FORM ABSTRACTION IN DGRAPHS

William Bricken
January 1997

A significant issue for the simplification and optimization of digital circuitry is the efficient processing of very large circuits with tens of thousands of gates. The size of EDIF (Electronic Design Interchange Format) files describing the netlist of large circuits is often several megabytes.

Boundary mathematics provides an elegant and easy to process representation of both the behavior and the structure of digital circuitry in the form of distinction graphs (Dgraphs). One strength of this representation is that logical boundaries, as opposed to the traditional logical gates, permit deep algebraic transformations of a circuit. A deep transformation is one that simplifies distant parts of the circuit while keeping behavior invariant. The Losp implementation of Dgraph reduction has been demonstrated to be efficient in both memory and processing time, relative to traditional circuit minimization algorithms. However Dgraphs represent the circuit at a very fine grain of detail. As a consequence, Dgraphs need abstraction transformations which support a larger grain-size for processing.

Since the behavior preserving transformations of boundary mathematics are algebraic, they apply equally well at all grain-sizes. This paper describes several hierarchical abstraction mechanisms which have been developed to provide form abstraction of circuitry. Most of these mechanisms are implemented in versions of the Losp code.

The motivation for Dgraph abstraction is not entirely straightforward, since the goals of circuit minimization are highly dependent on target technology and on the context of available hardware resources. The space of behavior invariant circuit representations is very large, and many abstraction techniques are mutually exclusive. Thus the objectives of the Losp engine interface are to provide the circuit designer with a variety of transformations to choose from, and to construct an initial implementation suggestion based on designer selected parameters. Thus the Losp engine is intended to provide quick and efficient exploration of the design space for circuit synthesis.

In previous years, the Losp implementation has emphasized removal of logical redundancy and manipulation of critical path length. This research focuses on hierarchical abstraction of Dgraphs, and includes the following techniques:

Structural

Coalesce
graphs with identical structure
Grouping
multiple subgraphs with identical structure
Distribute
graphs with identical behavior but multiple reference
Pivot
graphs with identical behavior but different structure

Functional

Abstraction
abstraction of identical functions
Equivalence
identical functionality over various inputs
Symmetry Groups
group theoretic signatures
Partitioning
graphs with separable inputs

Our approaches to the logic synthesis abstractions cited by Sangiovanni-Vincentelli follow:

1. Graph abstraction

All of our abstraction operations are on a graph data structure, so our approach is fundamentally that of a graph transformation system.

2. Functional abstraction

We map logical functional onto graph representation using BM. Thus we do not treat logical abstraction as different from graph abstraction. Every subgraph of a Dgraph can be interpreted as an unambiguous logical function.

3. Synchronous abstraction

By attaching a time index to each Dgraph between registers, we develop a timed Boolean calculus that permits the same approach to equational transformation of sequential circuits as we currently have for combinational circuits. The timing graph is independent of the logic graph.

4. Quantification

We retain the quantifiers of Predicate Calculus (forall, exists) since we use an equational model. All input variables are universally quantified over their domain. Existential variables are abstracted via Skolemization.

5. Cyclic time

Harmonics in the time indices of Dgraph cycles an be used to build a dependency abstraction over timed signals.

6. Power abstraction

Although we have not addressed power directly, we do have tools to analyze the average-case gate utilization of a circuit, and the gate utilization under specific distributions of input values.

7. Similarity abstraction

This memo describes several techniques (coalesce, group symmetry) which implement abstraction over similar circuitry.

Each of these abstractions must be reversible, that is, each abstraction "level" is itself a composition in another abstraction, so it can be analyzed as well as constructed.

The Losp engine is the only research tool which combines the efficiency of boundary representation with the above breadth of abstraction techniques. It should also be noted that all the following techniques generalize in a straightforward manner to timed Boolean functions and thus to sequential circuits, although at a cost of considerable computational effort.

The following assumes that the reader has some familiarity with parens notation, a typographical form of representation of Dgraphs.

STRUCTURAL TECHNIQUES

Structural techniques provide a variety of design choices which influence layout parameters such as circuit area (gate count), critical path timing, and wiring. These abstraction tools are grouping and rearrangement techniques rather than hierarchical techniques.

Coalesce

The Coalesce algorithm implements graph structure sharing. The algebraic rule is

A A = A COALESCE

Coalesce can be applied to any identical graph structure, regardless of depth. As an example, in the following graph

((A B) (C (D (A E))) (F (A G)))

the three occurrences of the subgraph A can all share common circuitry through the Coalesce algorithm.

Coalesce is useful for both database reference sharing and for structure sharing in hardware. The hardware design tradeoff is between a reduced gate

count when sharing is on or a decoupled propagation path when sharing is off. Thus, in the Dgraph

```
A (B C) (B D)
```

when the two occurrences of subgraph B share structure, the signals from (B C) and (B D) to A are coupled by the timing of B, even if the signal from C (or from D) is available. Without structure sharing, the signals from (B C) and from (B D) can propagate to A independently.

Grouping

When more than one subgraph can be coalesced, there is a design tradeoff between path length and connectivity. This choice is manifested by the Clarify rule applied to more than one subgraph:

```
A B = ((A B))                          CLARIFY
```

In the following example,

```
( (A C D E) (B C D E) )
```

the subgraphs C, D, and E can each be coalesced separately. They can also be coalesced together, forming a single grouping:

```
( (A ((C D E))) (B ((C D E))) )
```

The double distinction is necessary in order to combine the three signals, but it introduces a delay of two distinctions along the propagation path. In general, when m subgraphs are grouped and coalesced n times, m+n+1 wires are required. When not grouped, m*n wires are needed while the path is two nodes shorter. In actual hardware, wires represent fan-out and thus power consumption.

Distribute

The Distribute algorithm permits a tradeoff between circuit area and signal propagation time. The transformation is:

```
((A B) (A C)) = A ((B)(C))          DISTRIBUTE
```

This logical transformation differs from the structural transformation of Coalesce in that the duplicate reference is logically redundant and not contextual. Thus the design choice is purely one of the timing: whether or not it is desirable to have the signals from B and C coupled to that of A. Note that when A is referenced twice (as on the left side of the equality), it can still be subject to the structure sharing induced by Coalesce.

Pivot

The Pivot algorithm identifies subgraphs which are behaviorally identical but structurally different. Its form is:

```
( (A B) ((A) C ) ) = (A (B)) ((A)(C))          PIVOT
```

Pivoting identifies a difficult to detect form of behavioral redundancy, and occurs in its most elementary form as the two different structures of Xor:

```
( (A B) ((A)(B)) ) = (A (B)) ((A) B )
```

In general it is desirable to standardize on one form of a pivotable circuit and to eliminate the other form, since this permits the Coalesce option.

FUNCTIONAL TECHNIQUES

Functional techniques provide hierarchical abstractions which simplify the behavioral description and representation of a circuit. They provide a courser grain size for circuit manipulation. All structural abstraction techniques also apply to functionally abstracted units.

Traditional Functional Techniques

The traditional techniques for manipulation of logical functions are all subsumed by our techniques. As well, the boundary techniques, being deep, are much stronger and more general than those in the literature.

Decomposition (same as Coalesce)

Identical subgraphs are identified in a single graph:

```
F = (A B C) (A B D)

      yields

F = (X C) (X D)
X = A B
```

Extraction (same as Coalesce)

Identical subgraphs are identified in more than one graph:

```
F = (A B C)
G = (A B D)

      yields

F = (X C)
G = (X D)
X = A B
```

Factoring (same as Distribute)

A graph is distributed:

```
F = ((A B) (A C))

      yields

F = A ((B)(C))
```

Substitution (same as Functional Abstraction, see below)

A graph is identified within another graph and a labeled substitution is made:

```
G = (A B)

      into

F = (A B) C

      yields

F = G C
```

Elimination (inverse of Functional Abstraction)

Flattening by eliminating a substitution:

```
F = G C
G = (A B)

      yields

F = (A B) C
```

Functional Abstraction

Standardized Dgraphs make pattern matching for functional abstraction fairly easy. For instance, the canonical pattern for logical equality of A and B is

A=B == (A B) ((A)(B)) EQUALITY

Any graph which matches the general pattern

(1 2) ((1)(2))

in which numbers stand in place of arbitrary subgraphs, can be abstracted to a single labeled graph node 1=2 and two annotated pointers 1 and 2 which identify the particular subgraph which are equal. (Note that this is not an assertion of equality, it represents the behavior that the subgraph returns a true or high signal whenever the two subgraph signals do evaluate to be the same, and a low signal otherwise.)

Should another complex graph component abstract to the same two subgraphs, then Coalesce and the other structural rules can be applied to the node labeled A=B. It should be recognized, however, that functional abstraction loses the homogeneity of the Dgraph by introducing different types of nodes. Thus a theory of types must be introduced into the reduction engine. This theory does not differ from the standard theory of distinction reduction when applied to identical types of nodes.

In order for different types of nodes to interact during circuit manipulation, a set of rules (a cross-domain theory) must be introduced into the transformation engine. Other researchers (for example Middlehoek at the University of Twente) have found that this generalization quickly explodes into an unmanageable transformation system. Thus, 1) it is desirable to introduce functional abstraction only when necessary, and 2) it is highly desirable to provide a cross domain theory.

We have implemented functional abstraction for equality (and its negation Xor) and for if-then-else, and have developed cross domain theories for both. Fortunately, such theories are easy to develop between distinctions and an arbitrary function. For example, the rules for combining equality and distinctions are:

(A)=B == A=(B) == (A=B)

The Extract algorithm, which is the workhorse of the Losp engine, can also be generalized to apply over equality nodes:

```
 A  A=B     ==>   A   void=B   ==>   A (B)
(A) A=B     ==>  (A)  (  )=B   ==>  (A) B
 A (A=B)    ==>   A  (void=B)  ==>   A  B
```

These theorems can also be verified by reintroducing the distinction definition of equality and reducing via the distinction calculus.

For generic functional abstraction, a pattern matching language (not yet implemented) could provide arbitrary specification and control. Such a language would allow the user to specify any specific subgraph, for example

(1 (2 (3 4)))

which would then be algorithmically abstracted from the circuit.

Functional Equivalence

In identifying functions for abstraction, it can be useful to select subgraphs which have the same functional behavior but have different input subgraphs. The functional abstraction algorithm can be weakened to include all structurally identical subgraphs regardless of inputs. For example,

(A (B)) is functionally equivalent to (C (D))

but not behaviorally identical. This technique is useful when further transformation on the input subgraphs may establish a relationship between the inputs. It also serves as a basis for a further generalization to symmetry group identification.

Symmetry Groups

Symmetry groups show up quite often in circuit specifications, and are a generalization of many common logical functions. For example, the majority function for three inputs can be expressed as

((1 2) (1 3) (2 3))

The structural symmetry is apparent. Often very large circuits can be abstracted to recursive applied symmetry groups, thus greatly simplifying the description of the circuit's functionality. As well, by identifying mathematical symmetry patterns, all possible structure sharing relationships within a circuit are also identified. The general technique is to search for functional equivalences in a Dgraph representation, and the to further identify permutation groups among the input labels.

Symmetry abstraction can be provided semantically in the description of the circuit's functionality, such as in a 16-bit adder which repeats the symmetry of a 1-bit adder sixteen times. Or it can be recovered from the netlist by pattern matching search. Bit-width plays an central role in inducing massive symmetry repetition in netlists. Most importantly, a circuit can be optimized at selected occurrences of a symmetry group rather than at *every* occurrence.

Thus, for example, in a 16-bit adder, it is possible to apply one adder layout to the top four bits, a different layout to the next four bits, and a third configuration the lower eight bits, should the design benefit from such a strategy. (This is a multi-level carry-skip approach for adders.)

To date we have not implemented a general symmetry abstraction capability, but we have applied specific symmetry abstractions from the bottom up to benchmark circuits with excellent results in reducing the size of the circuit specification.

Partitioning

Again, the Dgraph approach provides efficient algorithmic tools for partitioning algorithms such as k-LUT mapping. The general technique is that labeled subgraphs under different distinction nodes are independent.

Let <> represent subgraph clusters that are partitionable. In the example

```
((A)(B)) (C (D)) (E F)
```

the following partitions are readily identified:

```
AB <> CD <> EF
```

Partitioning by boundary identification is also a deep (i.e. hierarchical) operation. Thus, the graph

```
(A) ((B C (D E F)) G)
```

has the permitted partitions

```
A <> ((BC <> DEF) <> G)
```

and the graph

```
((A B) (C D)) ((A (B)) (C E))
```

permits the partitions

```
AB <> CD   in one subgraph, and
AB <> CE   in the other.
```

Graph partitioning across distinction nodes can be applied to any component subgraph, and can be freely combined with structural transformations during search. These approaches have not yet been automated.