example.

5.1 Register Allocation and Binding

Values generated in one control step and used in a later step must be stored in a register during the intermediate control step transitions. The *lifetime* of a variable spans between the time it is generated and its last use. Two variables whose lifetimes do not overlap can be stored in the same register. Register assignment is modeled as coloring of a interval graph of a CDFG, where an interval graph [DeM94] is constructed by creating a node for each variable in the CDFG and an edge if the lifetimes of adjacent nodes (variables) overlap. The GRAPH K-COLORABILITY problem is solvable in polynomial time for $K = 2$, but remains NP-complete for all fixed $K \geq 3$ [Gar79].

During design-for-EC, the initial interval graph is augmented with edges that enable flexibility for recoloring. The edges are augmented in correspondence with two types of errors that may occur. The first type of errors are the ones where a variable V (with lifetime $[C_V^S, C_V^E]$) is modified to be used as an operand in an operation O_i that is executed out of the lifetime of variable $V, (C_{O_i} > C_V^E)$. Such error is

modeled by adding edges (type-I) to the interval graph. The procedure that adds edges of type-I is presented in Figure 4. The goal of this procedure is heuristically defined and targets expansion of variables with short lifetimes. Given M, the maximal number of alive variables at any control step C_i ($M = max(AliveVars(C_i), i = 1, \ldots, |C|)$), the procedure expands the lifetime of a variable if this expansion does not increase the number of alive variables at any control step over $M - 1$. In addition, at each control step C_i, only $M - 1 - AliveVars(C_i)$ variables with the shortest lifetimes can be expanded for a single step. Figure 5 shows how the lifetimes of variables $A1, C1, A5, A3, C2$, and $C3$ (bold edges in the CDFG and interval graph) are expanded. Such register assignment can be used to resolve a number of corrections with minimal update as shown in Figure 9.

$M = max(AliveVars(C_i), i = 1, \ldots, |C|)$
Repeat
  For each control step C_i
    Subset of variables $W = \{V_i, C^E_{V_i} = C_{i-1}\}$.
    Select subset $W_k \in W$ of $K < M - 1 - AliveVars(C_i)$
    variables with shortest lifetimes.
    For each $V_i \in W_k$
      $C^E_{V_i} = C_i$.
until W_k is not empty

Figure 4: Procedure used to embed edges of type-I into an interval graph.

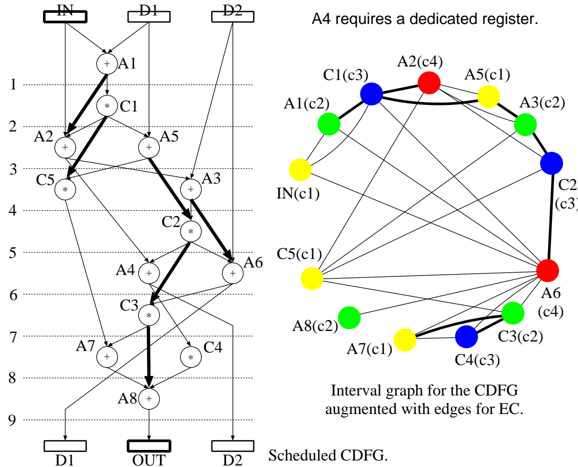

Figure 5: An example of addition of type-I constraints to the graph coloring problem.

The second type of errors are the ones where entire operations (variables) are added to the spec. If such an operation is added at the part of the interval graph where a maximal clique occurs, the EC process would require an addition of a new register. Otherwise, it may happen that this variable can be stored in the existing register file by recoloring the graph. While the first type of a consequence can be trivially solved, the second one requires more attention. To enable effective rescheduling, we identify and/or enable tuples of variables which can switch their registers arbitrarily.

Definition 1. A clique of nodes $V(V_i, i = 1, \ldots, k)$ in a graph for which each node V_j that is a neighbor to $V_k \in V$, V_j is also a neighbor to all other nodes in V, is called **k!-way colorable k-clique**.

The pre-processing, which identifies and enables k!-way colorable k-clique of nodes, consists of two steps. In the first step, as many as possible cliques of large cardinality are heuristically selected and augmented with edges (type-II) such that they become k!-way colorable. In the second step, a heuristic search identifies as many as possible cliques which are already k!-way colorable or require an addition of a small number of edges to become such. The goal of both steps is to maximize the number of nodes which are part of a k!-way colorable k-clique.

Sort $CS = Sort(C)$ according to ascending $AliveVars(C_i)$
$M = max(AliveVars(C_i), i = 1, \ldots, |C|)$
For each control step CS_i and its clique CL
  Find a set of edges $E+$ necessary to be added to nodes
  $V_i \in CL$ such that CL can be arbitrarily colored.
  For each edge $E_i \in E+$
    If addition of E_i to C_i results in $AliveVars(C_i) < M$
    Then break; **Else**
      Add edges $E_i \in E$ to the interval graph
      Remove nodes in CL and adjacent edges from $InterGraph$
For each edge $E_i \in InterGraph$ between nodes A and B
  If $|Nei(A) \notin Nei(B) \cup Nei(B) \notin Nei(A)| > \alpha$
    Add edges $A - Nei(B) \notin Nei(A) \cup B - Nei(A) \notin Nei(B)$
    if none of them results for any C_i that $AliveVars(C_i) < M$

Figure 6: Procedure used to embed edges of type-II into an interval graph.

The procedure that augments type-II edges into a graph coloring instance is outlined using the pseudo-code in Figure 6. In its first phase, the procedure sorts the set of control steps in descending order of the number of alive variables. Then, for each control step C_i, it identifies the variables which constitute a clique of cardinality $AliveVars(C_i)$. The neighborhood of the clique is analyzed whether it has a good potential for embedding edges of type-II. "Good potential" is heuristically defined with a bound on the number of edges adjacent to the nodes in the clique that has to be added in order to enable clique's k!-way colorability. In addition, for each control step, the added edges should not increase $AliveVars(C_i)$ above M. The bound M can be increased if the designers decide to include extra EC registers.

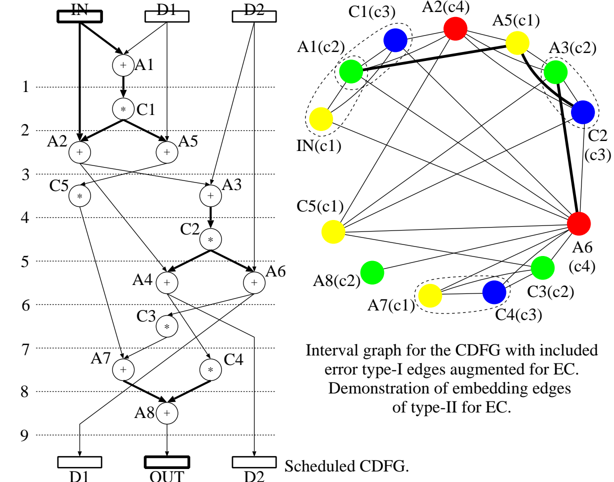

Figure 7: An example of addition of type-II constraints to the graph coloring problem.

The second phase, which enables a large number of 2-way colorable 2-cliques, starts by assigning weights to edges in the interval graph. The weight for an edge between nodes A and B is equal to the sum of number of nodes which are adjacent to one but not both nodes A and B. Next, all edges with weights greater than a predetermined threshold value α are removed from the graph. For each edge $E_{A,B}, W(E_{A,B}) < \alpha$, we add a set of edges $E+$ to nodes A an B such that can be arbitrarily colored. Of course, the addition of each edge $E \in E+$ is bounded by the increase of $AliveVars(C_i)$ beyond M for any control step C_i. An example of addition of such edges is shown in Figure 7. Pairs of nodes $\{IN, A1\}$, $\{A1, C1\}$, $\{A3, C2\}$, and $\{A7, C4\}$ are

enabled for arbitrary colorability by addition of edges drawn in bold in the appropriate interval graph.

Once the error is detected, the goal of the post-processing for EC is to update the smallest part of the design in order to achieve the desired functionality (timing). We have developed an approach which iteratively identifies a locality around the zone that needs to be updated, manipulates the interval graph that is out of the identified locality, and applies off-the-shelf coloring tools on the merger of the manipulated and updated specification. The post-processing procedure is explained in detail in Figure 8.

The first step in the post-processing function performs a simple binary search on the size of the subgraph rBD_a which will be left intact by the EC process. The linear parameter on which we perform the binary search is the maximal distance from any correction on the interval subgraph $OBD_a = BD_a - rBD_a$ to any node in rBD_a. The second step involves manipulation of constraints in the subgraph rBD_a. The result, $rcBD_a$, of this step is obtained in such a way that nodes colored with the same color in the optimized solution $OptD_a$ are merged as formally described in Figure 8. The new merged node inherits all edges adjacent to the parent nodes. Next, an off-the-shelf graph coloring algorithm is applied to the merger of $rcBD_a$ and OBD_a. The colors of nodes in OBD_a from the coloring of the merger of $rcBD_a$ and OBD_a are copied to the optimized solution $OptD_a$ resulting in a corrected coloring $cOptD_a$.

An example of a merger of nodes is shown in Figure 9. The correction of the CDFG is shown in the left part of Figure 9. The new operation X is also added to the interval graph. The intent is to modify the colors of nodes only in the shaded area. Therefore all other nodes are manipulated into four nodes $\{A2\}$, $\{C4, C1\}$, $\{A3, A1, A8\}$, and $\{IN, A5, C5, A7\}$. Obviously, the coloring of this new instance satisfies the new constraints added by X while preserving the colors of nodes outside the shaded area.

Repeat
  $OBD_a = BinarySearch(IntervalGraph, distance[i])$.
  $rBD_a = IntervalGraph - OBD_a$.
  $rcBD_a = Manipulate(rBD_A)$.
For each color C_i
  Create a new node V_I.
  For each node $V_j \in rBD_a$ colored in C_i
    For each edge $E_{j,k}$ adjacent to V_j
      If V_I is not adjacent to V_k
        Add an edge $E_{I,k}$ between V_I and V_k.
    Remove V_j from rBD_a.
$rcBD_a = rBD_a$.
  $cOptD_a = GraphColoring(rcBD_a \cup OBD_a)$
until solution found
Update colors $OptD_a$ of all nodes in OBD_a with their appropriate colors in $cOptD_a$.

Figure 8: Procedure used to perform the error correction process with minimal hassle.

5.2 Operation Scheduling

The design-for-EC procedure is described formally using the pseudo-code in Figure 10. Initially, it adds a K-input single output unit X to the design. Next, a chain O_{chain} of CP successive operations of type X are added to the CDFG in order to force a critical path of length CP. At control steps at which a particular unit U is desired to be idle, operations of type U are attached to the augmented chain O_{chain} as shown in Figure 11. Using this approach, the added operations can either guarantee that a particular unit is idle at some control step or that in some range of control steps a particular unit has at least one idle control step. Obviously, parameter K is equal to the maximum number of idle units at a single control step. The frequency of adding the constraints is calculated as a user-specified percentage α of the fraction of total idle control steps and the number of functional units.

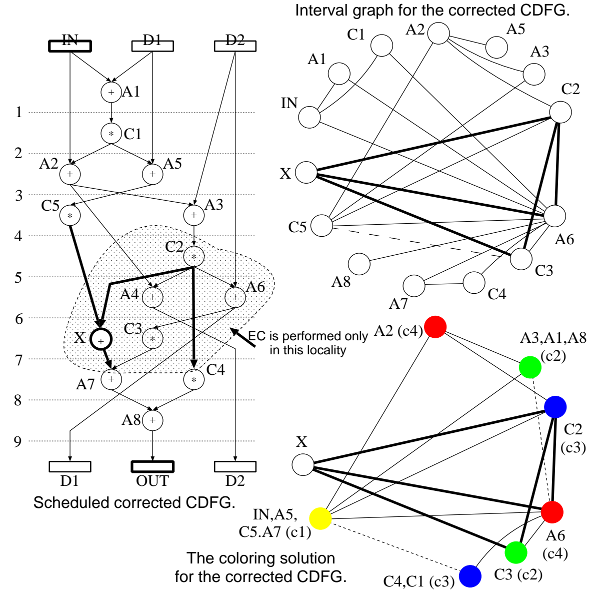

Figure 9: Post-processing for EC: graph bipartitioning, constraint manipulation, and coloring.

An example of such constraint augmentation is presented in Figure 11. The chain of operations of type X as well as augmented additions and multiplications are presented in the shaded area. While the added multiplication enforces that a multiplication unit is idle in control step 6, the two added additions enforce that at least one addition unit is idle during control steps $\{7, 8\}$ and $\{4, 5\}$. The appropriate scheduling that satisfies such constraints is presented in the same Figure.

Compute critical path CP.
Add a costless functional unit X to the allocation list.
Add a chain of CP operations of type X to the design spec.
Count the number of idle steps CS_U in the computation for each set of functional units of type U.
For each set of identical functional units U
  For $i = 1, \ldots, \alpha \cdot CS_U$
    Add an operation of type U which uses as operands variables created by operations of type X executed in control step $ASAP = \frac{i-1}{\alpha \cdot CS_U} \cdot CP$ and generates a variable which is used in control step $ALAP = \frac{i}{\alpha \cdot CS_U} \cdot CP$.

Figure 10: Procedure used to perform the design-for-EC process for operation scheduling solutions.

The error correction process of selected areas in the CDFG is performed using the same sequence of steps as described in Section 4. The only algorithm that is specific to operation scheduling is the constraint manipulation procedure. This procedure has two inputs: a selection of operations O within the CDFG which can be altered; and a frame $\{C_{start}, C_{end}\}$ of control steps in the iteration. The routine creates a new CDFG from all operations O_+ within $\{C_{start}, C_{end}\}$. The As Soon As Possible and As Late As Possible scheduling boundaries for each operation in O_+ are updated to be within the selected frame. Then, the procedure manipulates the constraints of the subgraph $O_+ - O$ in the following way. It adds a costless K-input single output computation unit of type X to the hardware allocation. A chain O_{chain} of $C_{end} - C_{start}$ successive operations of type X is added to the new CDFG in order to enforce a critical path of length CP. Scheduled operations of the subgraph $O_+ - O$ are connected to the

Design					Graph Coloring			Operation Scheduling					
Description	Available control steps	Critical path	Vari-ables	Regi-sters	D-for-EC and EC	Only EC	Complete resynthesis	D-for-EC and EC		Only EC		Complete resynthesis	
								1E	2E	1E	2E	1E	2E
8th CF IIR	$36 \rightarrow 33$	18	35	19	22	23	21	92%	87%	90%	84%	94%	91%
Linear GE Ctlr	$24 \rightarrow 22$	12	48	23	25	28	25	90%	86%	88%	81%	93%	91%
Wavelet Filter	$32 \rightarrow 29$	16	31	20	22	24	21	95%	92%	91%	87%	96%	95%
Modem Filter	$20 \rightarrow 18$	10	33	15	18	19	18	96%	92%	90%	84%	97%	96%
Volterra 2nd ord.	$12 \rightarrow 10$	24	28	15	17	19	17	94%	92%	86%	82%	95%	93%
D/A Converter	$132 \rightarrow 120$	264	354	171	182	187	179	98%	94%	94%	92%	98%	96%
Long Echo Cnclr	$5132 \rightarrow 4500$	2566	1082	1061	1068	1072	1065	97%	93%	94%	91%	99%	95%

Table 1: Overhead of performing modifications on register allocation and operation scheduling instances using design-for-EC and EC, only EC, and complete resynthesis.

chain O_{chain} in a way that any algorithm can retrieve, as trivial, a solution for scheduling equivalent to the one that exists in the initial scheduling. An explanatory example of such manipulation is illustrated in Figure 11. The correction introduces the new addition NEW to the CDFG. All corrections are done within the frame of control steps $\{4, \ldots, 8\}$. Operation $C4$ is within the selected frame but it is not assumed to be rescheduled. Therefore, its variables are fed from the chain of additional operations of type X which enforce it to be scheduled as in the initial scheduling.

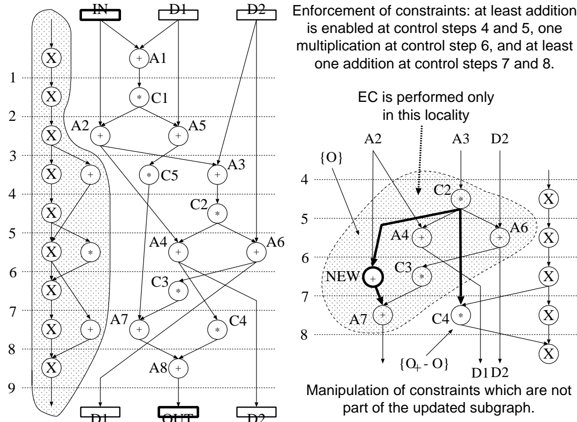

Figure 11: Constraint augmentation and manipulation for pre- and post-processing for EC of operation scheduling.

6 Experimental Results

In order to evaluate the developed EC algorithms, we have conducted experiments on several real-life designs using HYPER as a behavioral compiler [Rab91]. The collected data is presented in Table 1. Column 1 shows the description of the design. Columns 2-5 show the design properties: available control steps, critical path, number of variables, and registers. To test the EC methodology for graph coloring, we have reduced the number of available control step approximately 10% for each design. We have employed three EC techniques: one that performs both the design-for-EC and post-processing step, another that post-processes the design for EC, and finally, we used complete recoloring. The number of registers required to store all variables for modified designs for each of the three EC techniques is presented in columns 6-8. To test the EC algorithms for operation scheduling, we have iteratively generated (1000 iterations for each design) one or two errors in the behavioral description and then applied the aforementioned three EC methods to reschedule the design. Only one type of errors has been induced: changing types of operations (e.g. addition to subtraction). Modifications for each error were searched within a frame of one tenth of the total available control steps. The frames were symmetrically positioned with respect to the error. Pairs of columns 9-10, 10-11, and 12-13 show the percentage of successfully performed modifications for a single and double error for all three methods respectively.

7 Conclusion

We have introduced a novel design methodology which facilitates design-for-EC and post-processing to enable EC with minimal perturbation. Initially, as a synthesis preprocessing step, the original specification is augmented with additional design constraints which ensure flexibility for future correction. Upon alteration of the initial design, a novel post-processing technique achieves the desired functionality with near-minimal perturbation of the optimized design. As a key contribution, we highlighted the constraint manipulation technique which enables reduction of an arbitrary EC problem into its corresponding classical synthesis problem. Consequently, traditional synthesis algorithms can be used to enable flexibility and perform local alterations.

8 References

[Bra94] D. Brand, etal. Incremental synthesis. ICCAD, p.14-18, 1994.

[Buc97] P. Buch, et al. EC for power optimization using global sensitivity and synthesis flexibility. Low Power Electronics and Design, pp.88-91, 1997.

[Cha97] S.-C. Chang et al. Postlayout logic restructuring using alternative wires. TCAD, Vol.16, (no.6), pp.587-96, 1997.

[DeM94] G. De Micheli. Synthesis and optimization of digital circuits. McGraw-Hill, New York, NY, 1994.

[Edw97] S. Edwards et al. Design of embedded systems: formal models, validation, and synthesis. Proceedings of the IEEE, Vol.85, (no.3), pp.366-90, 1997.

[Fan97] W.-J. Fang, et al. A real time RTL engineering change method supporting online debugging for logic emulation applications. DAC, pp.101-6, 1997.

[Gar79] M.R. Garey and D.S. Johnson. Computers and intractability: a guide to the theory of NP-completeness. W.H. Freeman, 1979.

[Hwa91] C.-T. Hwang, J.-H. Lee, and Y.-C. Hsu. A formal approach to the scheduling problem in high level synthesis. TCAD, Vol.10, (no.4), pp.464–475, 1991.

[Kha96] S.P. Khatri, et al. Engineering change in a non-deterministic FSM setting. DAC, pp.451-6, 1996.

[Kur87] F.J. Kurdahi and A.C. Parker. REAL: a program for REgister ALlocation. DAC, pp.210–215, 1987.

[Lak98] G. Lakshminarayana, et al. Incorporating speculative execution into scheduling of control-flow intensive behavioral descriptions. DAC, pp.108-13, 1998.

[Lee87] E.A. Lee and D.G. Messerschmitt. Synchronous dataflow. Proc. of the IEEE, Vol.75, (no.9), pp.1235–45, 1987.

[Mad89] J.C. Madre, et al. Automating the diagnosis and the rectification of design errors with PRIAM. ICCAD, pp.30-3, 1989.

[Pau89] P.G. Paulin, et al. Force-directed scheduling for the behavioral synthesis of ASICs. TCAD, Vol.8, (no.6), pp.661–679, 1989.

[Rab91] J. Rabaey, et al. Fast prototyping of data path intensive architectures. Design & Test, Vol.8, (no.2), pp.40–51, 1991.

[Sha95] G.A. Shaw, et al. Assessing and improving current practice in the design of ASSPs. ICASSP, pp.2707-10, 1995.

[Sto89] L. Stok and R. van den Born. EASY: multiprocessor architecture optimisation. Logic and Arch. Synthesis for Silicon Compilers, pp.313–328, 1989.

[Swa97] G. Swamy, et al. Minimal logic re-synthesis for engineering change. ISCS, Vol.3. pp.1596-9, 1997.

[Wat91] Y. Watanabe and R.K. Brayton. Incremental synthesis for EC. ICCAD, pp.40-3, 1991.