Designing Correct Recursive Circuits Using Semantics-Preserving Tranformations of Nets

Alexander Gamkrelidze **Günter Hotz** **Bin Zhu**

Technical Report A 02/98 December 1998

e-mail: sandro@cs.uni-sb.de, hotz@cs.uni-sb.de, zhu@cs.uni-sb.de
WWW: http://www-hotz.cs.uni-sb.de

Fachbereich Informatik

Fachbereich 14 Informatik
Universität des Saarlandes
Postfach 15 11 50
66041 Saarbrücken
Germany

Designing Correct Recursive Circuits Using Semantics-Preserving Transformations of Nets

Alexander Gamkrelidze, Günter Hotz and Bin Zhu
Fachbereich 14 - Informatik, Universität des Saarlandes
66041 Saarbrücken, Germany
e-mail: sandro@cs.uni-sb.de, hotz@cs.uni-sb.de, zhu@cs.uni-sb.de

Abstract

This paper will present a method of formal synthesis to design correct recursive circuits by using semantics-preserving transformations of nets (SPTNs). Its theoretical base is an algebraic calculus of nets. The calculus of nets is a hardware-specific calculus, and the transformations are circuit transformations themselves. Thus, it is much better adapted to the synthesis domain. The start point of the method is a conceptually simple specification for the required function. This specification can be easily proved to be correct, thereby the perplexed problem of the specification validation can be avoided. The specification is described compactly and graphically by a small kernel of recursive equations, and the synthesis task is simplified to transform these recursive equations in the kernel. Because only semantics-preserving transformations are allowed in synthesis procedures, the synthesis result is not only a hardware implementation, but also a proof of correctness. We will illustrate two ways to transform a basic sorter into a odd-even-merging sorter, one being based on local incremental transformations and the other being based on global partitions. The results show that there are circuits of practical interest, which can derived formally by using this method.

1 Introduction

Formal methods have widely been used to guarantee the correctness of circuits designed. Generally, formal methods can be classified into three classes in terms of the applied time. An ideal method is *pre-synthesis verification.* In fact, it is software verification for synthesis tools. But software verification can be performed now only for very small programs. Another most frequently used method is *post-synthesis verification*, such as *model-checking* and *theorem-proving.* A natural extension is to use formal method during synthesis. This method is called also *formal synthesis.*

Formal synthesis means formally deriving an implementation from a given specification within some logical calculus. In contrast to conventional synthesis, only correctness-preserving transformations are allowed in formal synthesis. Therefore, not only the synthesis result is a hardware implementation, but also its correctness is guaranteed in an implicit manner. There are two classes of transformational derivations, one being based on a hardware-specific calculus and the other being based on a general purpose calculus. In the former, the transformations are circuit transformations themselves, whereas in

the latter, the transformations are logical transformations. Some design systems supporting correctness-preserving transformations have been developed, such as RLEXT (Register Level EXploration) [KnWi92], DDD (Digital Design Derivation) [BoJo93] and RUBY [JoSh90], which are based on hardware-specific calculus, and LAMBDA/DIALOG [FFFH89], VERITAS [HaLD89] and HASH (Higher order logic Applied to Synthesis of Hardware) [EiKu95], which are based on general purpose calculus. [Camp89] indicates how to use behavior-preserving transformations in high-level synthesis. A method of combining functional transformations with deductive theorem-proving techniques can be found in [Busc92]. [SaGG92] showed how machine-checked verification can support an approach to circuit design based on transformations. [KBES96] gives an excellent survey about formal synthesis.

In this paper, we will present a formal synthesis method based on *semantics-preserving transformations of nets* (SPTNs). Its theoretical base is an algebraic calculus named *free D-category*, which was introduced by G. Hotz in 1965 [Hotz65]. This calculus is the first extension of boolean algebra in the direction of representing circuit structures, and has been proved to be universal in the sense where each computable function can be defined by factorizations. It was later generalized to a *bi-category* in connection with the development of the design system CADIC for parametrized hierarchical circuits [BeHK87, BeBH90, HoBZ95]. Based on this bi-category, a hardware description language *2dL* has been developed to describe circuit designs hierarchically and graphically in CADIC [Moli88, HoRe96]. Consider a circuit laid out into a rectangle. A set of its topological layouts is called the *net* of this circuit, if all topological layouts in the set can be transformed each other by a sequence of geometrical deformations. Net is a basic date type of 2dL language. A recursive refinements of initial descriptions of nets will produce a final topological layout. This is the meaning of "hierarchical design" to be used in this paper.

Our target is to provide a method to design correct recursive circuits. The start point of the method is a conceptually simple specification for the required function, such as some school methods for arithmetic circuits. This specification can be easily proved to be correct. Thus, the perplexed problem of the specification validation can be avoided. The specification has a compact and graphical description with a small kernel of recursive equations by using 2dL language. The synthesis task is simplied to transform these recursive equations in the kernel by using semantics-preserving transformations of nets. The calculus of nets is a hardware-specific calculus, and the transformations are circuit transformations themselves. It is much better adapted to the synthesis domain. Differing from general hardware-specific calculus, the core of transformations of nets is small because some transformations are axiomatized and others can be derived from these axiomatized transformations. The transformations can be classified two kinds. The first one is *universal transformation*, which is based on universal algorithms and independent of applied fields. The second one is *synthesis-specific transformation*. Synthesis-specific transformations are dependent on applied fields. Different synthesis problems need different transformation rules, in which some rules can be obtained from exact algorithms, and the others are *ad hoc* transformations based on heuristic strategies. As an example, we will discuss how to use semantics-preserving transformations to derive a correct recursive implementation of odd-even-merging sorter from a basic sorter based on the principle of insertion. Two design approaches will be provided, one being based on local incremental transformations and the other being based on global partitions. In the former, both algorithmic transformations and *ad hoc* transformations are used, and in the latter, only several *ad hoc* transformations are needed.

The remainder of this report is organized as follows: Section 2 is a brief introduction to the calculus of nets and the hardware description language 2dL. In section 3, we define semantics-preserving transformations of nets, and discuss some important transformation rules. Two approaches of designing correct recursive odd-even-merging sorter are shown in section 4. Finally, section 5 gives conclusions and points out further work.

2 Formal Description of Hardware Circuits

In this section, we will present only the necessary underlying knowledge. More basic theory can be found in [Hotz65, Hotz74, Koll86, Moli88, HoRe96, HoZh97].

2.1 Hardware description language 2dL

Hardware description language *2dL* describes circuit designs hierarchically and graphically by abstract floorplans. It is fundamentally two-dimensional. 2dL can be used simultaneously to describe functional properties and topological structures of circuits. The basic elements of the 2dL language are *rectangles* and *polygonal lines* in the plane. On the four boundaries of a rectangle there are the *connectors*. We define the four boundaries of each rectangle as the northern, southern, western, and eastern boundary, and assume that the connectors on northern and southern boundaries are assigned from left side to right one and those on western and eastern boundaries from top side to bottom one. Furthermore, each polygonal line links up exactly two connectors. Two different polygonal lines have no common point.

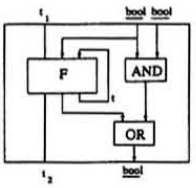


Figure 1:

Net is a basic date type in 2dL, and abstracted as a set of rectangles and polygonal lines. A rectangle is used as the frame of net. Other rectangles represent basic cells or macro cells, and are placed within the frame. These cells are called the *nodes* of the net. Polygonal lines represent interconnections between different cells and between cells and external contacts. The set of all nets is defined as $\mathcal{N}$. An example of net is shown in Figure 1.

Each connector has two attributes, one being the *type attribute*, and the other being the *direction attribute*.

Types name the kinds of connectors. The type of a connector s is denoted by $\tau(s)$. There are two kinds of types, one being the *atomic types*, and the other being the *bundled types*. Examples of atomic types are $\underline{bool}$, $\underline{int}$, and $\underline{real}$, which denote boolean, integer, and real number, respectively. Bundled types are built from atomic types by using the concatenation operation "$\cdot$". For example, $\underline{bool}^3 = \underline{bool} \cdot \underline{bool} \cdot \underline{bool}$ is a bundled boolean type. We will employ the concept of *parametric type*. $\tau(s) = \underline{bool}^n$ $n \in \mathbb{N}$ is an example of parametric type. For convenience, we use $\underline{n}$ to represent $\underline{bool}^n$ as well. In many cases, we will also use *type variable*.

$\omega(s)$ is used to denote the direction attributes of the connector s. We stipulate that each bundled connector has exactly one direction. There are three kinds of directions: input, output and bi-direction (or non-direction), which are denoted by 1, -1, and 0,

respectively. We define $\Omega = \{0, 1, -1\}$. Let s_1 and s_2 be two connectors. If $\omega(s_1) = -\omega(s_2)$ or $\omega(s_1) \cdot \omega(s_2) = 0$, the two connectors are called *compatible to each other* in the directions.

The type and direction of a polygonal line z is determined by the connectors that are incident to z. Obviously, z can be connected to two connectors s_1 and s_2, if and only if s_1 and s_2 are compatible to each other in the directions. If z is connected to bundled connectors, it is called a *bundled polygonal line*. Suppose z is connected to two connectors s_1 and s_2. Then, it holds that $\tau(z) = \tau(s_1) = \tau(s_2)$. If $\tau(s_1)$ and $\tau(s_2)$ are parametric types or type variables, the condition $\tau(z) = \tau(s_1) = \tau(s_2)$ is regarded as a "consistency condition of types".

2.2 Operations of nets

Let T be the set of all types. Suppose $T^* = \{(t_1, t_2, \ldots, t_k) \mid t_i \in T, i = 1, 2, \ldots, k, k \in \mathbb{N}\}$ is the set of all strings generated by T. The concatenation between $(t_1, \cdots, t_k)$ and $(t'_1, \ldots, t'_m) \in T^*$ is represented by $(t_1, \cdots, t_k) \cdot (t'_1, \cdots, t'_m) = (t_1, \cdots, t_k, t'_1, \cdots, t'_m) \in T^*$. ε is used to denote the empty word. Then, $(T^*, \cdot)$ is a free monoid generated by T. We use $\psi(t)$ to denote the range of the type $t \in T$, and consider $\psi(u \cdot v)$ as $\psi(u) \times \psi(v)$ for $u, v \in T^*$. For example, $\psi(\underline{bool}) = \{0, 1\}$, and $\psi(\underline{bool} \cdot \underline{bool}) = \psi(\underline{bool}) \times \psi(\underline{bool})$.

Given $B \in \mathcal{N}$. Let N be a mapping $N : \mathcal{N} \to (T \times \Omega)^*$ such that $N(B) = (N_\tau(B), N_\omega(B))$. $N(B)$ is called the *northern interface* of B. If $N_\tau(B) = \varepsilon$, $N_\omega(B)$ is not existent and $N(B)$ is reduced to $N(B) = (N_\tau(B)) = \varepsilon$. Similarly, we can define *southern, western* and *eastern interfaces* for B, and denote them with $S(B)$, $W(B)$, and $E(B)$, respectively.

Two kinds of composition operations are defined on the set of nets, one being the horizontal operation $\ominus$, and the other being the vertical operation $\oplus$.

Let $F, G \in \mathcal{N}$. The vertical operation $F \oplus G \in \mathcal{N}$ is defined by placing F above G, iff $S(F)$ and $N(G)$ satisfy:

1. $S_\omega(F) * N_\omega(G) \in \{0, -1\}^*$, where $*$ is the element multiplication of $S_\omega(F)$ and $N_\omega(G)$;
2. $S_\tau(F) = N_\tau(G)$.

For the second condition, if both $S_\tau(F)$ and $N_\tau(G)$ have explicit types, they must be equal. If one of them contains parametric type or type variable, the condition can be regarded as a consistency condition.

Similarly, we can define the horizontal operation $F \ominus G \in \mathcal{N}$ by placing F on the left side of G, provided $E(F)$ and $W(G)$ satisfy

1. $E_\omega(F) * W_\omega(G) \in \{0, -1\}^*$, where $*$ is the element multiplication of $E_\omega(F)$ and $W_\omega(G)$;
2. $E_\tau(F) = W_\tau(G)$.

The algebraic structure $(\mathcal{N}, \ominus, \oplus)$ is called a *bi-category*. The operations $\oplus$ and $\ominus$ are associative. They fulfill the law of distribution which represents two possibilities of decomposing the nets (see Figure 2).

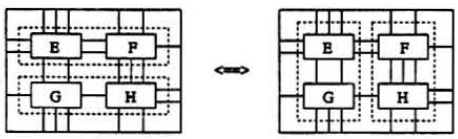


Figure 2:

Therefore, it holds:

Lemma 1 *Given $E, F, G, H \in \mathcal{N}$ with $E \ominus F$, $G \ominus H$, $E \oplus G$ and $F \oplus H$ defined, respectively. Then,*

$(E \ominus F) \oplus (G \ominus H) = (E \oplus G) \ominus (F \oplus H)$

Therefore, all combinational circuits can be formally defined as nets in $\mathcal{N}$:
(1) $\mathcal{C} \subset \mathcal{N}$;
(2) $\forall F, G \in \mathcal{N}$. If $F \oslash G$ is defined, $F \oslash G \in \mathcal{N}$;
(3) $\forall F, G \in \mathcal{N}$. If $F \ominus G$ is defined, $F \ominus G \in \mathcal{N}$.

Another important operation is the refinement of nets. It is a homomorphic mapping in our calculus. We consider the bi-functor $\eta = (\eta_1, \eta_2)$ in $(\mathcal{N}, \ominus, \oplus)$. Suppose $\eta_1 : (T \times \Omega)^* \to (T \times \Omega)^*$ is a monoid homomorphism, for which $\eta_1(t) = t, \forall t \in T$ and $\eta_1(\omega) = \varepsilon \Longrightarrow \omega = \varepsilon \ \forall \omega$. Then, $\eta_2 : \mathcal{N} \to \mathcal{N}$ is a refinement of nets, iff $X(\eta_2(F)) = \eta_1(X(F))$, $\forall F \in \mathcal{N}$, where $X \in \{N, S, W, E\}$.

3 Semantics-Preserving Transformations of Nets

In this section, we will discuss the properties of the algebraic calculus of nets, the classification of semantics-preserving transformations and how to generate correct transformation rules by using the production system of transformations.

3.1 Interpretation of net

In order to be able to treat the general cases, such as bistable circuits and oscillating circuits, in which there are no-monotone information flows, we use *relation* to interpret semantics of nets. That is, the semantics of a net is given by the set of all possible configurations of its external connectors. In these configurations the circuit is stable. The formal definition of the semantics of nets can be found in [HoRe96]. Simple relations can be combined into more complex ones. The relations of all single node within a net compose the relation of the net. This construction defines a mapping of nets. A bi-category is a bi-functor φ, a structure respecting mapping. A semantics-preserving transformation ϕ is a substitution of subnet F within a net by another subnet F' such that (1) $X(F) = X(F')$, $X \in \{N, S, W, E\}$; (2) The semantics relations defined by $\varphi(F)$ and $\varphi(F')$ are the same. Special semantics-preserving transformations can be described by bi-functors, which are mappings from $\mathcal{N}$ to $\mathcal{N}$. We use recursion of such bi-functors to define nets hierarchically.

3.2 Classification of transformations

The calculus of nets is hardware-specific calculus, and the transformations are the circuit transformations themselves. It was proved that, for combinational circuits, there is a complete set of semantics-preserving transformations of nets [Hotz65, Clau71]. [Moli88] further indicated that the complete set of semantics-preserving transformations is not an enumerable relation system in the general case. But differing from general hardware-specific calculus, the calculus of nets has small core of transformations, because some transformations are axiomatized and the other transformations can be derived from these axiomatized transformations by using general algebraic rules.

Transformations can be classified in terms of applied fields or approaches generating transformation rules. According to applied fileds, they can be classified two classes, one being universal transformations and the other being synthesis-specific transformations. Based on approaches generating transformations, they can be classified also two classes,

one being algorithmic transformations based on exact algorithms and the other being *ad hoc* transformations based on heuristic strategies.

3.2.1 Universal transformations

Universal transformations are usually algorithmic transformations, and independent of the applied objects. Some universal transformation most in use are introduced as follows:
(a) ITE Transformation

An important universal transformation is the "If-Then-Else" operation. Given $F \in \mathcal{N}$. Suppose $u = (t_1, \ldots, t_k)$ and $v = (t'_1, \ldots, t'_n) \in T^*$. Let x have boolean type and f be the semantics of F. Then, we have

$$f(u, x) = x \cdot f(u, 1) + \overline{x} \cdot f(u, 0)$$

This is a description of "If-Then-Else" for the semantics f with respect to x. We call it the *ITE transformation.* Figure 3 gives its diagram.

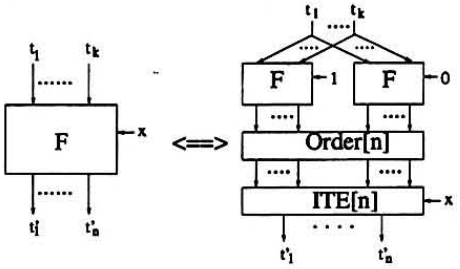


Figure 3:

Especially, if u and v are two strings of boolean type *bool*, the ITE transformation is reduced to the so-called Shannon transformation:

$$f(u, x) = x \wedge f(u, 1) \vee \overline{x} \wedge f(u, 0)$$

In this case, ITE transformation is a generalized decision diagram of BDD [LaBr92].

The net ITE[n] is composed of n multiplexers, and can be constructed recursively by two subcircuits ITE$\lfloor \frac{n}{2} \rfloor$ and ITE$\lceil \frac{n}{2} \rceil$. ITE$_1$ consists of one multiplexer MUX, and serves as base of the induction. The net Order[n] has the function to order the input sequence $\{t_1, t_2, \cdots, t_{\frac{n}{2}}, t_{\frac{n}{2}+1}, \cdots, t_{n-1}, t_n\}$ into the output sequence $\{t_1, t_{\frac{n}{2}+1}, t_2, t_{\frac{n}{2}+2}, \cdots, t_{\frac{n}{2}}, t_n\}$. Their graphical equations are shown in Figure 4.

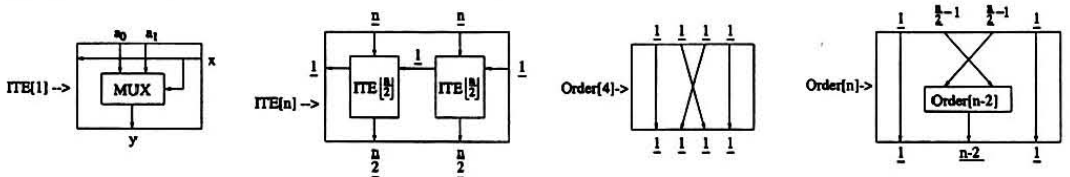


Figure 4:

(b) Replacing:

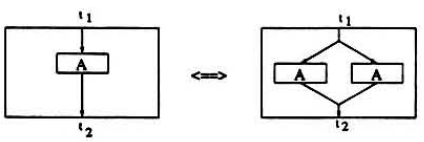


Figure 5:

(c) Merging:

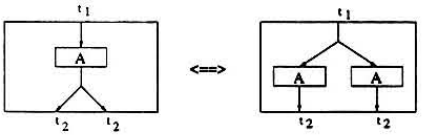


Figure 6:

3.2.2 Synthesis-specific transformations

In synthesis-specific transformations, some are algorithmic transformations and another are *ad hoc* transformations.

ad hoc transformations are widely used in formal synthesis practice. They usually come from designer's experiences and are mainly based on heuristic strategies. Generally, they are individual transformations and lack interrelations. The correctness of *ad hoc* transformations can be proved by manual proof or by mechanical checkers.

To a great extent, algorithmic transformations in synthesis-specific transformations are dependent on synthesis procedures. The correctness of transformations can be guaranteed by algorithms themselves, but their applied ranges are limited within given objects. However, we notice that there are interrelations among these transformations. Some transformation rules can be derived by another known rules. For synthesis problem specially designated, some simple rules can be composed of only several active and passive nets. Based on these simple rules, we can further derive some complex rules. Therefore, we can obtain the required rules by a production system of transformations.

3.3 Production system of transformation rules

If a net $F \in \mathcal{N}$ is used to generate another net, F is called a *producer*. If two nets have same semantics, we call them *equivalent*. A production system of transformation rules is a smallest set of producers, which can be used to derive a required set of equivalent nets. Arbitrary a pair of nets in the set of equivalent nets make up a pair of semantics-preserving transformations of nets.

Now, we take sorting net as example to illustrate how a production system of transformations can be constructed.

We have convention that for an n-element sorting net, elements $x_1, x_2, \cdots, x_n$ enter to top, and output from bottom in order $z_1, z_2, \cdots, z_n$ from left to right. The comparator CMP is a basic active net. It sorts its two input elements x_1 and x_2 in order, i.e., $z_1 = min\{x_1, x_2\}$ and $z_2 = max\{x_1, x_2\}$. For convenience, we can use a horizontal connection between two lines to represent CMP as shown in Figure 7(a). We will alternatively use two representations below. An example of 3-element insertion sorting net represented by the CMP cells and the symbols respectively is given in Figure 7(b). If nets A and B can be derived each other by using semantics-preserving transformation rules of nets, we use sign "$A \Longleftrightarrow B$" to represent it.

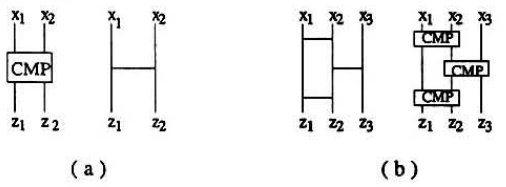


Figure 7:

3.3.1 Producing 3-element sorting nets

A 2-element sorting net consists of only an active cell CMP. In order to obtain all 3-element sorting nets, we construct a production system, which consists of three basic cells: two cross line cell $Cross$, one vertical line cell "|" and the basic comparator CMP (see Figure 8). Where, the cell $Cross$ has function $Cross(a, b) = (b, a)$.

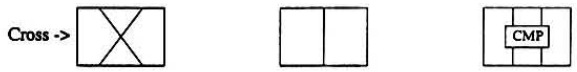


Figure 8:

Based on these basic cell, we can generate all 3-element minimum-comparison networks by using the insertion algorithm of sorting networks as follows:

$Sort1[3] = (CMP \ominus |) \oplus (| \ominus CMP) \oplus (CMP \ominus |)$
$Sort2[3] = (CMP \ominus |) \oplus (Cross \ominus |) \oplus (| \ominus CMP) \oplus (Cross \ominus |) \oplus (| \ominus CMP)$
$Sort3[3] = (| \ominus CMP) \oplus (| \ominus Cross) \oplus (CMP \ominus |) \oplus (| \ominus Cross) \oplus (CMP \ominus |)$
$Sort4[3] = (| \ominus Cross) \oplus (CMP \ominus |) \oplus (| \ominus Cross) \oplus (CMP \ominus |) \oplus (| \ominus CMP)$
$Sort5[3] = (Cross \ominus |) \oplus (| \ominus CMP) \oplus (Cross \ominus |) \oplus (| \ominus CMP) \oplus (CMP \ominus |)$
$Sort6[3] = (| \ominus CMP) \oplus (CMP \ominus |) \oplus (| \ominus CMP)$

Thier schematic equations are shown in Figure 9. We define

$SORT_3 = \{Sort1[3], Sort2[3], Sort3[3], Sort4[3], Sort5[3], Sort6[3]\}$

$SORT_3$ is a set of equivalent nets for 3-elemnet sorters. It is complete, and arbitrary two nets in $SORT_3$ compose a pair of semantics-preserving transformations.

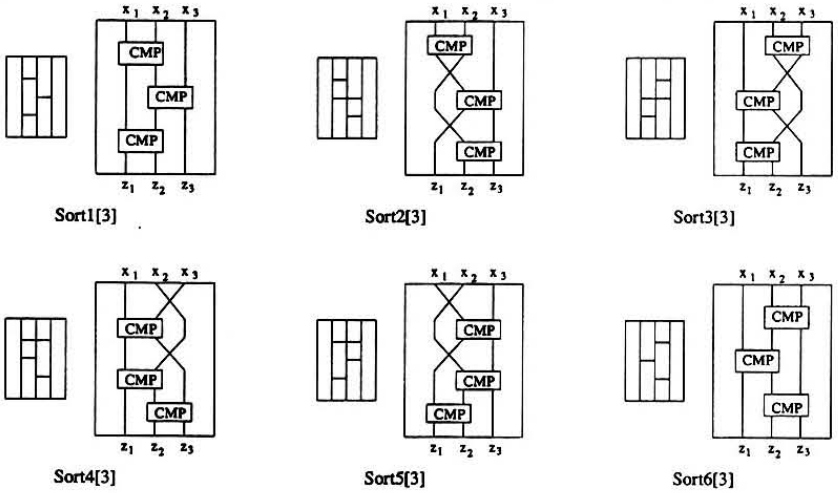
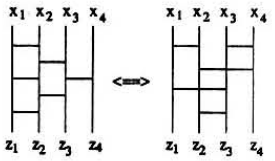


Figure 9:

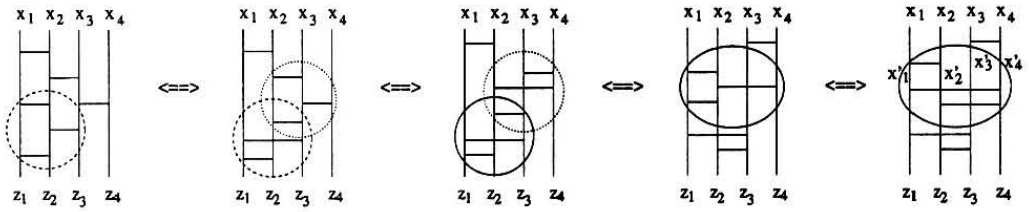
3.3.2 Producing 4-element sorting nets

With the aid of 3-element sorters we can derived some 4-element sorters. Now we prove an useful Lemma.

Lemma 2 *The 4-element odd-even-merging sorter can be derived from 4-element basic sorter by using the class of 3-element equivalent transformations.*



Proof:



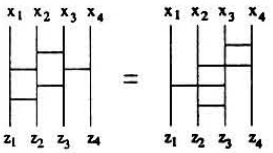
Now, we have $x'_1 \leq x'_2$ and $x'_3 \leq x'_4$.

(1) If $x'_1 \leq x'_4$, the comparator $CMP(x'_1, x'_4)$ can be eliminated because of no exchange between x'_1 and x'_4;

(2) If $x'_1 > x'_4$, it holds that $x'_3 \leq x'_4 < x'_1 \leq x'_2$, which can be implemented just by $CMP(x'_1, x'_3)$ and $CMP(x'_2, x'_4)$. Thus, the comparator $CMP(x'_1, x'_4)$ can be eliminated as well. Q.E.D.

Based on the Lemma 2, we can get the following Corollary:

Corollary 1 *If x_1, x_2 are sorted, then it holds that:*



where z_1, z_2, z_3, z_4 are sorted.

4 Designing Recursive Odd-Even-Merging Sorter: A Case Study

In this section, we will discuss how to design correct recursive odd-even-merging sorter by using semantics-preserving transformations of nets. Two ways will be provided, one being based on local incremental transformations and the other being based on global partitions. The specification is a recursive basic sorter. It is constructed by using the principle of insertion, and obviously correct. Because semantics will be preserved during synthesis, the design result gives not only an implementation of odd-even-merging sorter but also a proof of correctness.

4.1 Basic sorter

A simple way to construct a sorting network is to use the principle of *insertion* [Knut73]. Given n elements $x_1, x_2, \cdots, x_n$. Suppose that the first $n-1$ elements $x_1, x_2, \cdots, x_{n-1}$ have been sorted, the element x_n can be inserted into its proper place by using $n-1$ comparators as shown in Figure 10(a). The Figure 10(b) gives an example of the 6-element basic sorter. The correctness of this design is obvious. We call it *basic sorter.*

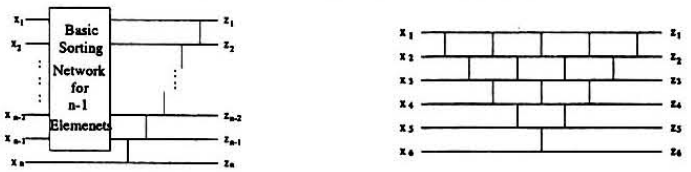
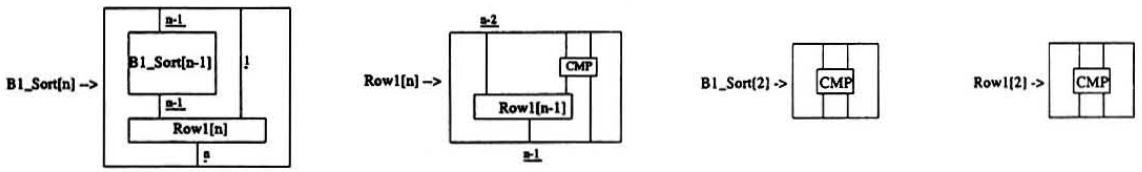


Figure 10:

We define $B1\_Sort[n]$ as the implementation of the basic sorter. $B1\_Sort[n]$ can be designed recursively by using the following equations:

$$
\begin{aligned}
B1\_Sort[n] &= (B1\_Sort[n-2] \ominus |_2) \oplus Row1[n] && (1)\\
B1\_Sort[2] &= CMP && (2)\\
Row1[n] &= (|_{n-2} \ominus CMP) \oplus (Row1[n-1] \ominus |) && (3)\\
Row1[2] &= CMP && (4)
\end{aligned}
$$

Their schematic representations are shown in Figure 11.



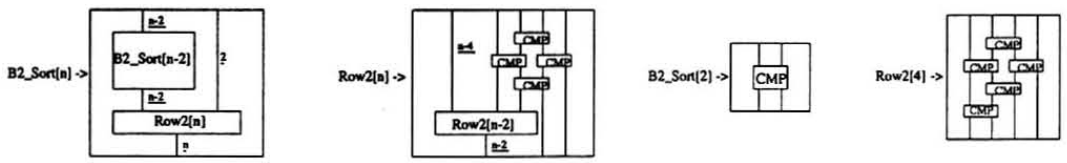
(a) (b) (c) (d)

Figure 11:

In the above design, $B1\_Sort[n]$ is constructed by adding one row of CMP cells in each recursive step. If n is restricted only within even numbers, we can construct the basic sorter by adding two rows of comparators CMP in each recursive step. In this case, the recursive equations are changed into:

$$
\begin{aligned}
B2\_Sort[n] &= (B2\_Sort[n-2] \ominus |_2) \oplus Row2[n] && (5)\\
B2\_Sort[2] &= CMP && (6)\\
Row2[n] &= (|_{n-4} \ominus | \ominus CMP \ominus |) \oplus (|_{n-4} \ominus CMP \ominus CMP) \oplus (|_{n-4} \ominus | \ominus CMP \ominus |)\\
&\quad \oplus (Row2[n-2] \ominus | \ominus |) && (7)\\
Row2[4] &= (| \ominus CMP \ominus |) \oplus (CMP \ominus CMP) \oplus (| \ominus CMP \ominus |) \oplus (CMP \ominus | \ominus |) && (8)
\end{aligned}
$$

Their schematic representations are shown in Figure 12.



(a) (b) (c) (d)

Figure 12:

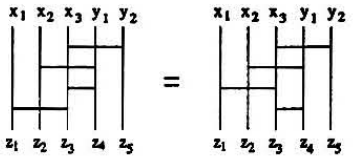
4.2 Incremental design based on local transformations

Incremental design is a synthesis method used widely by circuit design engineers. Differing from the general incremental design, our method will allows to use only semantics-preserving transformations of nets.

4.2.1 *ad hoc* transformations

Aiming at the recursive design of sorter, we need several extra *ad hoc* transformations, which are based on heuristic strategies during design procedure.

Lemma 3 *If x_1, x_2, x_3 and y_1, y_2 are sorted, respectively, then there exists:*



Proof: On the left side, we have:

$z_1 = (x_3 \wedge y_2) \wedge (x_2 \vee y_1) \wedge x_1 = x_1 \wedge y_2$

$z_2 = x_2 \wedge y_1$

$z_3 = ((x_3 \wedge y_2) \wedge (x_2 \vee y_1)) \vee x_1 = (x_2 \wedge y_2) \vee (x_3 \wedge y_1) \vee x_1$

$z_4 = (x_3 \wedge y_2) \vee x_2 \vee y_1$

$z_5 = x_3 \vee y_4$

On the right side there are:

$z_1 = x_1 \wedge x_3 \wedge y_2 = x_1 \wedge y_2$

$z_2 = x_2 \wedge y_1$

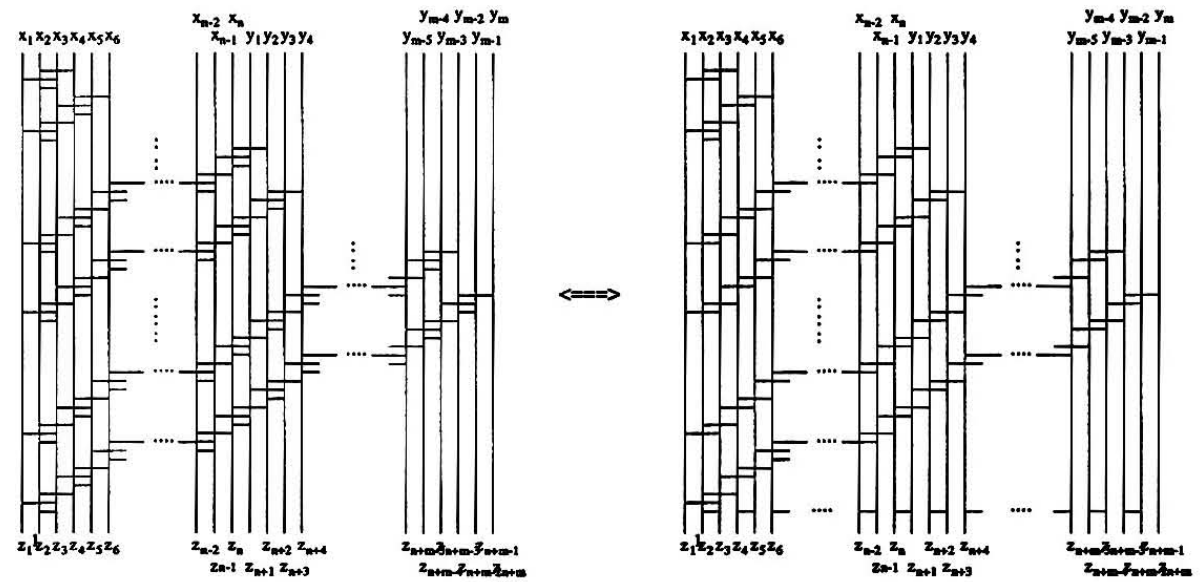
$z_3 = ((x_3 \wedge y_2) \vee x_1) \wedge (x_2 \vee y_1) = (x_3 \wedge y_2 \wedge (x_2 \vee y_1)) \vee (x_1 \wedge (x_2 \vee y_1))$

$= (x_2 \wedge y_2) \vee (x_3 \wedge y_1) \vee x_1$

$z_4 = ((x_3 \wedge y_2) \vee x_1) \vee x_2 \vee y_1$

$z_5 = x_3 \vee y_4$ Q.E.D.

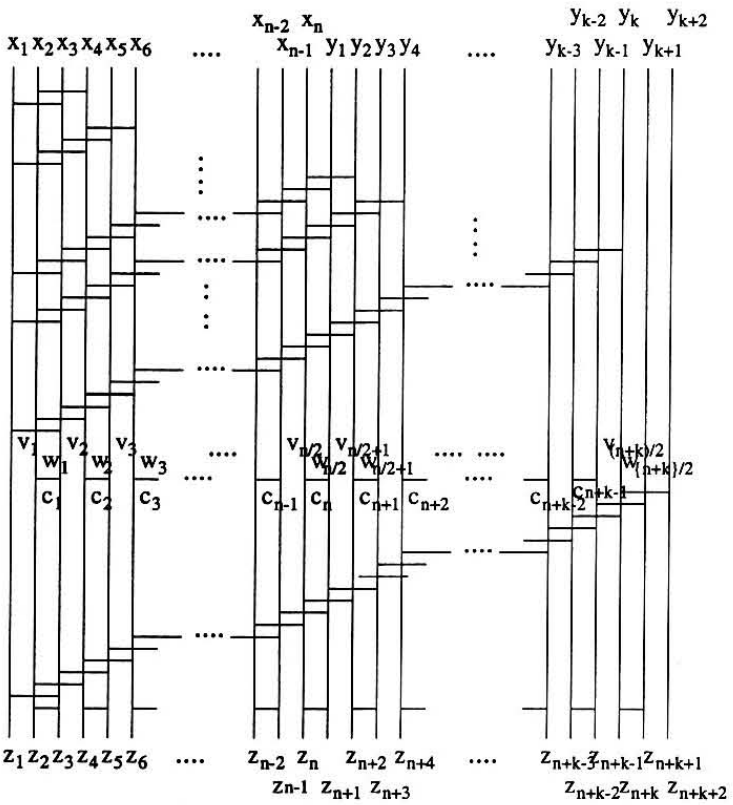
Lemma 4 *Suppose $x_1, x_2, \cdots, x_n$ and $y_1, y_2, \cdots, y_m$ are sorted, respectively $(n, m = 2k, k \in \mathbb{N})$. Then, it holds that:*



Proof: With the aid of induction.

For $\forall n$ $(n = 2i, i \in \mathbb{N})$. When $m = 2$, because $x_1, x_2, \cdots, x_n$ are sorted, the conclusion is true in terms of Lemma 3.

Suppose the conclusion is true when $m = k$. If $m = k + 2$, we have



Where $v_i \leq v_{i+1}$, $w_i \leq w_{i+1}$ and $v_i \leq w_i$, $(i = 1, 2, \cdots, \frac{n+k}{2})$.

(1) If $w_i \leq v_{i+1}$, the conclusion holds obviously.

(2) If $w_i > v_{i+1}$, we call that there is an *inversion* $I_{i,i+1}$. Define the set $I = \{I_{i,i+1} | i = 1, 2, \cdots, \frac{n+k}{2} - 1\}$. Suppose $w_j = y_k$. Because k is an even number, there is no inversion in the subsequence $S_2 = (v_{j+1}, w_{j+1}, v_{j+2}, w_{j+2}, \cdots, v_{\frac{n+k}{2}}, w_{\frac{n+k}{2}})$. Consider the following cases:

(a) If both y_{k+1} and y_{k+2} are larger than $w_{\frac{n+k}{2}}$, then no new inversion will be generated, and the order of elements in the subsequence $S_1 = (v_1, w_1, v_2, w_2, \cdots, v_j, w_j)$ will not be changed. Therefore, $n = k - 1$ comparators $C_1, C_2, \cdots, C_{n+k-1}$ can be eliminated.

(b) If $y_{k+1} < v_{\frac{n+k}{2}}$ and $y_{k+2} > v_{\frac{n+k}{2}}$, since $y_{k+2} > y_{k+1} > y_k$, y_{k+1} can be inserted only in S_2, and maybe generated inversion. But, the order of elements in S_1 is not changed. So, $n = k - 1$ comparators $C_1, C_2, \cdots, C_{n+k-1}$ can be eliminated.

(c) Similarly, we can prove the conclusion when $y_{k+1} < y_{k+2} < w_{\frac{n+k}{2}}$. Q.E.D.

4.2.2 Semantics-preserving design

Our start point is the specification $B2\_Sort[n]$ of the basic sorter. A recursive implementation of odd-even-merging sorter $OE\_Sort[n]$ will be derived from $B2\_Sort[n]$ by using several rules of semantics-preserving transformations. The derivation will be aided by induction.

The principle of odd-even-merging sorter is to use a odd-even-merging procedure to two sorted sequences. Thus, we assume that $OE\_Sort[n]$ consists of two $\frac{n}{2}$-element odd-even-merging sorters $OE\_Sort[\frac{n}{2}]$ and one n-element merger $Merg[n]$. That is,

$$OE\_Sort[n] = (OE\_Sort[\frac{n}{2}] \ominus OE\_Sort[\frac{n}{2}]) \oplus Merg[n] \tag{9}$$

which provides a recursive equation for designing $OE\_Sort[n]$. The corresponding

schematic representation is shown in Figure 13(b). Therefore, our task is just to find recursive equations needed by designing $Merg[n]$.

Step 1: Given a sequence $x_1, x_2, \cdots, x_n$. When $n = 4$, according to Lemma 2 we have $B2\_Sort[4] = OE\_Sort[4]$ (see Figure 13(a)). Thus, the conclusion is true.

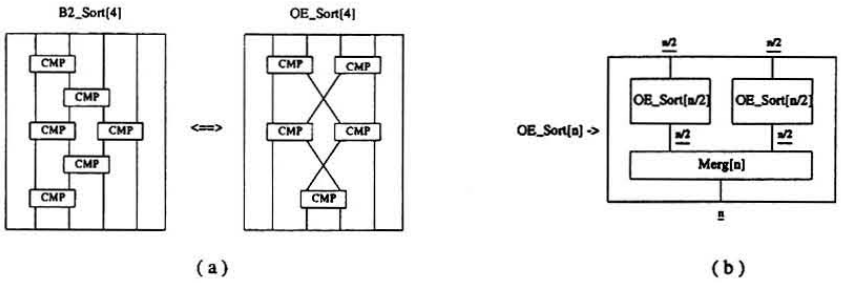


Figure 13:

Step 2: Suppose when $n = \frac{m}{2}$, $(m = 2k, k \in \mathbb{N})$, there exists

$$B2\_Sort[\frac{m}{2}] = OE\_Sort[\frac{m}{2}] \tag{10}$$

Let $n = m$. We rewrite the recursive equations of $B2\_Sort[m]$ as follows (see Equations 5 as well) :

$$B2\_Sort[m] = (B2\_Sort[m-2] \ominus |_2) \oplus Row2[m] \tag{11}$$

Because $B2\_Sort[m]$ sorts an arbitrary input sequence $x_1, x_2, \cdots, x_m$ to a sorted sequence $z_1, z_2, \cdots, z_m$, we can first sort two subsequences $x_1, x_2, \cdots, x_{\frac{m}{2}}$ and $x_{\frac{m}{2}+1}, x_{\frac{m}{2}+2}, \cdots, x_m$ by using two odd-even-merging sorters $OE\_Sort[\frac{m}{2}]$, respectively. Thus, we have the equivalent representation $Be\_Sort[m]$:

$$Be\_Sort[m] = (OE\_Sort[\frac{m}{2}] \ominus OE\_Sort[\frac{m}{2}]) \oplus B2\_Sort[m] \tag{12}$$

Its schematic representation is shown in Figure 14.

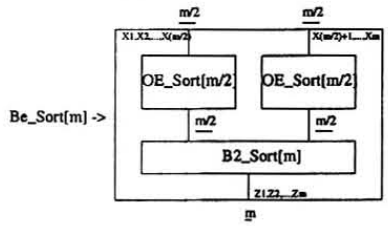


Figure 14:

Step 3: Consider the cell $Row2[m]$ in $B2\_Sort[m]$, whose recursive equation is shown in the equation 7. According to Corollary 1, $Row2[4]$ is equivalent to $Rows[4]$, where

$$Rows[4] = (|\ominus|\ominus CMP) \oplus (|\ominus Cross \ominus|) \oplus (CMP \ominus CMP) \oplus (|\ominus Cross \ominus|) \oplus (|\ominus CMP \ominus|) \tag{13}$$

We substitute the core $Row2[4]$ by $Rows[4]$, and obtain the equivalent recursive equation:

$$\begin{aligned} Rows[m] \;=\; & (|_{m-4} \ominus | \ominus | \ominus CMP) \oplus (|_{m-4} \ominus | \ominus Cross \ominus |) \oplus (|_{m-4} \ominus CMP \ominus CMP) \\ & \oplus (|_{m-4} \ominus | \ominus Cross \ominus |) \oplus (|_{m-4} \ominus | \ominus CMP \ominus |) \oplus (Rows[m-2] \ominus | \ominus |) \end{aligned}$$

Because x_{m-1} and x_m are sorted, the comparator $CMP(x_{m-1}, x_m)$ can be eliminated. The equation is simplized to:

$$\begin{aligned} Rows[m] \;=\; & (|_{m-4} \ominus | \ominus Cross \ominus |) \oplus (|_{m-4} \ominus CMP \ominus CMP) \oplus (|_{m-4} \ominus | \ominus Cross \ominus |) \\ & \oplus (|_{m-4} \ominus | \ominus CMP \ominus |) \oplus (Rows[m-2] \ominus | \ominus |) \end{aligned} \tag{14}$$

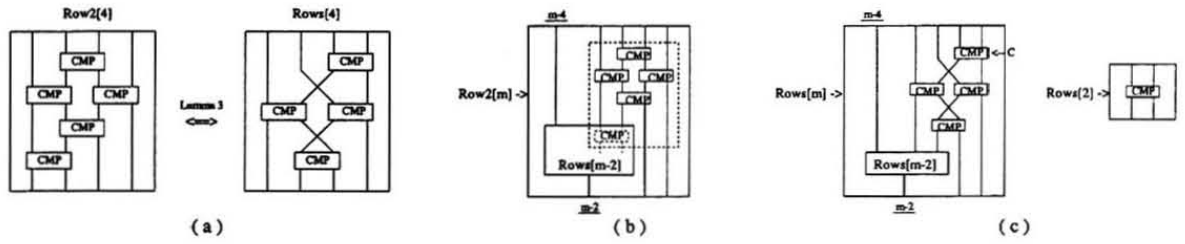


Figure 15:

The corresponding schemantic representations are given in Figure 15.

Step 4: Using Lemma 3, the comparator C in $Rows[m]$ can be moved to the position under $Rows[m-2]$. The new recusive equation is called $Rowt[m]$. We have

$$\begin{aligned} Rowt[m] &= (|_{m-4} \ominus | \ominus Cross \ominus |) \oplus (|_{m-4} \ominus CMP \ominus CMP) \oplus (|_{m-4} \ominus | \ominus Cross \ominus |) \\ &\quad \oplus (Rowt[m-2] \ominus | \ominus |) \oplus (|_{m-3} \ominus CMP \ominus |) \qquad (15) \\ Rowt[2] &= | \ominus | \qquad (16) \end{aligned}$$

Its schematic equations are shown in Figure 16(a)). Move the comparators on the lowst row from $Rowt[m-2]$, and define them together with the comparator C as the new cell $COMP[m-2]$. The rest part of $Rowt[m-2]$ is called $Rowh[m-2]$. Therefore, we obtain:

$$\begin{aligned} Rowt[m] &= Rowh[m] \oplus (| \ominus COMP[m-2] \ominus |) \qquad (17) \\ Rowh[m] &= (|_{m-4} \ominus | \ominus Cross \ominus |) \oplus (|_{m-4} \ominus CMP \ominus CMP) \qquad (18) \\ &\quad \oplus (|_{m-4} \ominus | \ominus Cross \ominus |) \oplus (Rowh[m-2] \ominus | \ominus |) \qquad (19) \\ Rowh[2] &= | \ominus | \qquad (20) \\ COMP[m] &= COMP[m-2] \ominus CMP \qquad (21) \\ COMP[2] &= CMP \qquad (22) \end{aligned}$$

Their schematic equations are shown in Figure 16.

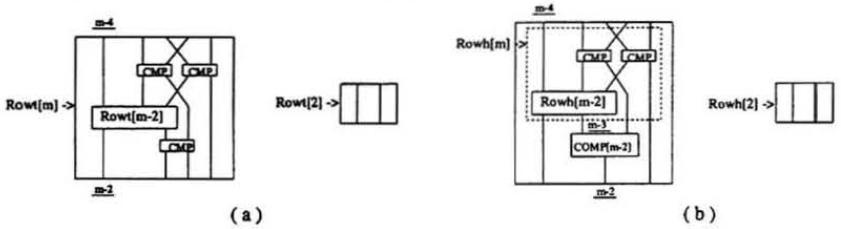


Figure 16:

Step 5: Substituting $Rowt[m]$ for $Row2[m]$ in Equation 11 and defining the new circuit as $Bt\_Sort[m]$, we obtain the recursive equations:

$$Bt\_Sort[m] = (Bt\_Sort[m-2] \ominus |_2) \oplus Rowt[m] \qquad (23)$$

According to Lemma 4, the cells $COMP[m-4], COMP[m-6], \cdots, COMP[2]$ in $Bt\_Sort[m-2]$ can be eliminated. Defining the rest part of $Bt\_Sort[m-2]$ as $Bh\_Sort[m-2]$, we obtain:

$$\begin{aligned} Bt\_Sort[m] &= Bh\_Sort[m] \ominus (| \ominus COMP[m-2] \ominus |) \qquad (24) \\ Bh\_Sort[m] &= (Bh\_Sort[m-2] \ominus |) \oplus Rowh[m] \qquad (25) \end{aligned}$$

The corresponding schematic equations are given in Figure 17.

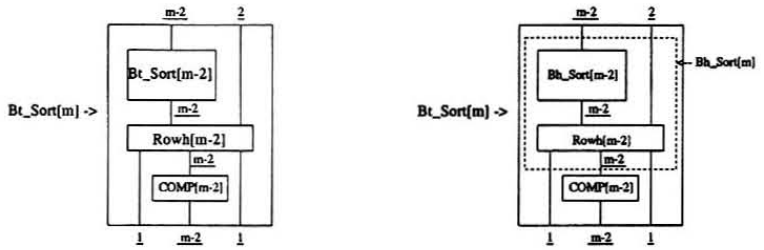


Figure 17:

Step 6: It can be seen that $Bh\_Sort[m]$ consists of two $\frac{m}{2}$-element basic sorts $B2\_Sort[\frac{m}{2}]$, which sort the odd-sequence $v_1, v_3, \cdots, v_{m-1}$ and the even-sequence $v_2, v_4, \cdots, v_m$, respectively. We introduce two wiring cells, one being $OddEven[m]$ and the other being $Shuffle[m]$. The former separates the input sequence $v_1, v_2, \cdots, v_m$ into two subsequences, and place the odd-sequence $v_1, v_3, \cdots, v_{m-1}$ in the left side and the even-sequence $v_2, v_4, \cdots, v_m$ in the right side. The latter is the inversion of the above transformation. With using these cells, we can separate two basic sorters $B2\_Sort[\frac{m}{2}]$, and obtain the recursive equations:

$$Bh\_Sort[m] = OddEven[m] \oplus (B2\_Sort[\frac{m}{2}] \ominus B2\_Sort[\frac{m}{2}]) \oplus Shuffle[m] \quad (26)$$

$$OddEven[m] = (OddEven[\frac{m}{2}] \ominus OddEven[\frac{m}{2}]) \oplus (|_{\frac{m}{4}} \ominus Cross[\frac{m}{4}, \frac{m}{4}] \ominus |_{\frac{m}{4}}) \quad (27)$$

$$OddEven[4] = | \ominus Cross \ominus | \quad (28)$$

$$Shuffle[m] = (|_{\frac{m}{4}} \ominus Cross[\frac{m}{4}, \frac{m}{4}] \ominus |_{\frac{m}{4}}) \oplus (Shuffle[\frac{m}{2}] \ominus Shuffle[\frac{m}{2}]) \quad (29)$$

$$Shuffle[4] = | \ominus Cross \ominus | \quad (30)$$

Their schematic equations are given in Figure 18.

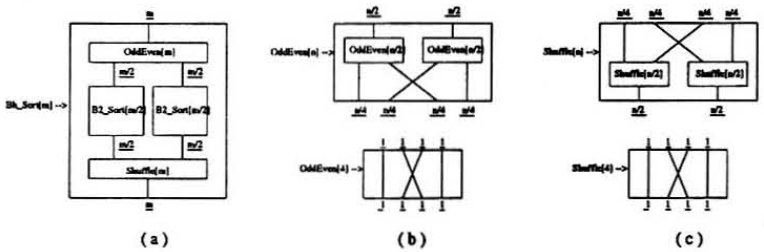


Figure 18:

Step 7: Because

$$B2\_Sort[\frac{m}{2}] = OE\_Sort[\frac{m}{2}]$$
$$= (OE\_Sort[\frac{m}{4}] \ominus OE\_Sort[\frac{m}{4}]) \oplus Merg[\frac{m}{2}] \quad (31)$$

Puting it into the Equation (26), we obtain the recursive equation:

$$Bh\_Sort[m] = OddEven[m] \oplus (OE\_Sort[\frac{m}{4}] \ominus OE\_Sort[\frac{m}{4}] \ominus OE\_Sort[\frac{m}{4}]$$
$$\ominus OE\_Sort[\frac{m}{4}]) \oplus (Merg[\frac{m}{2} \ominus Merg[\frac{m}{2}) \oplus Shuffle[m] \quad (32)$$

Because the four sub-sequences $(v_1, v_3, \cdots, v_{\frac{m}{2}-1})$, $(v_{\frac{m}{2}+1}, v_{\frac{m}{2}+3}, \cdots, v_{m-1})$, $(v_2, v_4, \cdots, v_{\frac{m}{2}})$ and $(v_{\frac{m}{2}+2}, v_{\frac{m}{2}+4}, \cdots, v_m)$ all are sorted, the four sorters $OE\_Sort[\frac{m}{4}]$ can be eliminated.

Therefore, we have:

$$Bh\_Sort[m] = OddEven[m] \oplus (Merg[\frac{m}{2}] \ominus Merg[\frac{m}{2}]) \oplus Shuffle[m] \qquad (33)$$

The schematic equations are shown in Figure 19.

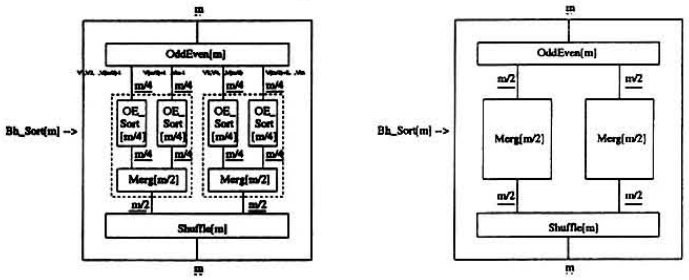


Figure 19:

Step 8: Since

$$\begin{aligned} Be\_Sort[m] &= (OE\_Sort[\frac{m}{2}] \ominus OE\_Sort[\frac{m}{2}]) \oplus Bt\_Sort[m] \\ &= (OE\_Sort[\frac{m}{2}] \ominus OE\_Sort[\frac{m}{2}]) \oplus (Bh\_Sort[m] \oplus (| \ominus COMP[m-2] \ominus |)) \\ &= (OE\_Sort[\frac{m}{2}] \ominus OE\_Sort[\frac{m}{2}]) \oplus (OddEven[m] \oplus (Merg[\frac{m}{2}] \ominus Merg[\frac{m}{2}]) \\ &\quad \oplus Shuffle[m] \oplus (| \ominus COMP[m-2] \ominus |)) \end{aligned} \qquad (34)$$

Let

$$Merg[m] = OddEven[m] \oplus (Merg[\frac{m}{2}] \ominus Merg[\frac{m}{2}]) \oplus Shuffle[m] \oplus (| \ominus COMP[m-2] \ominus |) \qquad (35)$$

we obtain

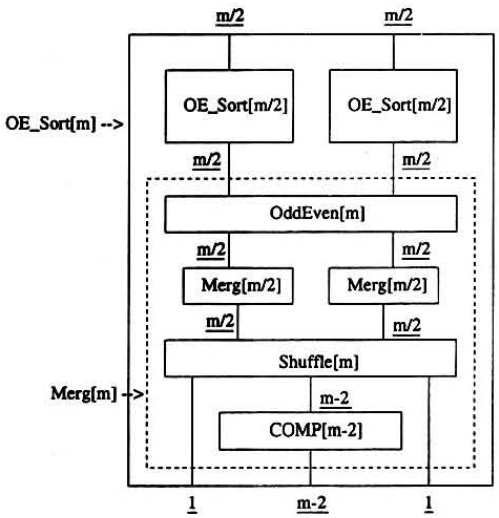


Figure 20:

$$Be\_Sort[m] = OE\_Sort[m]$$

Up to now, we obtain a recursive equation array to design odd-even-merging sorter $OE\_Sort[m]$:

$$OE\_Sort[m] = (OE\_Sort[\frac{m}{2}] \ominus OE\_Sort[\frac{m}{2}]) \oplus Merg[m]$$

$$Merg[m] = OddEven[m] \oplus (Merg[\frac{m}{2}] \ominus Merg[\frac{m}{2}]) \oplus Shuffle[m] \oplus (| \ominus COMP[m-2] \ominus |)$$

Their schematic equations are shown in Figure 20, in which the dotted line part is just the recursive equation of $Merg[m]$.

Now, we complete the semantics-preserving derivation from the basic sorter $B2\_Sort[m]$ to $OE\_Sort[m]$. The derivation procedure is not only to guarantee the correctness, but also is constructive. The derivation result naturally gives the recursive equation to construct merging cell $Merg[m]$. This is just one of advantages using our formal method to design recursive circuits.

4.3 Design based on global partitions

Now consider the basic sorter $B1\_Sort[2^n]$ (see Figure 11). Since the order of its input elements does not affect the result, we can add two circuits $B1\_Sort[2^{n-1}]$ as shown in Figure 21.

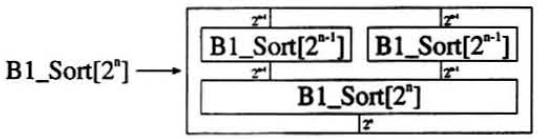


Figure 21:

$$B1\_Sort[2^n] = (B1\_Sort[2^{n-1}] \ominus B1\_Sort[2^{n-1}]) \oplus B1\_Sort[2^n]$$

Here we get the first similarity to the circuit $OE\_Sort[2^n]$.

Assuming that $OE\_Sort[2^{n-1}] \Leftrightarrow B1\_Sort[2^{n-1}]$ ($OE\_Sort[1] = B1\_Sort[1]$), we need only to prove that $Merg[2^n] \Leftrightarrow B1\_Sort[2^n]$.

Note that the circuit $B1\_Sort[2^n]$ consists of $n-1$ 'slices', where the j-th slice is the circuit $Slice[j]$. We define the operation Cut that 'cuts' the circuits $B1\_Sort[n]$ and $Slice[n]$ in two peaces:

$$Cut(CMP, 1) := |^2$$

that means, it substitutes CMP with two vertical lines. Applying it at $Slice[n]$, we get:

$$Cut(Slice[i+k], i) := Slice[i] \ominus Slice[k]$$

It cuts the i-th CMP in $Slice[i+k]$ (see Figure 22)

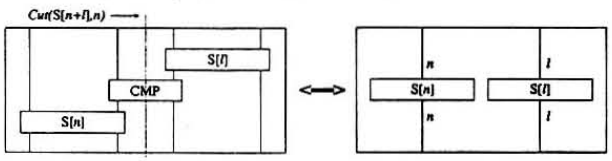


Figure 22:

Now we define $Cut(B1\_Sort[n+1], l)$. It cuts each j-th slice of $B1\_Sort[n+1]$ according to $Cut(Slice[j], l)$:

$$Cut(B1\_Sort[n+1], l) \Leftrightarrow Cut(B1\_Sort[n], l) \oplus Cut(Slice[n+1], l)$$

Figure 23 shows an example of $Cut(B1\_Sort[4], 2)$.

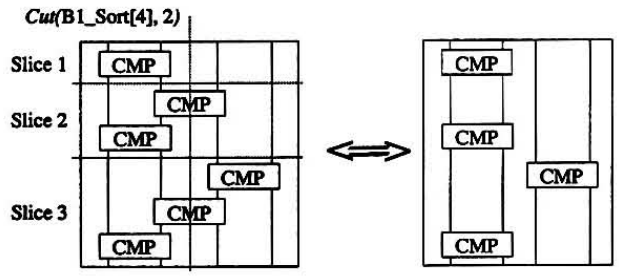


Figure 23:

Lemma 5 *For $\forall n \in \mathbb{N}$, the following transformation holds:*

$$Cut(B1\_Sort[2^n], 2^{n-1}) \Leftrightarrow B1\_Sort[2^{n-1}] \ominus B1\_Sort[2^{n-1}]$$

Proof: It is easy to check that the transformation holds for $n = 4$:

$$Cut(B1\_Sort[4], i) \Leftrightarrow B1\_Sort[i] \ominus B1\_Sort[4 - i] \quad (i = \overline{1,3})$$

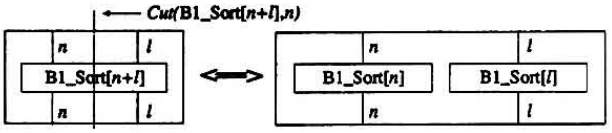


Figure 24:

We assume that the transformation $Cut(B1\_Sort[n+l], n) \Leftrightarrow B1\_Sort[n] \ominus B1\_Sort[l]$ holds for some fixed n and l (Figure 24). Now consider $Cut(B1\_Sort[n+l+1], n)$ (Figure 25):

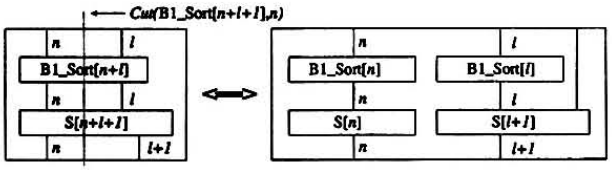


Figure 25:

Since $B1\_Sort[n] \oplus Slice[n] \Leftrightarrow B1\_Sort[n]$ and $(B1\_Sort[l] \ominus |) \oplus Slice[l+1] = B1\_Sort[n+1]$, we have (see Figure 26 as well):

$$Cut(B1\_Sort[n + l + 1]) \Leftrightarrow B1\_Sort[n] \ominus B1\_Sort[l + 1]$$

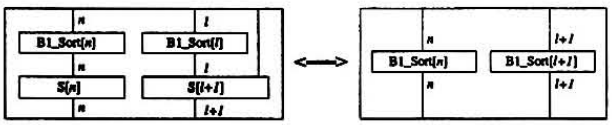


Figure 26:

So, there are $Cut(B1\_Sort[n + l], n) \Leftrightarrow B1\_Sort[n] \ominus B1\_Sort[n]$ for $\forall n, l \in \mathbb{N}$ and $Cut(B1\_Sort[2^n], 2^n) \Leftrightarrow B1\_Sort[2^{n-1}] \ominus B1\_Sort[2^{n-1}]$ Q.E.D.

Assume that $B1\_Sort[2^{n-1}] \Leftrightarrow Merg[n-1]$, $(B1\_Sort[2] = Merg[1])$. Then, we get

$$B1\_Sort[2^n] \Leftrightarrow Merg[n-1] \ominus Merg[n-1]$$

From this assumption and $Cut(B1\_Sort[2^n], 2^{n-1}) \Leftrightarrow B1\_Sort[2^{n-1}] \ominus B1\_Sort[2^{n-1}]$, we have

$$B1\_Sort[2^n] \Leftrightarrow Merg[n-1] \ominus Merg[n-1]$$

Now consider a circuit in Figure 27. The sequences A, B, A_1, A_2, C, D and S are defined as follows:

$A = (a_1, a_2, ..., a_{2^{n-1}})$, $B = (b_1, b_2, ..., b_{2^{n-1}})$,
$A_1 = (a_1, a_3, ..., a_{2^{n-1}-1}, b_2, b_4, ..., b_{2^{n-1}})$,
$A_2 = (a_2, a_4, ..., a_{2^{n-1}}, b_1, b_3, ..., b_{2^{n-1}-1})$,
$C = (c_1, c_2, ..., c_{2^{n-1}})$, $D = (d_1, d_2, ..., d_{2^{n-1}})$,
$S = (s_1, s_2, ..., s_{2^n})$
(note that for $\forall i, k \in N$, $i + k \leq 2^{n-1}$, $a_i \geq a_{i+k}$ and $b_i \geq b_{i+k}$). The sequence

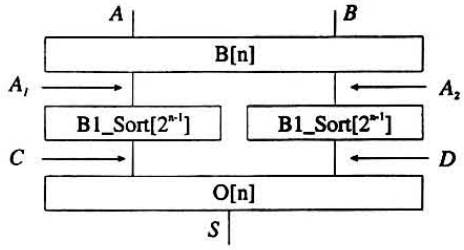


Figure 27:

(A_1, A_2) is exactly the same as the output sequence of $OddEven[n]$, so we can substitute the circuit $O[n]$ with $OddEven[n]$.

Assuming that $B1\_Sort[2^{n-1}]$ sorts the input elements correctly (obviously, $B1\_Sort[2]$ does), we get an obvious result: the maximal and the minimal elements of the sequence $\{A; B\}$ are $s_1 = max\{a_1, b_1\}$ and $min\{a_{2^{n-1}}, b_{2^{n-1}}\}$.

Consider the case when $c_2 = b_1$. Following combinations are possible:

1. $d_2 = a_2 \Rightarrow d_1 = b_2 \Rightarrow c_2 = b_1 \geq b_2 \geq a_2 = d_2$;

2. $d_2 = b_2 \Rightarrow c_2 = b_1 \geq b_2 = d_2$.

That means, $c_2 \geq d_i$ for $j \in \{2, ..., 2^{n-1}\}$. Analogously, we can prove: $d_1 \geq c_j$ for $j \in \{3, ..., 2^{n-1}\}$.

Generalizing this principle, we can prove: to get the $2i$-th and $2i-1$-th highest elements ($i \in \{1, ..., 2^{2n-2}\}$) of the sequence $\{A; B\}$, we have to calculate $CMP\{c_{2i}, d_{2i-1}\}$ for $i \in \{1, ..., 2^{2n-2}\}$. Thus, to get a correct sorting system, we have to substitute the circuits $B[n]$ and $O[n]$ by $OddEven[n]$ and $Shuffle \oplus (| \ominus COMP[n-2] \ominus |)$ (see Figure 20), respectively. That means,

$$\begin{array}{lll} B1\_Sort[2^n] & & \\ & \Leftrightarrow & OddEven[n] \oplus (B1\_Sort[2^{n-1}] \ominus B1\_Sort[2^{n-1}]) \\ & & \oplus Shuffle \oplus (| \ominus COMP[n-2] \ominus |) \\ & = & OddEven[n] \oplus (OE\_Sort[2^{n-1}] \ominus OE\_Sort[2^{n-1}]) \\ & & \oplus Shuffle \oplus (| \ominus COMP[n-2] \ominus |) \\ & = & OE\_Sort[2^n] \end{array}$$

5 Conclusions

Deriving correct recursive circuits using semantics-preserving transformations is a feasible way. We have derived a recursive basic sorter into a recursive odd-even-merging sorter

by using two approaches, one being based on local incremental transformations and the other being based on global partitions. The results show that there are circuits of practical interest, which can derived formally by using this method.

The algebraic calculus of nets is a hardware-specific calculus, and the transformations are the circuit transformations themselves. Thus, it is much better adapted to the synthesis domain. The calculus of nets has a small core of transformations. Some transformations are axiomatized and others can be derived from these axiomatized transformations by using exact algorithms or general algebraic rules. Universal transformations are usually algorithmic ones, and independent of applied fields. Synthesis-specific transformations are, to a great extent, dependent on applied domains. In synthesis-specific transformations, some are algorithmic ones and some *ad hoc* ones. The correctness of algorithmic transformations can be guaranteed by the correctness of exact algorithms and algebraic rules. *ad hoc* transformations are usually based on heuristic strategies, and their correctness can be proved by manual proofs or mechanical checkers.

The specification validation can be solved by using conceptually simple specification for the required function, such as some school methods for arithmetic circuits. In many cases, the correctness of conceptually simple specifications is obvious, or can be easily proved. Because the specification can be described compactly and graphically with a small kernel of recursive equations by using 2dL language, the synthesis task is simplified to transform only these recursive equations in the kernel.

Our formal synthesis method is not automatic but must be controlled by designers. Therefore, it is necessary to establish an interactive synthesis environment. Such an environment is being developed under the CADIC framework. The environment includes a library of semantics-preserving transformation rules, and contains tools for the hierarchical inspection of designs and to support the applications of transformations to parts of the design. The library of transformation rules will continually be expanded by deriving new semantics-preserving transformation rules in the practice.

References

[BeBH90] B.Becker, T.Burch, G.Hotz, D.Kiel, R.Kolla, P.Molitor, H.G.Osthof, U.Sparmann, "A Graphical System for Hierarchical Specifications and Checkups of VLSI Circuits," *Proc. of EDAC'90*, 1990, pp.174-179

[BeHK87] B.Becker, G.Hotz, R.Kolla, P.Molitor and H.G.Osthof, "Hierarchical Design Based on a Calculus of Nets", *Proc. of DAC'87*, 1987, pp.649-653

[BoJo93] B.Bose and S.D.Johnson, "DDD-FM9001: Derivation of a Verified Microprocessor, An Exercise in Integrating Verification with Formal Derivation", *Proc. Conf. on Correct Hardware Design and Verification Methods, LNCS683*, 1993, pp.191-202

[Busc92] H.Busch, "Transformational Design in a Theorem Prover", *Proc. of International Conference on Theorem Provers in Circuit Design : Theory, Practice and Experience, V.Stavridou etc. (Eds.)*, 1992, pp.175-196

[Camp89] R.Camposano, "Behavior-Preserving Transformations for High-Level Synthesis", *Proc. of Hardware Specification, Verification and Synthesis: Mathematical Aspects, LNCS408*, 1989, pp.106-128

[Clau71] V.Claus, "Ein Vollständigkeitssatz für Programme und Schaltkreise", Acta Informatica 1, pp.64-78, 1971

[EiKu95] D.Eisenbiegler, R.Kumar, "Formally Embedding Existing High Level Synthesis Algorithms", *Proc. of CHARME'95, LNCS987*, 1995, pp.71-83

[FFFH89] S.Finn, M.P.Fourman, M.Francis, R.Harris, "Formal System Design – Interactive Synthesis Based on Computer-Assisted Formal Reasoning", *Proc. of IMEC-IFIP Intl. Workshop on Applied Formal Methods in Correct VLSI Design, L.Claesen (Ed.)*, 1989, pp.97-110

[HaLD89] F.K.Hanna, M.Longley, N.Daeche, "Formal Synthesis of Digital Systems", *Proc. of IMEC-IFIP Intl. Workshop on Applied Formal Methods in Correct VLSI Design, L.Claesen (Ed.), North-Holland*, 1989, pp.532-548

[HoBZ95] G.Hotz, T.Burch and B.Zhu, *CADIC: A Top Down VLSI Design System – User's Guide*, Fachbereich 14 - Informatik, SFB 124, Universität des Saarlandes, Germany, September 1995

[HoRe96] G.Hotz and A.Reichert, "Hierarchischer Entwurf komplexer Systeme", in *I.Wegener (Ed.): Highlights aus der Informatik*, Springer Verlag, 1996

[Hotz65] G.Hotz, "Eine Algebraisierung des Syntheseproblems für Schaltkreise", *EIK 1*, pp.185-205, *EIK 2*, pp.208-231, 1965

[Hotz74] G.Hotz, *Schaltkreistheorie*, Walter de Gruyter · Berlin · New York, 1974

[HoZh97] G.Hotz and B.Zhu, "Verifying Parametrized Recursive Circuits Using Semantics-Preserving Transformations of Nets", *Technical Report 11/1997, SFB 124-B1, University of Saarland*, 1997

[JoSh90] G.Jones and M.Sheeran, "Circuit Design in Ruby", *Formal Methods for VLSI Design, J.Staunstrup (Ed.), North-Holland*, 1990, pp.13-70

[KBES96] R.Kumar, Ch.Blumenröhr, D.Eisenbiegler and D.Schmid, "Formal Synthesis in Circuit Design – A Classification and Survey", *Proc. of FMCAD'96, LNCS1166, M.Srivas and A.Camilleri Eds.*, 1996, pp.294-309

[Knut73] D.E.Knuth, "*Sorting and Searching, The Art of Computer Programming*", Addison-Wesley, 1973

[KnWi92] D.W.Knapp, M.Winslett, "A Prescriptive Model for Data-path Hardware", *IEEE Trans. on CADICAS, Vol.11, No.2*, 1992

[Koll86] R.Kolla, *Spezifikation und Expansion logisch topologischer Netze*, PhD thesis, Fachbereich 14 - Informatik, Universität des Saarlandes, Germany, 1986

[LaBr92] W.K.C.Lam and R.K.Brayton, "On Relationship Between ITE and BDD", In *Proc. of ICCD'92*, 1992, pp.448-451

[Moli88] P.Molitor, "Free Net Algebras in VLSI-Theory", *Fundamenta Informaticae*, XI, 1988, pp.117-142

[PeKu94] Z.Peng and K.Kuchcinski, "Automated Transformation of Algorithms into Register-Transfer Level Implementations", *IEEE Trans. on CADICAS, Vol.13, No.2*, 1994, pp.150-166

[SaGG92] J.B.Saxe, S.J.Garland, J.V.Guttag and J.J.Horning, "Using Transformations and Verification in Circuit Design", *Proceedings of the Second IFIP WG 10.2/WG 10.5 Workshop on Designing Correct Circuits, Edited by J.Staunstrup and R.Sharp*, 1992, pp.1-26