Verifying Parameterized Recursive Circuits Using Semantics-Preserving Transformations of Nets

Günter Hotz and Bin Zhu

Technical Report 11/1997
SFB 124 - B1

Universität des Saarlandes
Fachbereich 14 – Informatik
Postfach 151150
66041 Saarbrücken
Germany

Verifying Parameterized Recursive Circuits Using Semantics-Preserving Transformations of Nets

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

Abstract

This report will present a theorem-proving method for verifying parametrized recursive circuits. The method is based on *semantics-preserving transformations of nets* (SPTNs). Its theoretical bases are an algebraic calculus introduced by G. Hotz in 1965 and the hardware description language *2dL* based on this calculus. The starting point of our method is a high-level description using bi-category for a conceptually simple realization of the required logical function. The verification procedure is to find a sequence of semantics-preserving transformations of nets, which transform the simple realization into the implementation that fulfills the special specifications of the function as additional requirements. In this report, we will apply this concept to transforming some realizations of school methods for arithmetic functions into time efficient implementations, such as the conditional sum adder and a family of fast multipliers. The results show that there are circuits of practical interest, which can easily be verified by using this method.

Most verification approaches based on theorem-proving use complex mathematical logic, and the verification procedure is one of proving many theorems and lemmas. Therfore, using these approaches is difficult, and requires a great deal of expertises on the part of users. It is these difficulties that hinder the acceptability of theorem-proving methods in industry. Our method to be proposed will emphasize to solve these problems. First, the method uses a calculus of Boolean algebra, which design and verification engineers know well. Second, the calculus allows us simultaneously to describe functional properties and topological structures of circuits. This makes it possible to use structural information in the procedure of refining high-level behavioral specification to low-level layout implementation. In many procedures of verification, using structural information can avoid deriving complicated formulas and theorems. Finally, constructs in the calculus have a graphical interpretation. This makes it possible to express the verification procedure by using graphical descriptions rather than textual descriptions, which will facilitate understanding and manipulating verification procedures. Our method can be used not only in post-synthesis based on theorem-proving, but also in formal synthesis.

1 Introduction

With increasing design complexity of computers and other digital systems, verifying the correctness of hardware is becoming more and more important.

Usually, the approaches of formal hardware verification are classified into *pre-synthesis verification, during-synthesis verification* (also called *formal synthesis*), and *post-synthesis verification*. Pre-synthesis verification means proving the correctness of the automatic synthesis programs. However, software verification can now be performed only for very small programs (about 1000 lines of C code). Therefore, formal synthesis and post-synthesis verification make up the current mainstream in the field of hardware verification. Formal synthesis restricts synthesis to only correctness preserving logical transformations so as to guarantee that the result is not only a hardware implementation but also a proof of its correctness with respect to the specification. Post-synthesis verification is the most frequently used approach. The methods span a spectrum from the highly expressive but not-so-automated interactive theorem-provers to the significantly restricted but highly automated model-checkers and equivalence-checkers. Each class solves a subset of the problems solved by the previous class.

Currently, there are three main features in the methodology of hardware designs. The first feature is that hierarchic designs have widely been used. Designers will develop behavioral models in high-level and then directly refine them into implementations in lower-level. The design using refinement paradigm will become popular around the year 2000 [Palm97]. The second feature comes from the issue of reuse for existing designs. Use of parameterized designs in the form of standard libraries has been emphasized as an emerging trend [Geus92]. The third feature is to use recursive designs in terms of regular algorithms, such as sorting networks, adders, multipliers, FFT, and so on. Recursive implementations provide very compact representations, thereby save memory and time. This makes it possible to use visibility and navigability in the designs of large circuits.

Parametrized and recursive designs based on hierarchic refinement are sternly challenging current techniques of hardware verification. Both equivalence checking and model-checking are focused on verifying fixed-sized instances of circuits, and used as point verification tools. Existing equivalence-checkers mainly exploit internal equivalent function points between two compared circuits. But, in a parameterized circuit designed recursively, identifying equivalent function points is almost impossible since the circuit has an implicative and hierarchical description. Model-checkers cannot be used to verify parameterized circuits, and cannot even attempt to verify that a recursively defined FFT (Fast Fourier Transform) algorithm is correct. In the future, hardware verification tools will be used more to enhance the design flow. A practical verification tool should be able to prove important behavior at the high-level and to carry this proof through the hierarchic refinement phases. Therefore, formal-synthesis tools and theorem-provers will provide suitable frameworks to effectively transition hardware verification to industry [CySr95, KuBE96].

In this report, we will present a theorem-proving method. Our goal is logic-level verification for large parametrized combinational circuits, especially those that are designed recursively and hierarchically. The method is based on *semantics-preserving transformations of nets* (SPTNs). Its theoretical bases are an algebraic calculus - *D-category theory* introduced by G. Hotz in 1965 [Hotz65, Hotz74] and the hardware description language

2dL based on this calculus.

In our method, specification and implementation are described by bi-categorical expressions and hierarchic refinements. The specification is a conceptually simple realization for the required function. This realization is assumed to be correct, and called *sample circuit*. The implementation is the *circuit to be verified*. The verification procedure is to find a sequence of semantics-preserving transformations of nets, which transform the sample circuit into the circuit to be verified. In this report, we will apply this concept to transforming some school methods for arithmetic functions into time efficient implementations, such as the conditional sum adder and a family of fast multipliers. In verifying the conditional sum adder and a family of fast multipliers, a ripple-carry adder and a basic array multiplier are used as the sample circuits, respectively. The verification for the conditional sum adder is based only on the ITE (If-Then-Else) transformation, and the verification for a family of fast multipliers relies only on several simple permutations. These experimental results show that there are circuits of practical interest, which can easily be verified by using our method.

The remainder of this report is organized as follows. Section 2 gives a brief review for various methods using correctness-preserving transformations in formal synthesis and theorem-provers, and for several approaches verifying parameterized circuits. Section 3 is an introduction to the calculus of nets and the hardware description language 2dL. In section 4, we define semantics-preserving transformations of nets, and give some important transformation rules. Section 5 indicates the basic verification principle using SPTNs. In sections 6 and 7, we give two examples of verifications for arithmetic circuits, one being the verification for the conditional sum adder and the other being for a family of fast multipliers. Finally, section 8 gives conclusions and points out further work.

2 Prior Work

Correctness-preserving transformation is a technique widely used in hardware designs. It is also an essential component of most formal synthesis systems. Early in 1965, G. Hotz expounded the method of designing circuits recursively by using semantics-preserving transformations in his algebraic calculus [Hotz65]. This calculus is the first extension of boolean algebra in the direction of representing circuit structures. It was later generalized to develop a hardware description language 2dL and to establish a VLSI design system - CADIC [Koll86, Moli88, HoRe96]. The CADIC system is a hierarchical design system giving excellent graphical support for parameterized and recursive designs [BeHK87, BeBH90, BuHZ95, Burc95, Ritt97]. Other design systems using correctness-preserving transformations include Ruby, VERITAS, LAMBDA, HASH and so on. Ruby is a language based on relations and functions for specifying VLSI circuits. The synthesis tools around Ruby have been developed at the Programming Research Group - Oxford, University of Glasgow and Technical University of Denmark, Lyngby [Shee88, JoSh90, Ross90, ShRa93, ShRa95]. VERITAS is a theorem prover based on an extension of typed higher order logic, and developed by the University of Kent. In VERITAS, transformations are interactively applied for converting the specification into an implementation [HaLD89, HaDa92]. The LAMBDA proof system has also been conceived as an environment for CAD to develop digital circuits, integrating design and verification [FFFH89]. Its core is a theorem prover based on higher-order logic. This is the only commercial tool for performing formal synthesis [HFFM93, BBCC95]. HASH is based on the theorem prover HOL. It is a toolbox for implementing formal synthesis programs comprising of circuit transformations that correspond to those of conventional synthesis programs [EiKu95]. R. Camposano suggested a method of behavior-preserving transformations for high-level synthesis [Camp89]. In this method, the goal of the transformations is to meet certain hardware constraints that allow a trivial conversion of behavior into structure. H. Busch presented also an alternative design methodology that combines functional transformation with deductive theorem-proving techniques [Busc90, Busc92]. [SaGG92] showed how machine-checked verification can support an approach to circuit design based on transformations. This approach starts with a conceptually simple (but inefficient) initial design and uses a combination of *ad hoc* and algorithmic transformations to produce a design that is more efficient (but more complex).

Parameterized hardwares have been used in VLSI designs widely [BeBH90, ClFe86, DrWa92, LuVu83]. Verifying fixed-sized instances of circuits does not indicate whether arbitrary-sized instances of circuits are error-free. Even though sometimes there is indeed a specific upper bound on the number of sizes, verifying large size instances may be intractable because of state explosion. Therefore, verification of parameterized hardwares has practical significance. Its main advantage is that a single proof can serve to ensure the correctness for a family of circuits.

Some efforts verifying parameterized hardwares have been made. As early as 1986, Bühler verified the correctness of recursive descriptions of some arithmetic circuits by hand [Bühl86]. Theorem provers like HOL [CaGM86, Melh93] and the Boyer-Moore theorem prover [BoMo84, BoMo88] are suited for verification of parameterized circuits because of their ability to perform inductive proofs on recursively defined structures. German and Wang [GeWa85] did some early research on verification of parameterized hardwares with Boyer-Moore, but did not use it to verify complex practical hardware circuits. In [VeCM90], the correctness of parameterized hardware module generators is verified by using the Boyer-Moore logic and the associated theorem prover. These modules are the basic building blocks for the CATHEDRAL-II silicon compiler. Recently, Kapur and Subramaniam mechanically verified a family of parameterized multiplier circuits, including the linear array, the Wallace tree and the 7-3 multiplier, based on rewriting and induction using an automated theorem prover *RRL* (Rewrite Rule Laboratory) [KaSu96]. An approach using the automata-based technique in iterative systems was presented by [RhSo92]. This approach treats only unilateral systems. They later extended the previous work, and addressed bilateral interconnections and circular arrangements of cells [RhSo93].

Furthermore, motived by the advantages of both reasoning by induction and symbolic tautology-checking, Gupta and Fisher presented a novel method to verify parameterized circuits by combining the two techniques [GuFi93a, GuFi93b]. The approach is based on automatic symbolic manipulation of classes of inductive Boolean functions (IBF). Two kinds of IBF have been proposed, one being *linearly inductive function* (LIFs) and the other being *exponentially inductive functions* (EIFs). But, to examine other interesting inductive circuits, e.g. multipliers, may require characterization of additional classes of IBFs (other than LIFs and EIFs). Another drawback is the effects of variable orderings within IBF implementations. Especially, the problem seems to be as critical as it is for BDD-based manipulations.

Most verification methods based on theorem-proving have two main drawbacks. The first one is using complex mathematical logic to represent the behavioral specification and hardware implementation. These logic theories range from the propositional logic to first-order predicate logic, higher-order logic, modal logic (e.g. temporal logic, extended temporal logic), mu-calculus, and so on. The other drawback lies in that theorem-proving is a process of finding the proof for theorems, which often needs to derive new definitions and intermediate lemmas. Therefore, most theorem-provers require that their users are also mathematic experts for the logic theories used in verification tools. These drawbacks have seriously hindered the acceptability of theorem-provers in industrial applications.

Our method to be proposed will emphasize to solve these two drawbacks. On the one hand, the method is based on a calculus of Boolean algebra, with which design engineers are familiar. This makes it easy for designers to use this calculus to represent their specification and implementation. On the other hand, the calculus allows us simultaneously to describe functional properties and topological structures of circuits. This makes it possible to use structural information in the procedure of refining high-level behavioral specification to low-level layout implementation. In many procedures of verification, we can compare structural equivalence instead of comparing functional equivalence between the reference circuit and the circuit under verification. This avoids deriving complicated formulas and theorems. It is observed that two circuits are functionally equivalent, if both of them are structurally equivalent. But the converse is not true. Therefore, the strategy of using structural equivalence is to limit the solution space of theorem-proving to its subset. Furthermore, since constructs in the calculus have a graphical interpretation, schematic floor-plans can also be generated. This makes it possible to express verification procedure by using graphical descriptions rather than textual descriptions. The hierarchical graphical description is the most natural manner of circuit design. We believe that a logic verification tool based on graphical description will facilitate understanding and simplifying verification procedure, and be manipulated easily. Our method can be used not only in post-synthesis based on theorem-proving, but also in formal synthesis.

3 2dL Language and Formal Description of Hardware Circuits

The hardware description language 2dL can be used simultaneously to describe functional properties and topological structures of circuits. In this section, we will present only the necessary underlying knowledge. More basic theory can be found in [Hotz65, Hotz74, Koll86, Moli88, HoRe96].

3.1 Elements of the 2dL language

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*, which are used to connect polygonal lines to the rectangle. These connectors can move along the boundary, but the order among connectors cannot be changed. We define the four boundaries of each rectangle as the northern, southern, western, and eastern boundary. Notice that the northern boundary (also other boundaries) of a rectangle only distinguishes exactly one boundary of the rectangle, but it may not be directed to the "north". Furthermore, each polygonal line links up exactly two connectors. All polygonal lines are simple. That is, they do not intersect themselves, and two different polygonal lines have no common point.

In terms of D-category theory, the set of all layouts that can be transformed into each other by a sequence of geometrical deformations is called a *logic topological net*, simply *net*. These geometrical deformations do not produce overlappings and crossings, i.e. they maintain the planar topological structure of the layout. In the 2dL language, a net can be abstracted as a set of rectangles and polygonal lines. In a net, a rectangle is used as the frame, connectors on whose boundaries represent external contacts of the circuit. Other rectangles represent basic cells or macro cells, and are placed within the frame. Polygonal lines represent interconnections between different cells and between cells and external contacts. Each net has a name to indicate its function. The name can include parameters. For example, Adder, $\mathsf{\&}$, and $\mathsf{FloatingMult}[2^n]$ all are legal names. We define the set of all nets as $\mathcal{N}$. An example of a net is shown in Figure 1.

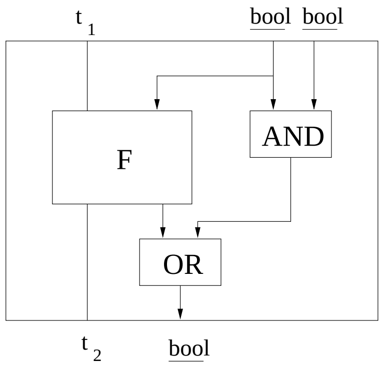


Figure 1:

A net consisting of only a basic cell is the simplest net. We define two kinds of basic cells, one being *active cells* that implement combinational logic functions, and the other being *passive cells* that are composed of only wiring structures, such as crossing, branching, unconnected end, diagonal. Figure 2 (a) and (b) show two nets, one including an active cell AND and the other including a passive cell. The set of all nets consisting of only one active cell is denoted by $\mathcal{C}_a$, and the set of all nets consisting of only one passive cell by $\mathcal{C}_p$. We define $\mathcal{C} = \mathcal{C}_a \cup \mathcal{C}_p$, and have $\mathcal{C} \subset \mathcal{N}$. Because passive cells are used, we can always represent a hardware circuit in the plane as a net that has crossing-free non-overlapping polygonal lines.

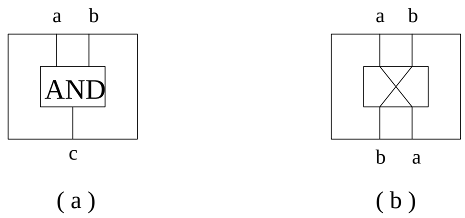


Figure 2:

3.2 Type and direction attributes

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

Types name the kinds of connectors and polygonal lines. Suppose s is a connector, and z a polygonal line. The types of s and z are denoted by $\tau(s)$ and $\tau(z)$, respectively. 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.

A connector possessing a bundled type is called a *bundled connector*. While designing hardware circuits hierarchically and recursively, parameters are widely used in bundled connectors. Thus, we employ the concept of *parametric type*. For example, suppose s is a bundled connector, and $n \in \mathbb{N}$ is a parameter. Then, $\tau(s) = \underline{bool}^n$ is a parametric type. For convenience, we use $\underline{n}$ to represent $\underline{bool}^n$ as well. In many cases, it is necessary to use *type variable*. For example, suppose the net CMP is a comparator, which sorts two numbers a and b in order. When a and b are two integers as shown in Figure 3(a), they have the same type $\tau(a) = \tau(b) = \underline{int}$. When a and b are two 1-bit number as shown in Figure 3(b), they have the same type $\tau(a) = \tau(b) = \underline{bool}$.

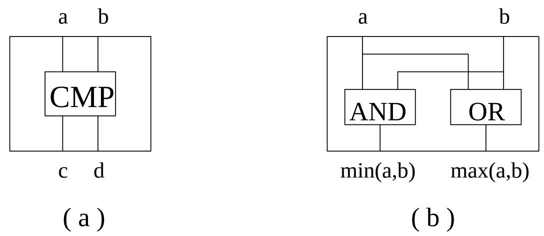


Figure 3:

The type of a polygonal line z is determined by the connectors that are incident to this polygonal line. If a polygonal line is connected to bundled connectors, it is called a *bundled polygonal line*. Suppose the polygonal line 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 type variables, the condition $\tau(z) = \tau(s_1) = \tau(s_2)$ is regarded as a "consistency condition of types".

A direction attribute indicates the flowing direction of a signal to be transferred through the corresponding connectors and polygonal lines. We stipulate that each bundled connector or bundled polygonal line has exactly one direction. $\omega(s)$ and $\omega(z)$ are used to denote the direction attributes of the connector s and the polygonal line z, respectively. 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, and their direction attributes be $\omega(s_1)$ and $\omega(s_2)$, respectively. 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. Thus, a polygonal line z can be connected to two connectors s_1 and s_2, if and only if they are compatible to each other in the directions.

3.3 Composition 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 $\psi(\underline{bool} \cdot \underline{bool}) = \psi(\underline{bool}) \times \psi(\underline{bool})$.

We can first define the *interfaces* for nets. 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. Here, we assume that the connectors on northern and southern boundaries are assigned from left side to right side, and those on western and eastern boundaries from top side to bottom side.

Example 1 *For the net G in figure 1, we have $N_\tau(G) = (t_1, \underline{bool}, \underline{bool})$, $S_\tau(G) = (t_2, \underline{bool})$, $W_\tau(G) = E_\tau(G) = \varepsilon$, $N_\omega(G) = (0, 1, 1)$ and $S_\omega(G) = (0, -1)$.*

Two kinds of composition operations are defined on the set of nets, one being the sequential operation $\ominus$, and the other being the parallel operation $\oslash$.

Let $F, G \in \mathcal{N}$. The parallel operation $F \oslash G \in \mathcal{N}$ 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 type variables, the second condition can be regarded as a consistency condition.

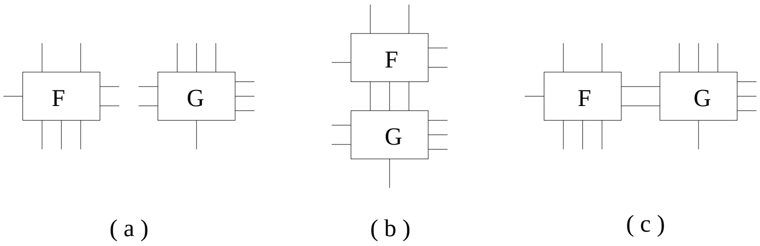


Figure 4:

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

1. $E_\omega(F) * W_\omega(G) \in \{0, -1\}^*$;
2. $E_\tau(F) = W_\tau(G)$.

Two examples of parallel and sequential operations are given in Figure 4(b) and (c), respectively.

The algebraic structure $(\mathcal{N}, \ominus, \oslash)$ is called a *bi-category*. It is clear that the following statements hold.

Lemma 1 *Given $F, G \in \mathcal{C}$ with $F \oslash G$ and $F \ominus G$ defined. Then,*

(1) $N(F \oslash G) = N(F)$, $S(F \oslash G) = S(G)$, $W(F \oslash G) = W(F) \cdot W(G)$, and $E(F \oslash G) = E(F) \cdot E(G)$;

(2) $N(F \ominus G) = N(F) \cdot N(G)$, $S(F \ominus G) = S(F) \cdot S(G)$, $W(F \ominus G) = W(F)$, and $E(F \ominus G) = E(G)$.

Lemma 2 *Given $F, G, H \in \mathcal{C}$ with $F \oslash G$, $G \oslash H$, $F \ominus G$, and $G \ominus H$ defined, respectively. Then,*

(1) $(F \oslash G) \oslash H = F \oslash (G \oslash H)$;

(2) $(F \ominus G) \ominus H = F \ominus (G \ominus H)$.

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, $F \ominus G \in \mathcal{N}$.

3.4 Refinement operation of nets

Another important operation is the refinement of nets, i.e. the replacement of each occurrence of a cell by the same net. This operation can be considered as a homomorphic mapping in our calculus. It plays an important role in the specification of nets defined hierarchically and recursively.

First, we introduce two important passive cells, one being *type factorization cell* and the other being *type product cell.*

Suppose $t_v \in T$ is a type variable. The type factorization cell realizes the function that factorizes the type t_v into a string of types $(t_1, t_2, \ldots, t_k) \in T^*$ satisfying $\psi(t_v) = \psi(t_1) \times \cdots \times \psi(t_k)$. The type product cell realizes the function that combines the type string $(t_1, t_2, \ldots, t_m) \in T^*$ into a type variable $t_w \in T$ satisfying $\psi(t_w) = \psi(t_1) \times \cdots \times \psi(t_m)$. We represent them by the symbols in Figure 5(a) and (b), respectively.

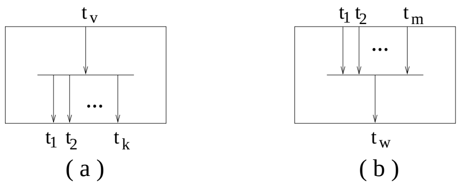


Figure 5:

We consider the bi-functor $\eta = (\eta_1, \eta_2)$ in $(\mathcal{N}, \ominus, \oslash)$. 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 $X \in \{N, S, W, E\}$.

In the net refinement, a necessary condition should be supplemented to clarify relations among type variables. This condition can be obtained by using type transformation cells. Given $F \in \mathcal{N}$ and $X \in \{N, S, W, E\}$. Suppose $X(F) = X_1(F) \cdot X_2(F) \cdot \ldots \cdot X_k(F)$, and $X(\eta_2(F)) = X'_1(\eta_2(F)) \cdot X'_2(\eta_2(F)) \cdot \ldots \cdot X'_k(\eta_2(F))$. Then, k type transformation cells $q_i \in \mathcal{C}_p$ $(i = 1, 2, \ldots, k)$ give k equations of type variables between $X(F)$ and $X(\eta_2(F))$:

$$q_i(X_i(F)) = X'_i(\eta_2(F)), \qquad i = 1, 2, \ldots, k$$

This system of equations is called the *consistency condition* for the refinement of nets.

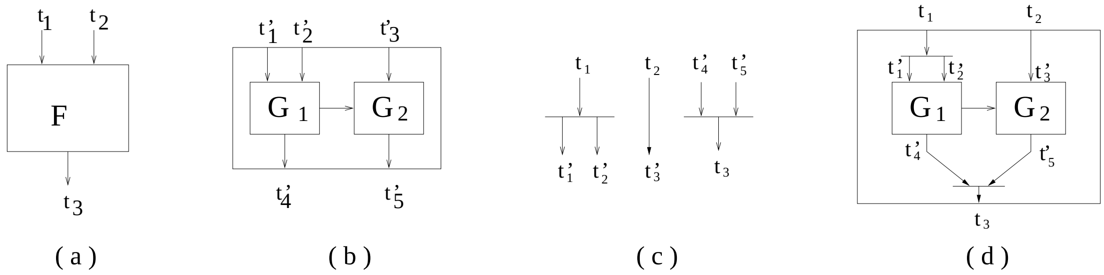


Figure 6:

Example 2 *Given $F \in \mathcal{N}$ with $N(F) = (t_1, t_2)$ and $S(F) = (t_3)$. Let $\eta_2(F) = G_1 \ominus G_2$ be the refinement of F, where $N(G_1) = ((t'_1, t'_2), (1, 1))$, $S(G_1) = ((t'_4), (-1))$, and $N(G_2) = ((t'_3), (1))$, $S(G_2) = ((t'_5), (-1))$, as shown in Figure 6(b). According to the condition $N(\eta_2(F)) = \eta_1(N(F))$, we have only the relations $t_1 \cdot t_2 = t'_1 \cdot t'_2 \cdot t'_3$ and $t_3 = t'_4 \cdot t'_5$. By means of the equations $q_1(((t_1), (1))) = ((t'_1, t'_2), (1, 1))$, $q_2(((t_2), (1))) = ((t'_3), (1))$, and $q_3(((t_3), (-1))) = ((t'_4, t'_5), (-1, -1))$ as shown in Figure 6(c), we can determine the exact relations $t_1 = t'_1 \cdot t'_2$, $t_2 = t'_3$, and $t_3 = t'_4 \cdot t'_5$. The result of the refinement is shown in Figure 6(d).*

4 Semantics-Preserving Transformations of Nets

In the field of programming languages, the syntax indicates the structure of the program, and the semantics represents the meaning of the program. Similarly, in the 2dL language, operations in the set of nets give structure relations within hardware circuits, which is just like the syntax. The function of a hardware circuit represents the meaning of the circuit, which is just like the semantics. In this section, we will discuss the semantics of nets and the semantics-preserving transformations of nets.

4.1 Interpretation of nets

It seems that the most natural way to define the behaviors of nets should be to use an algebraic expression and the functions of the elementary nets, e.g., by using homomorphisms. But, this can be done only when the flow of information is monotone, e.g., from top to bottom. There are many circuits that do not satisfy this condition, such as bistable circuits and oscillating circuits. Thus, a calculus of relations has been proposed to interpret semantics of nets so as to treat the general cases [Bühl86, Moli88, HoRe96]. That is, the semantics of a net can be defined by the set of all possible external connector configurations, for which the circuit is stable.

Let $\mathcal{T}^* = \{\psi(x) \mid x \in T^*\}$ and $F \in \mathcal{N}$. We define $r(F)$ as an actual configuration of the external connectors of F. r is a mapping $r : T^* \to \mathcal{T}^*$, which assigns a value

$r(s) \in \psi(\tau(s))$ to the connector s. The set of all possible external connector configurations is a relation, and denoted by $R(F)$. Then, the semantics of F can be described by a relation $\rho(F) \subseteq R(F)$. For example, suppose F is an active basic cell including only an AND gate. Then, $\rho(F)$ is a mapping $\rho(F) : \{0,1\}^2 \rightarrow \{0,1\}$.

Let $r(F) \in R(F)$. Suppose $s_1, \ldots, s_m$ are the connectors of F. Then, we define $r(F)|_{\{s_1,\ldots,s_m\}}$ the *restriction* of $r(F)$ with respect to the connectors $s_1, \ldots, s_m$. Especially, if F_1 is a basic cell within F, we call $r(F)|_{F_1}$ a *restriction* of $r(F)$ with respect to the connectors of F_1.

It is significant to investigate behaviors of semantics under operations of nets. Let F and G be two nets and $F \ominus G$ be defined. A formal way to describe semantics of $F \ominus G$ is to combine all configurations of F and G matching at the common border. Suppose the semantics of F and G are described by the relations $\rho(F)$ and $\rho(G)$, respectively. We represent the projections of the relation ρ on the northern, southern, western, and eastern sides by $N(\rho)$, $S(\rho)$, $W(\rho)$, and $E(\rho)$, respectively. Let $E(\rho(F)) \cap W(\rho(G)) \neq 0$, and $e_1, \ldots, e_m$ and $w_1, \ldots, w_n$ be the connectors on the eastern boundary of F and the western boundary of G, respectively. Then, obviously, there exist the interpretations $r(F)|_{\{e_1,\ldots,e_m\}}$ and $r(G)|_{\{w_1,\ldots,w_n\}}$ that are matchable in the common boundary, and the interpretation $r(F \ominus G)$ whose restrictions with respect to F and G are just $r(F)|_{\{e_1,\ldots,e_m\}}$ and $r(G)|_{\{w_1,\ldots,w_n\}}$. Each interpretation of $F \ominus G$ can be obtained by this method. Furthermore, it has been shown that $\ominus$ and $\oslash$ give not only operations for structures of nets, but also operations for the semantics of nets [HoWu95]. That is, $\forall F, G \in \mathcal{N}$, $\rho(F \ominus G) = \rho(F) \ominus \rho(G)$, $\rho(F \oslash G) = \rho(F) \oslash \rho(G)$, if $F \ominus G$ and/or $F \oslash G$ are defined. We can define the semantics of $F \ominus G$ as follows:

$$
\begin{aligned}
& \rho(F \ominus G) = \rho(F) \ominus \rho(G) \\
= \quad & \{r \mid \exists r(F) \in \rho(F), \exists r(G) \in \rho(G), E(r(F)) = W(r(G)), r|_F = r(F), r|_G = r(G)\}
\end{aligned}
$$

Similarly, we can define the semantics of $F \oslash G$. Therefore, we get a way to obtain the interpretation of a net from the interpretations of its components. We define $\mathcal{F}$ the set of all semantics of nets in $\mathcal{N}$. In the following, we use an upper case letter to represent a net, and a corresponding lower case letter to represent its semantics. For example, g is used to represent the semantics of $G \in \mathcal{N}$.

4.2 Transformations of nets

A mapping $\phi : \mathcal{N} \rightarrow \mathcal{N}$ is called a transformation of nets. The set of all transformations of nets is denoted by Φ.

Definition 1 *Let $\phi_p \in \Phi$ and $\rho : \mathcal{N} \rightarrow \mathcal{F}$ be two mappings. Given $F \in \mathcal{N}$ and its semantics $\rho(F) \in \mathcal{F}$. If $\rho(\phi_p(F)) = \rho(F)$, we call ϕ_p a* semantics-preserving transformation *of F. The set of all semantics-preserving transformations is denoted by Φ_p.*

Obviously, Φ_p is a subset of Φ. For combinational circuits, there is a complete set of semantics-preserving transformations of nets [Hotz65, Clau71]. In [Moli88], it was indicated that the complete set of semantics-preserving transformations is not an enumerable relation system in the general case.

Semantics-preserving transformations of nets can be classified into three kinds. In terms of enlarging, reducing and not changing sizes of circuits while transforming circuit structures, they are called *E-SPTNs*, *R-SPTNs*, and *N-SPTNs*, respectively.

Some examples of semantics-preserving transformations are shown in Figure 7. More examples can be found in [Moli88, HoRe96].

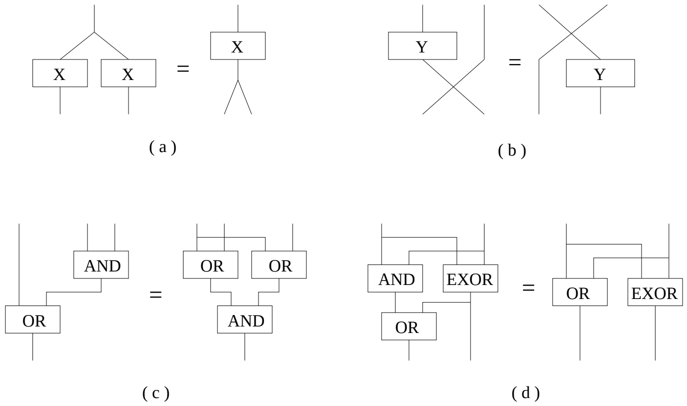


Figure 7:

An important semantics-preserving transformations is the “If-Then-Else” transformation. It belongs to E-SPTNs. Given $F \in \mathcal{N}$. We assume that $u = (t_1, \ldots, t_k)$ and $v = (t'_1, \ldots, t'_n) \in T^*$ are two strings of arbitrary types, and x has boolean type. Let 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 8 gives its diagram.

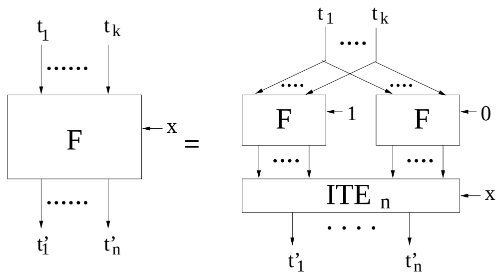


Figure 8:

Especially, if u and v are two strings of boolean type <u>*bool*</u>, 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].

ITE_n is composed of n multiplexers, and can be constructed recursively by two subcircuits $\mathrm{ITE}_{\lfloor \frac{n}{2} \rfloor}$ and $\mathrm{ITE}_{\lceil \frac{n}{2} \rceil}$ as shown in Figure 9. ITE_1 consists of one multiplexer MUX, and serves as base of the induction. It holds that $y = a_i$ for $i = x$. Figure 10 shows an example of the ITE transformation, in which both (a, b, c) and (d, e) are strings of type <u>*bool*</u>.

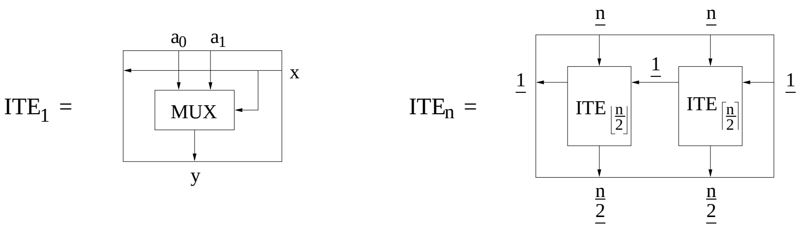


Figure 9:

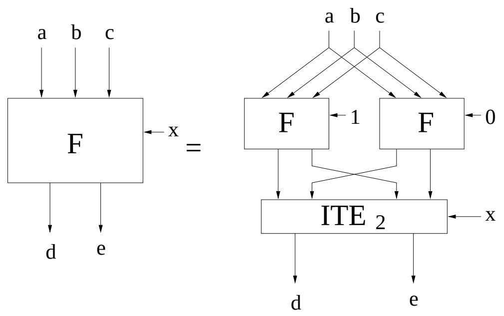


Figure 10:

4.3 Permutation

Permutation is one of the most popular transformations of nets. They do not change sizes of circuits while transforming circuit structures, and thus belong to N-SPTNs. In section 7, we will use permutations to verify a family of fast multipliers.

Definition 2 *Let U be the set of the connectors $u_1, u_2, \ldots, u_n$. A* permutation *of U is a one-to-one mapping $p : U \rightarrow U$, denoted by*

$$\begin{pmatrix} u_1 & u_2 & \cdots & u_n \\ u_{i_1} & u_{i_2} & \cdots & u_{i_n} \end{pmatrix}$$

where $p(u_1) = u_{i_1}$, $p(u_2) = u_{i_2}$, $\ldots$, $p(u_n) = u_{i_n}$. Especially, we call $\begin{pmatrix} u_j & u_k \\ u_k & u_j \end{pmatrix}$ a transposition *between u_j and u_k, and denote it $(u_j \ u_k)$, $j, k = 1, 2, \ldots, n$, $j \neq k$.*

Now, we investigate permutations in an array circuit of full adders. Given an array circuit ACF of $n \times m$ full adders $(n < m)$ in Figure 11. The subcircuit consisting of n full adders in the j-th column is denoted by ACF_j, $j = 0, 1, \ldots, m-1$ (from the rightmost column to the leftmost column). Let $x_i = x_i^{m-1} \ldots x_i^1 x_i^0$ $(i = 0, 1, \ldots, n)$ be $(n+1)$ m-bit numbers. Suppose $c_0^0 = c_1^0 = \cdots = c_{n-1}^0 = 0$. The function of ACF is to compute the sum of the $(n+1)$ m-bit numbers $x_0, x_1, \ldots, x_n$. We assume that $x_i^{m-1} = x_i^{m-2} = \cdots = x_i^{m-n+i-1} = 0$, $i = 0, 1, \ldots, n$. Then, $c_0^m = c_1^m = \cdots = c_{n-1}^m = 0$. Thus, the output of ACF is an m-bit number $s = s_n^{m-1} s_n^{m-2} \ldots s_n^1 s_n^0$, i.e.

$$\begin{aligned} value(s) &= \sum_{j=0}^{m-1} s_n^j 2^j \\ &= \sum_{j=0}^{m-1} (\sum_{i=0}^{n} x_i^j + \sum_{i=0}^{n-1} c_i^j - 2\sum_{i=0}^{n-1} c_i^{j+1}) 2^j \\ &= \sum_{j=0}^{m-1} (\sum_{i=0}^{n} x_i^j) 2^j + \sum_{i=0}^{n-1} c_i^0 - 2^m \sum_{i=0}^{n-1} c_i^m \end{aligned}$$

$$= \sum_{j=0}^{m-1}(\sum_{i=0}^{n} x_i^j)2^j$$

This describes the semantics of ACF. The circuit ACF will be used as the "Summand Summation Unit" in the basic array multiplier later.

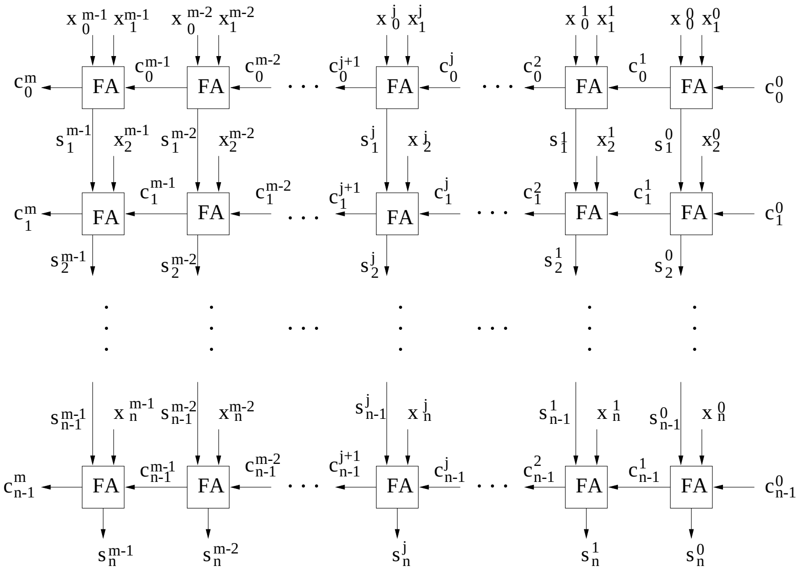


Figure 11:

In the following, we will give two lemmas about permutations in ACF, which will play important roles in verifying a family of fast multipliers. For ease of notation, we identify the name of the bit value with the name of the target connector of the directed polygonal lines.

Given the circuit ACF. Suppose $X^j = \{x_0^j, x_1^j, \ldots, x_n^j\}$ and $C^j = \{c_0^j, \ldots, c_{n-1}^j\}$. Then, $X^j \cup C^j$ is the set of all input elements in ACF_j. Let $X^{j,i} = \{x_i^j, x_{i+1}^j, \ldots, x_n^j\}$ and $C^{j,i} = \{c_i^j, c_{i+1}^j \ldots, c_{n-1}^j\}$. Let $\alpha, \beta \in X^j \cup C^j$ and $\alpha \neq \beta$. The transposition $(\alpha\ \beta)$ is equivalent to exchanging the order of adding two summands α and β, which will not affect the $value(s)$. Furthermore, let $\gamma \in X^{j,i+1} \cup C^{j,i}$. Then, the transposition $(s_i^j\ \gamma)$ is identical with altering the order of adding the summand γ and the partial sum $s_i^j = x_0^j + x_1^j + c_0^j + x_2^j + c_1^j + \cdots + x_i^j + c_{i-1}^j$ early or late. Obviously, it will not change the $value(s)$. Therefore, we have the following lemma:

Lemma 3 *It holds that*

(1) $\forall \alpha, \beta \in X^j \cup C^j$ and $\alpha \neq \beta$, the transposition $(\alpha\ \beta)$ is a semantics-preserving transformation;

(2) $\forall \gamma \in X^{j,i+1} \cup C^{j,i}$, the transposition $(s_i^j\ \gamma)$ is a semantics-preserving transformation.

Suppose $0 < i < k \leq n$ and $0 \leq m \leq (n-k)$. Let the permutation p_c be $\begin{pmatrix} c_{i-1}^j & s_{k-1}^j & s_{k+m}^j \\ s_{k-1}^j & s_{k+m}^j & c_{i-1}^j \end{pmatrix}$, which transforms ACF into the circuit illustrated in Figure 12. The permutation p_c is equivalent to first computing the partial sum $c_{i-1}^j + x_k^j + c_{k-1}^j + x_{k+1}^j + c_k^j + \cdots + x_{k+m}^j + c_{k+m-1}^j$, and then adding this partial sum to the other summands in ACF_j. This will not change the $value(s)$. Thus, the following statement holds:

Lemma 4 *The permutation* $p_c = \begin{pmatrix} c^j_{i-1} & s^j_{k-1} & s^j_{k+m} \\ s^j_{k-1} & s^j_{k+m} & c^j_{i-1} \end{pmatrix}$ *is a semantics-preserving transformation.*

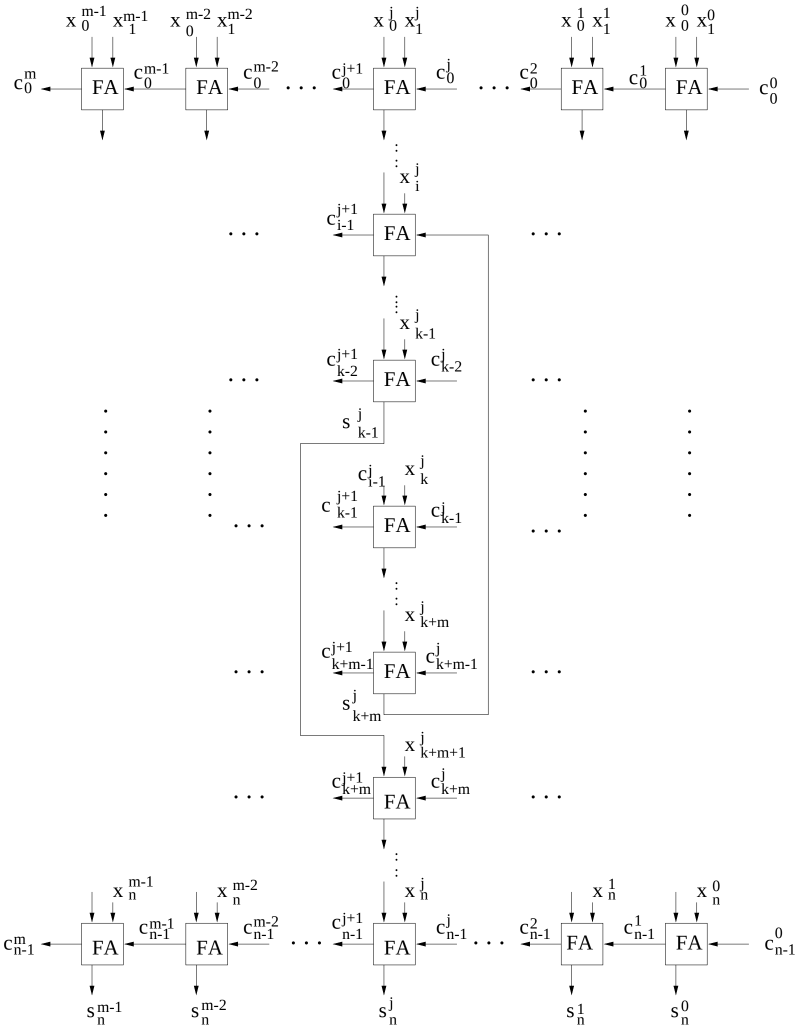


Figure 12:

Let P be the set of all permutations that are semantics-preserving transformations in ACF. Given $p_1, p_2 \in P$. We define $p_2 \cdot p_1(ACF) = p_2(p_1(ACF))$, and call $p_2 \cdot p_1$ the *product* of the permutations p_2 and p_1. Suppose $\rho(ACF)$ is the semantics of ACF. Because p_1 and p_2 are semantics-preserving transformations, it holds that $\rho(p_2 \cdot p_1(ACF)) = \rho(p_1(ACF)) = \rho(ACF)$. Thus, the product of the permutations p_1 and p_2 is a semantics-preserving transformation as well. Accordingly, we have the following important corollary:

Corollary 1 *Let* $p_1, p_2, \ldots, p_k \in P$, $k \in \mathbb{N}$. *Then, the product of the permutations* $p_k \cdot p_{k-1} \cdot \ldots \cdot p_1$ *is a semantics-preserving transformation.*

5 Basic Principle of Verification Using SPTNs

Our method is based on comparing the implementation with a sample circuit. Usually, sample circuits are conceptually simple circuits that realize fundamental algorithms. One typical example of sample circuits is the ripple-carry adder, which can be constructed easily by realizing a well-known school method of addition. By using this sample circuit, we can verify other addition circuits, such as the conditional sum adder, the carry lookahead adder, and so on. Furthermore, since our goal is the verification of large and complicated circuits, those circuits that have already been proved to be correct can be used as sample circuits as well.

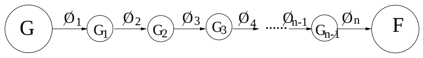


Figure 13:

Given $\rho : \mathcal{N} \rightarrow \mathcal{F}$ and $\phi_i \in \Phi_p$ $(i = 1, 2, \ldots, n)$. Let $G \in \mathcal{N}$ be the net of a sample circuit with semantics $\rho(G)$, and $F \in \mathcal{N}$ the net of the hardware circuit to be verified with semantics $\rho(F)$. The verification problem is to find a sequence of semantics-preserving transformations $(\phi_1, \ldots, \phi_n)$ satisfying:

(1) $\phi_1(G) = G_1, \phi_2(G_1) = G_2, \ldots, \phi_n(G_{n-1}) = F$;

(2) $\rho(G) = \rho(G_1) = \cdots = \rho(F)$.

This principle is illustrated in Figure 13.

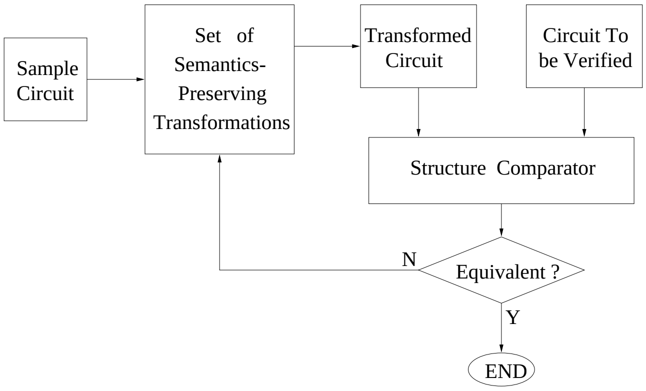


Figure 14:

In our method, an important step is the structure comparison.

After the sample circuit is transformed each time, the transformed circuit must be compared with the circuit to be verified structurally. If both circuits are structurally equivalent, the transformations will end. Otherwise, the transformations should continue. Here, the structural equivalence means that both nets corresponding to the transformed circuit and the circuit to be verified belong to the same logic topological net. In comparing the structural equivalence, we mainly use three strategies:

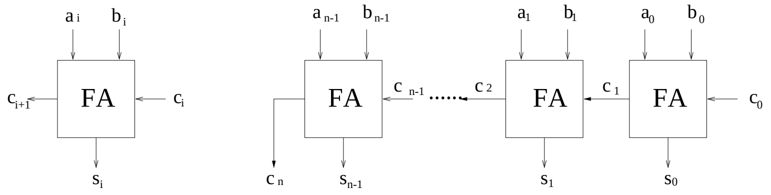
1. Checking syntax DAG of both circuits;
2. Checking I/O ports (the numbers of pins, pads and lines);
3. Checking netlist of both circuits in the same level for hierarchical designs.

If $\forall F, G \in \mathcal{N}$ are structurally equivalent, we denote them by $F \stackrel{s.e.}{=} G$. If they are functionally equivalent, we denote them by $F \stackrel{f.e.}{=} G$.

The diagram of the verification system is illustrated in Figure 14.

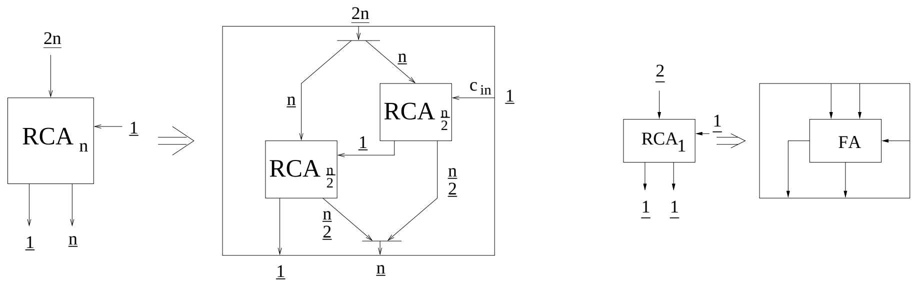
6 Verifying the Conditional Sum Adder

In this section, we will verify the conditional sum adder by using semantics-preserving transformations of nets. The sample circuit is the ripple-carry adder. Both sample circuit and the circuit to be verified have been specified in the CADIC system by using 2dL language. The verification procedure is based only on the ITE transformation.



(a) (b)

Figure 15:



(a) (b) (c)

Figure 16:

6.1 Ripple-Carry adder

A 1-bit full adder (FA) adds two binary digits a_i, b_i and one carry-in c_i to produce a sum output s_i and a carry-out c_{i+1} as shown in Figure 15(a). The two outputs are related to the three inputs by the following Boolean equations.

$$s_i = a_i \oplus b_i \oplus c_i$$

$$c_{i+1} = a_i b_i \vee b_i c_i \vee c_i a_i$$

We can connect n 1-bit full adders in cascade to form an n-bit ripple-carry adder as shown in Figure 15(b). Its diagram is denoted by RCA_n, and shown in Figure 16(a). $\underline{2n}$ represents the parametric type $\underline{bool}^{2n}$, $n \in \mathbb{N}$. The input sequence is $a_{n-1}, b_{n-1}, \ldots, a_0, b_0$. We define $\underline{2n} = \underline{n} \cdot \underline{n}$ and $\underline{n} = \underline{\frac{n}{2}} \cdot \underline{\frac{n}{2}}$. Suppose $n = 2^k, k \in \mathbb{N}$. Then, RCA_n can be realized recursively by two subcircuits $RCA_{\frac{n}{2}}$ as shown in Figure 16(b). The recursive base is RCA_1, which consists of a full adder cell FA (see Figure 16(c)).

6.2 Recursive design of the conditional sum adder

There are several different implementations of recursive designs for the conditional sum adder. Here, we consider two kinds among them. Given two n-bit numbers

$a = a_{n-1} \ldots a_1 a_0$ and $b = b_{n-1} \ldots b_1 b_0$. Let $a^{(1)} := a_{n-1} \ldots a_{\frac{n}{2}}$ be the number represented by the most significant $\frac{n}{2}$ bits, and $a^{(0)} := a_{\frac{n}{2}-1} \ldots a_0$ the number represented by the least significant $\frac{n}{2}$ bits. $b^{(1)}$ and $b^{(0)}$ are defined analogously.

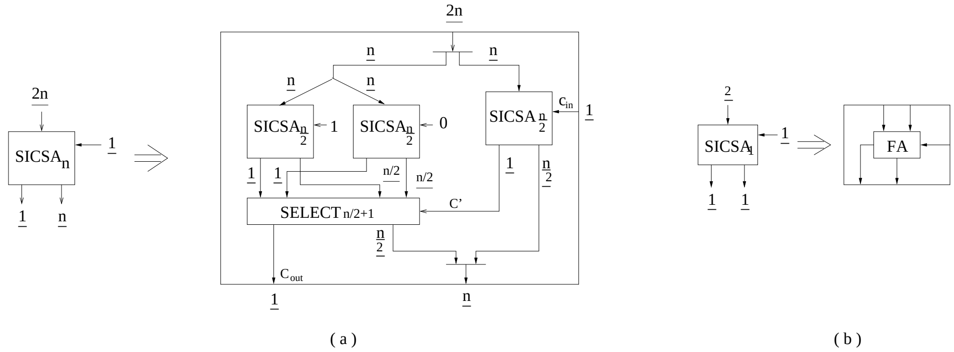


Figure 17:

A simple recursive realization has been proposed by [KePa95]. We call this recursive design SICSA_n (Simple Implementation of the Conditional Sum Adder). Let $n = 2^k, k \in \mathbb{N}$ and $\mathrm{SICSA}_{\frac{n}{2}}$ be an $(n/2)$-bit adder. Then, we can construct an n-bit conditional sum adder SICSA_n by three adders $\mathrm{SICSA}_{\frac{n}{2}}$ and an $(n/2+1)$-bit multiplexer $\mathrm{SELECT}_{\frac{n}{2}+1}$, as shown in Figure 17. In this circuit, $a^{(0)}$ and $b^{(0)}$ are sent to the $(n/2)$-bit adders on the right side to compute the least significant $\frac{n}{2}$ bits of the sum $s^{(0)} := s_{\frac{n}{2}-1} \ldots s_0$ and the carry-out c'. Whereas $a^{(1)}$ and $b^{(1)}$ are simultaneously sent to two $(n/2)$-bit adders on the left side, one having the carry-in 1 and the other having the carry-in 0. Both $(n/2)$-bit adders compute the most significant $\frac{n}{2}$ bits of the sum $s^{(1)} := s_{n-1} \ldots s_{\frac{n}{2}}$ in terms of the carry c' from the position $(n/2) - 1$ to the position $(n/2)$, respectively. SICSA_1 consists of a full adder FA, and serves as base of induction. The recursive bases of SELECT_n is SELECT_1, which is composed of a multiplexer MUX. Then, SELECT_n can be constructed by two subcircuits $\mathrm{SELECT}_{\lfloor \frac{n}{2} \rfloor}$ and $\mathrm{SELECT}_{\lceil \frac{n}{2} \rceil}$ as shown in Figure 18.

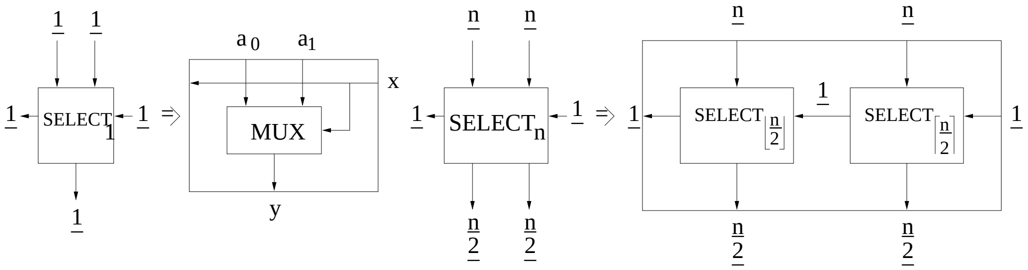


Figure 18:

Another popular recursive realization can be found in [BeHK87, BuHZ95]. Suppose we have already designed an $\frac{n}{2}$-bit adder to compute the sum and the sum plus 1 of its inputs simultaneously. Then $a+b$ and $a+b+1$ can be computed by two such $\frac{n}{2}$-bit adders and a multiplexer stage:

(1) Compute $a^{(0)} + b^{(0)}$, $a^{(0)} + b^{(0)} + 1$ and $a^{(1)} + b^{(1)}$, $a^{(1)} + b^{(1)} + 1$ by two $\frac{n}{2}$-bit adders in parallel. After that, the least significant $\frac{n}{2}$ bits of $a+b$ and $a+b+1$ are already computed.

(2) In order to complete the addition, select the most significant $\frac{n}{2} + 1$ bits of $a + b$

and $a+b+1$ by the relation

$$(a+b)_i = \begin{cases} (a^{(1)}+b^{(1)})_{i-\frac{n}{2}} & \text{if } (a^{(0)}+b^{(0)})_{\frac{n}{2}} = 0 \\ (a^{(1)}+b^{(1)}+1)_{i-\frac{n}{2}} & \text{if } (a^{(0)}+b^{(0)})_{\frac{n}{2}} = 1 \end{cases}$$

where $i = n, n-1, \ldots, \frac{n}{2}$. An analogous relation holds for $(a+b+1)_i$ as well. This results in a recursive design of the conditional sum adder in Figure 19(a), see also [BeHK87, BuHZ95]. The input sequence is $a_{n-1}, b_{n-1}, \ldots, a_0, b_0$, and the output sequence is $(a+b+1)_{n-1}, (a+b)_{n-1}, (a+b+1)_{n-2}, (a+b)_{n-2}, \ldots, (a+b+1)_0, (a+b)_0$. The inductive base is CSA_1. It consists of two full adders as shown in Figure 19(b), the left one computing the sum plus 1 and the right one computing the sum. The selector cell SEL_m can be realized recursively by two subcircuits $SEL_{\frac{m}{2}}$. SEL_1 is composed of two multiplexers MUX, and serves as base of the induction (see Figure 19(c) and (d)).

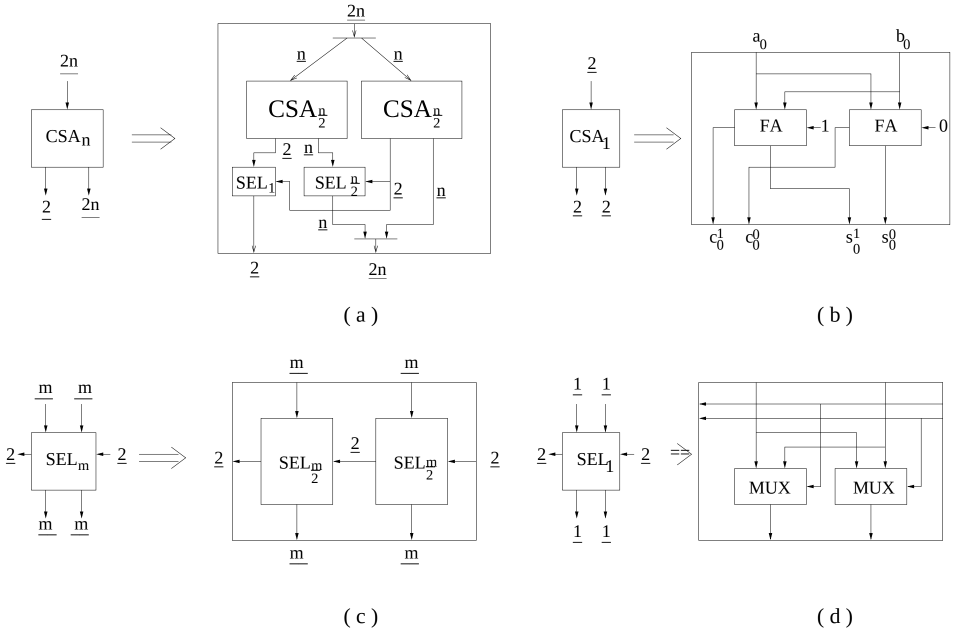


Figure 19:

6.3 Proving correctness

Verifying the correctness of SICSA$_n$ is equivalent to proving the following theorem:

Theorem 1 *The conditional sum adder SICSA$_n$ can be derived from the ripple-carry adder RCA$_n$ by using the ITE transformation.*

Proof: Applying the ITE transformation to the $(n/2)$-bit adder RCA$_{\frac{n}{2}}$ on the left side of RCA$_n$ in Figure 16(b), we obtain the new recursive design RCA'$_n$ in Figure 20. Through comparing structures of RCA'$_n$ and SICSA$_n$, we have:

RCA'$_1 \overset{s.e.}{=}$ SICSA$_1$,
ITE$_1 \overset{s.e.}{=}$ SELECT$_1$,
RCA'$_{\frac{n}{2}} \overset{s.e.}{=}$ SICSA$_{\frac{n}{2}}$,
ITE$_{\frac{n}{2}+1} \overset{s.e.}{=}$ SELECT$_{\frac{n}{2}+1}$.
Thus, it holds that RCA'$_n \overset{s.e.}{=}$ SICSA$_n$. Q.E.D.

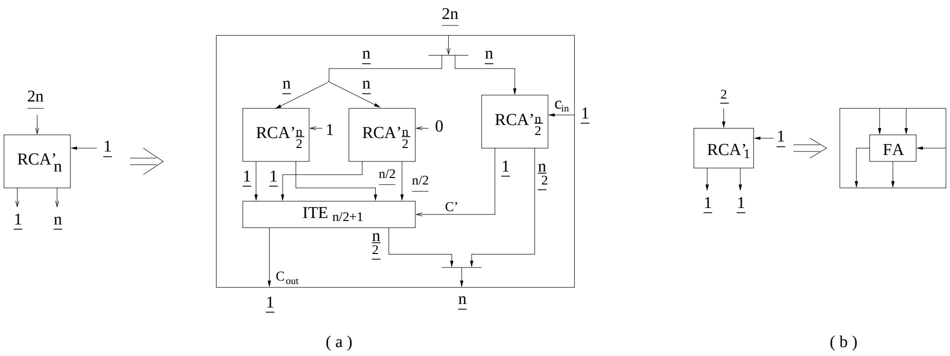


Figure 20:

In order to verify the correctness of the recursive implementation CSA_n, we need only to prove the following theorem.

Theorem 2 *The conditional sum adder CSA_n can be derived by using the ITE transformation from two ripple-carry adders RCA_n, which have the same inputs and compute the sum plus 1 and the sum of the inputs, respectively.*

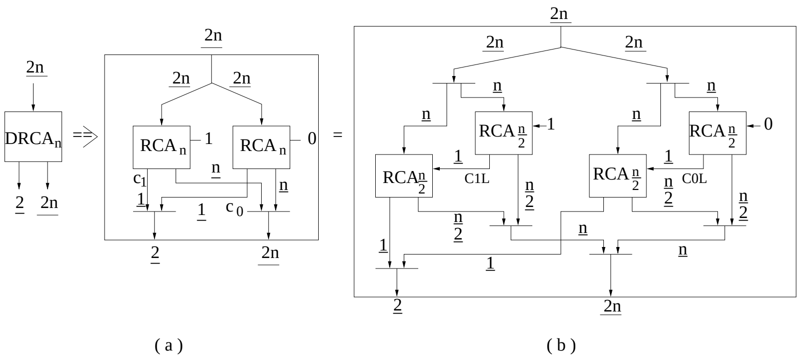


Figure 21:

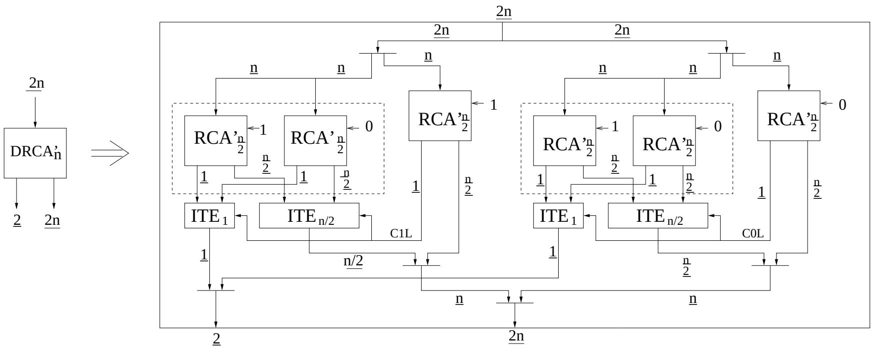


Figure 22:

Proof: Given the circuit DRCA_n consisted of two ripple-carry adders RCA_n (as shown in Figure 21(a)), in which the left and right adders are used to compute the sum plus 1 and the sum of the inputs, respectively. Each adder can be represented recursively by two subcircuits $RCA_{\frac{n}{2}}$ as shown in Figure 21(b). Applying the ITE transformation to the left-hand first and third $RCA_{\frac{n}{2}}$ cells, we obtain the circuit DRCA'$_n$ in Figure 22. There are two equal parts, one being the left two $RCA'_{\frac{n}{2}}$ of the circuit computing the sum plus 1 and the other being the left two $RCA'_{\frac{n}{2}}$ of the circuit computing the sum. Using merging transformation (see Figure 7(a)), we obtain the circuit DRCA"$_n$ as shown in Figure 23. Through comparing structures of DRCA"$_n$ and CSA_n, we have:

ITE'$_1 \stackrel{s.e.}{=}$ SEL$_1$,
ITE'$_{\frac{n}{2}} \stackrel{s.e.}{=}$ SEL$_{\frac{n}{2}}$,
DRCA"$_1 \stackrel{s.e.}{=}$ CSA$_1$,
DRCA"$_{\frac{n}{2}} \stackrel{s.e.}{=}$ CSA$_{\frac{n}{2}}$.
Thus, it holds that DRCA"$_n \stackrel{s.e.}{=}$ CSA$_n$. Q.E.D.

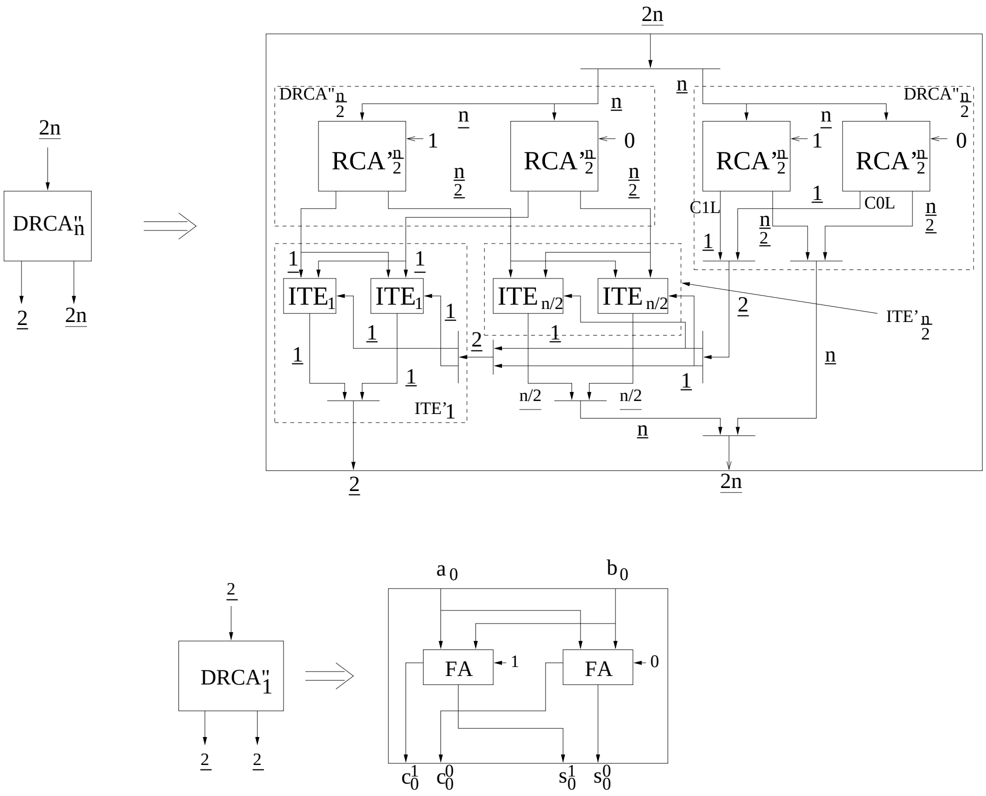


Figure 23:

7 Verifying A Family of Fast Multipliers

In this section, we will verify a family of fast multipliers. The sample circuit is a basic array multiplier, whose principle is well-known as the school method. The family of fast multipliers has a $log(n)$-time complexity, which is part of the floating point multiplier mentioned in [BeBH90]. Both circuits have be designed parametrically and recursively by using the CADIC system. Proving correctness will rely on several simple permutations.

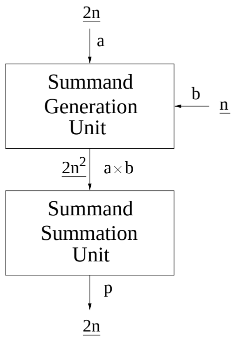


Figure 24:

7.1 Basic array multiplier

Let $a = a_{n-1} \dots a_1 a_0$ and $b = b_{n-1} \dots b_1 b_0$ be two n-bit numbers in binary representation, i.e. $value(a) = \sum_{i=0}^{n-1} a_i 2^i$ and $value(b) = \sum_{i=0}^{n-1} b_i 2^i$. The product of a and b is equal to the sum of the n $2n$-bit numbers given as the rows of the following matrix P_n:

$$P_n = \begin{pmatrix} a_{2n-1}b_0 & a_{2n-2}b_0 & \cdots & a_1 b_0 & a_0 b_0 \\ a_{2n-2}b_1 & a_{2n-3}b_1 & \cdots & a_0 b_1 & a_{-1}b_1 \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ a_n b_{n-1} & a_{n-1}b_{n-1} & \cdots & a_{2-n}b_{n-1} & a_{1-n}b_{n-1} \end{pmatrix}$$

where $a_{2n-1} = a_{2n-2} = \cdots = a_n = a_{-1} = a_{-2} = \cdots = a_{-n+1} = 0$. The term $a_i b_j$

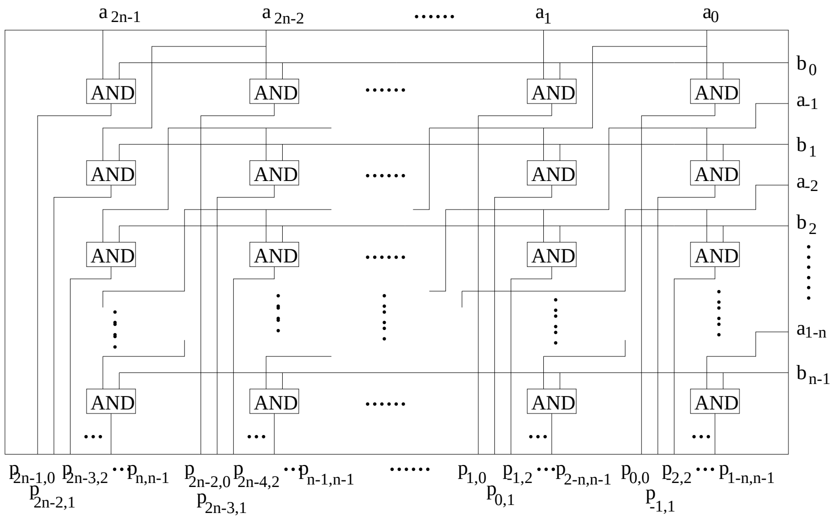


Figure 25:

means the logical 'and' of a_i and b_j. We call $a_i b_j$ a *summand*, and denote it $p_{i,j}$. The diagram of multiplier is shown in Figure 24 [Hwan79]. The summands are generated in the "Summand Generation Unit" by $2n^2$ "AND" cells (see Figure 25). Then, they are added in the "Summand Summation Unit". The school method of implementing the "Summand Summation Unit" is to add all summands in order from the rightmost column to the leftmost column in P_n and from top to bottom in each column. That is, the operation begins from the first column (the rightmost column). $p_{0,0}$ and $p_{-1,1}$ are first added. Then, their carry-out c_0^1 is sent to the second column and their sum is added to $p_{-2,2}$, ... and so on and so forth. After $p_{1-n,n-1}$ is added, we obtain the sum s_0 of the first column. Next, in the second column, $p_{1,0}$, $p_{0,1}$ and c_0^1 are first added. Their carry-out c_0^2 is sent to the third column and their sum is added to $p_{-1,2}$, ... and so on and so forth. The rest can be deduced by analogy. The final $2n$-bit number $s_{2n-1}s_{2n-1} \dots s_1 s_0$ is just the product of a and b. This school method can be implemented by an array circuit of $(n-1) \times 2n$ full adders, whose j-th column consists of $n-1$ full adders to compute the sum of the summands $p_{j,0}, p_{j-1,1}, \dots, p_{j+1-n,n-1}$ and the carries $c_0^j, c_1^j, \dots, c_{n-2}^j$, $j = 0, 1, \dots, 2n-1$. This array circuit is shown in Figure 26, where $c_0^0 = c_1^0 = \cdots = c_{n-2}^0 = 0$. The multiplier whose "Summand Summation Unit" consists of the above array circuit of full adders is called the *basic array multiplier*. In the following, we also call "Summand Summation Unit" *partial multiplier*, and the above array circuit the *partial basic array multiplier*.

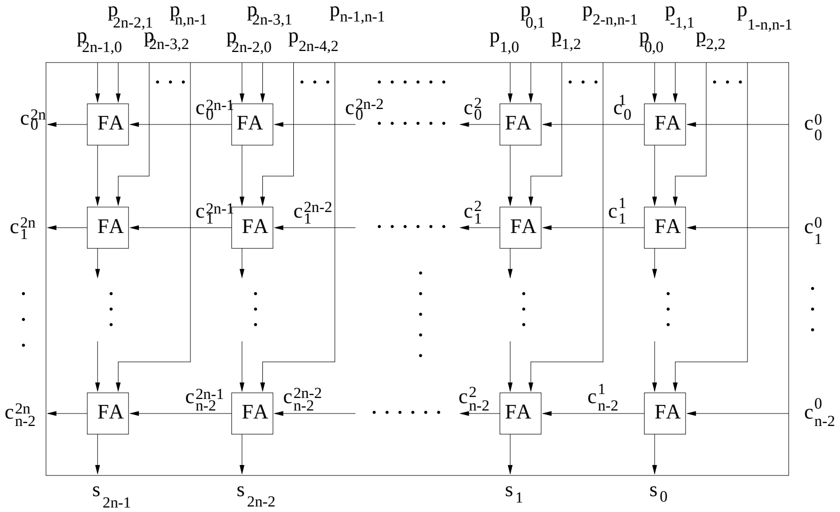
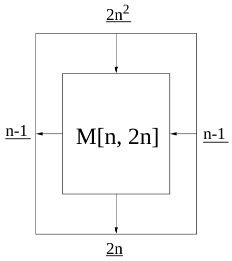


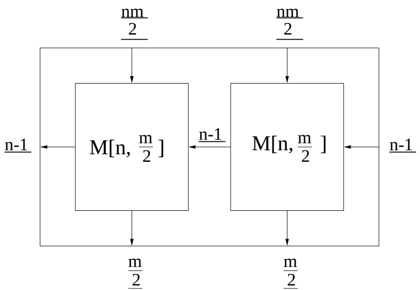
Figure 26:

The partial basic array multiplier can be realized parametrically and recursively as follows:

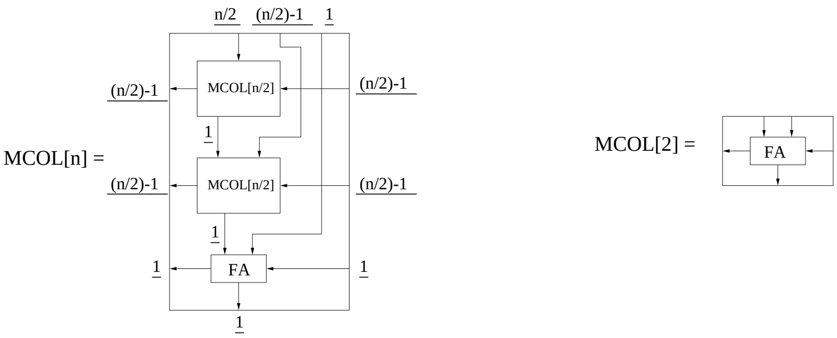
(1) An n-bit partial basic array multiplier is given by the circuit $M[n, 2n]$, where $M[n, m]$ consists of m columns of an n-bit partial basic array multiplier.



(2) In order to simplify representations, we assume that $n = 2^k$, $k \in \mathbb{N}$. $M[n, m]$ can be composed of two instances of $M[n, \frac{m}{2}]$. That is, $M[n, m] = M[n, \frac{m}{2}] \ominus M[n, \frac{m}{2}]$.



(3) $M[n,1]$ serves as base of the induction, and is denoted by $MCOL[n]$. $MCOL[n]$ can be implemented recursively by two cells $MCOL[n/2]$ and a full adder cell FA. $MCOL[2]$ consists of only a FA cell, and serves as the inductive base.

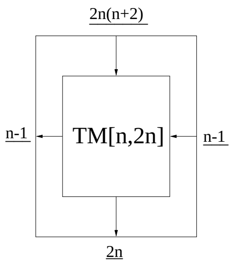


This basic array multiplier designed parametrically and recursively will be used as the sample circuit below.

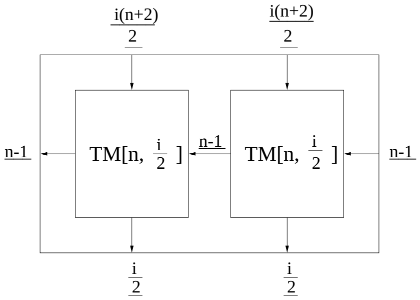
7.2 Parametric and recursive design of fast multiplier

The basic problem in designing a high-speed parallel multiplier is to reduce the time to add summands. In [LuVu83, Beck87], a $log(n)$-time multiplier is introduced. Its underlying algorithm is a Wallace tree multiplication. This design is well-suited for VLSI-implementation. In this report, we call this multiplier the *fast multiplier*. Because the "Summand Generation Unit" is equal to that of the basic array multiplier, we consider only the circuit of the "Summand Summation Unit", and call the "Summand Summation Unit" the *partial fast multiplier*. The partial fast multiplier can be implemented parametrically and recursively as follows (see also [BeBH90]):

(1) An n-bit partial fast multiplier is given by the circuit $TM[n,2n]$, where $TM[n,i]$ consists of i columns of an n-bit partial fast multiplier.

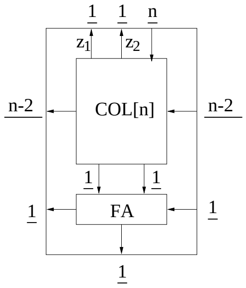


(2) In order to simplify representations, we assume that $n = 2^k$, $k \in \mathbb{N}$. $TM[n,i]$ consists of two instances $TM[n, \frac{i}{2}]$. That is, $TM[n,i] = TM[n, \frac{i}{2}] \ominus TM[n, \frac{i}{2}]$.



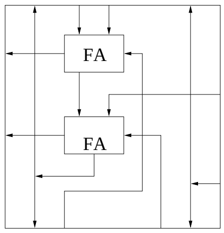
(3) $TM[n,1]$ is denoted by $TCOL[n]$ and serves as base of the induction. $TCOL[n]$ is composed of a cell $COL[n]$ and a full adder FA. Note that the polygonal lines z_1 and z_2 are convenient for recursive design. They will not affect the function of the circuit.

TCOL[n] =

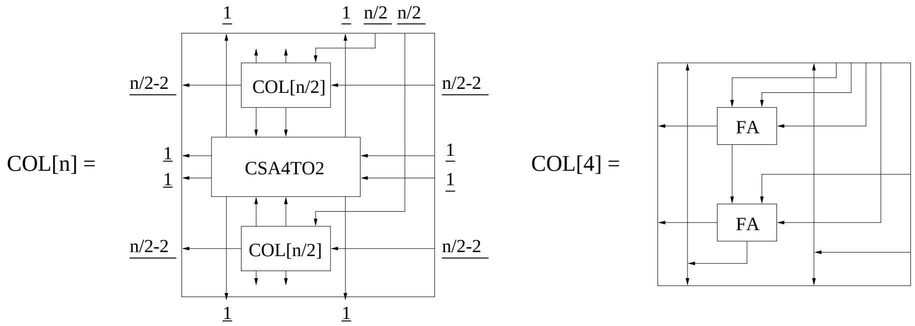


(4) Each row of matrix P_n represents a $2n$-bit number. The idea of the fast multiplier is to reduce the total number of rows in each step by half. This can be done by the cell $CSA4TO2$. A row of $2n$ such cells is used to reduce four numbers to two.

CSA4TO2 =



(5) $COL[n]$ can be implemented recursively by two $COL[n/2]$ cells and one 4to2-reduction cell $CSA4TO2$. $COL[4]$ consists of two full adders FA, and serves as the inductive base of $COL[n]$.



(a) (b)

In Figure 27, we give an example of $TCOL[8]$ to illustrate the inner construction of $TCOL[n]$.

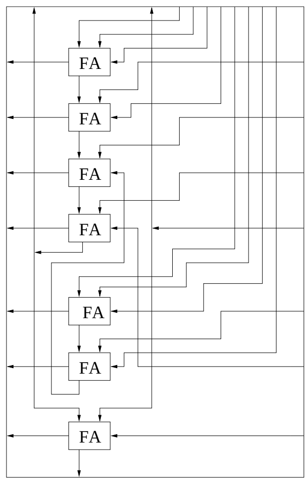


Figure 27:

7.3 Proving correctness

Verifying the correctness of the fast multiplier is equivalent to proving the following theorem.

Theorem 3 *The partial fast multiplier $TM[n, 2n]$ can be derived from the partial basic array multiplier $M[n, 2n]$ by using semantics-preserving transformations.*

Proof: We only need to prove that $TCOL[n]$ can be derived from $MCOL[n]$ by using semantics-preserving transformations. The proof will use induction, and rely on several permutations.

(1) We change $MCOL[n]$ by combining the lowest FA cell in the top $MCOL[n/2]$ and the upperest FA cell in the bottom $MCOL[n/2]$ as a new cell $CENTER$. The rest part of $MCOL[n/2]$ is denoted by $BOL[n/2]$. The cell $CENTER$ consists of two full adders FA, and $BOL[2]$ includes only two vertical wires (see Figure 28(b) and (c)). Both they serve as the inductive base of $BOL[n]$ in Figure 28(a). The new recursive description of $MCOL[n]$ is illustrated in Figure 29.

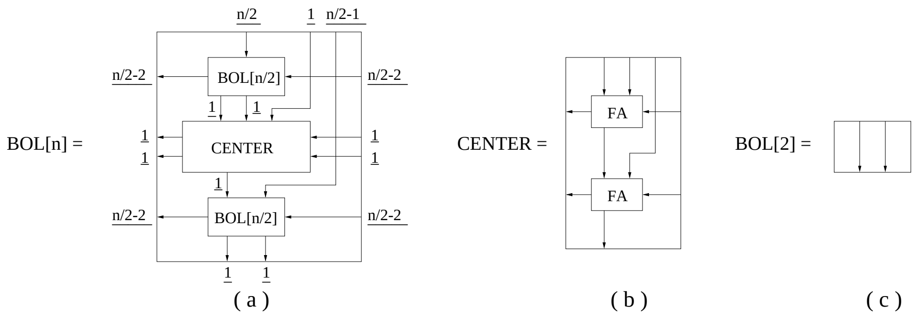


Figure 28:

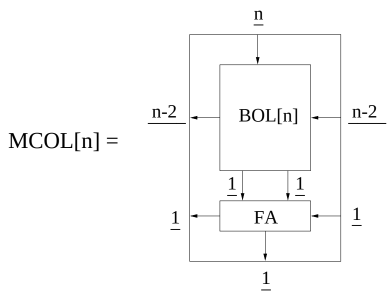


Figure 29:

(2) When $n = 4$. $MCOL[4]$ is shown in Figure 30(a). Making the transpositions $(c_0^0\ x_2)$ and $(c_1^0\ x_3)$, we obtain the circuit in Figure 30(b). Where, for purpose of recursive design, two polygonal lines z_1 and z_2 are drawn from u and v to northern boundary, respectively. They will not change the function of the circuit. This circuit is just $TCOL[4]$. Thus, the conclusion is true. Furthermore, if the outputs of $BOL[4]$ and $COL[4]$ are connected to the inputs of a full adder respectively, both they possess equivalent function. $BOL[4]$ and $COL[4]$ are shown by the dotted line frames in Figure 30(a) and (b), respectively.

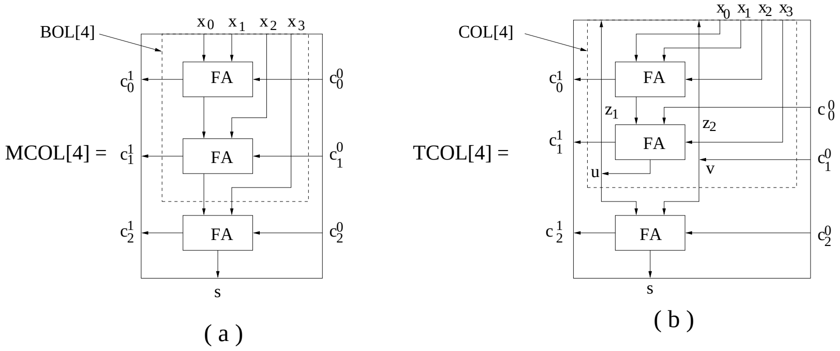


Figure 30:

(3) Suppose, for $n = m$, the conclusion is true and $BOL[m]$ and $COL[m]$ possess equivalent function. We consider $n = 2m$ due to $n = 2^k, k \in \mathbb{N}$. $MCOL[2m]$ is illustrated in Figure 31. Let the sequence of carry-in be $(c_0^0, c_1^0, \ldots, c_{2m-2}^0)$. For convenience, we draw the inner construction of $CENTER$ in the figure.

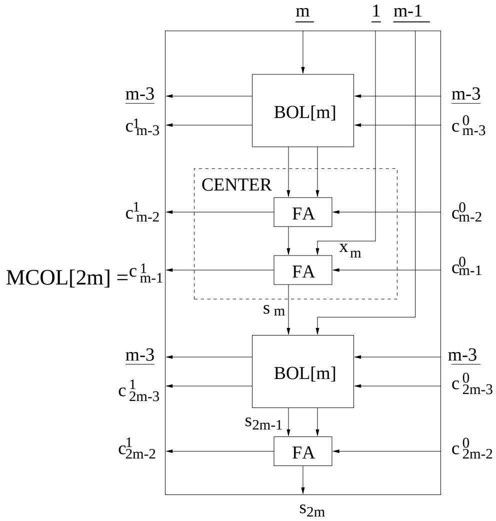


Figure 31:

(4) Because the outputs of $BOL[m]$ are connected to the inputs of a full adder FA, $BOL[m]$ and $COL[m]$ are equivalent. We replace $BOL[m]$ by $COL[m]$, as shown in Figure 32.

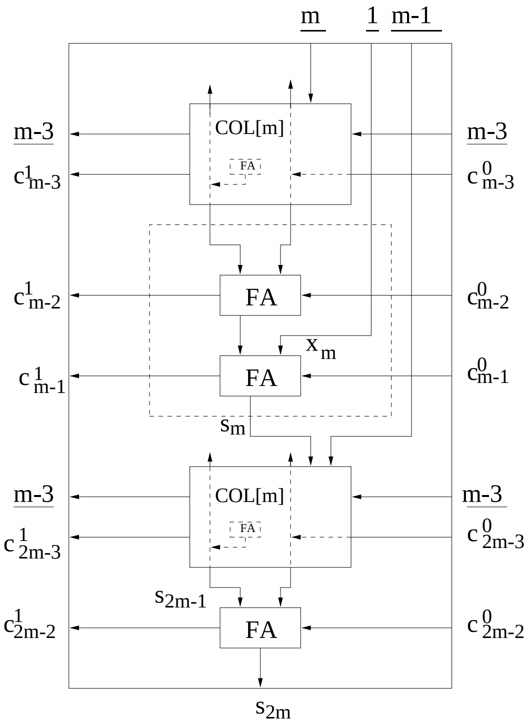


Figure 32:

(5) Making the transpositions $(x_m\ c^0_{m-2})$ and $(c^0_{m-1}\ c^0_{2m-3})$. Figure 33 gives the circuit transformed. The polygonal line z_2 is drawn from v to the northern boundary to be convenient for recursive design, which will not change the function of the circuit.

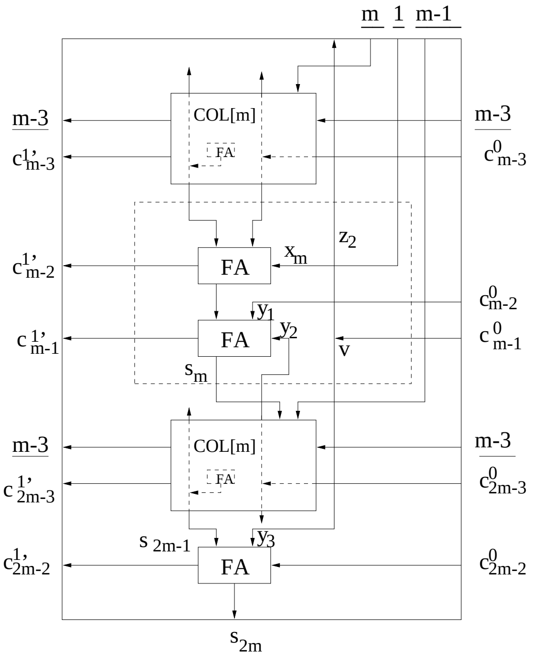


Figure 33:

(6) Finally, making the permutation $\begin{pmatrix} x_m & s_m & s_{2m-1} \\ s_m & s_{2m-1} & x_m \end{pmatrix}$, we obtain the circuit as shown in Figure 34. The polygonal line z_1 is drawn from w to the northern boundary to be convenient for recursive design, which will not change the function of the circuit. This circuit is just $TCOL[2m]$, whose middle part indicated by the dotted line frame is $CSA4TO2$ cell. Q.E.D.

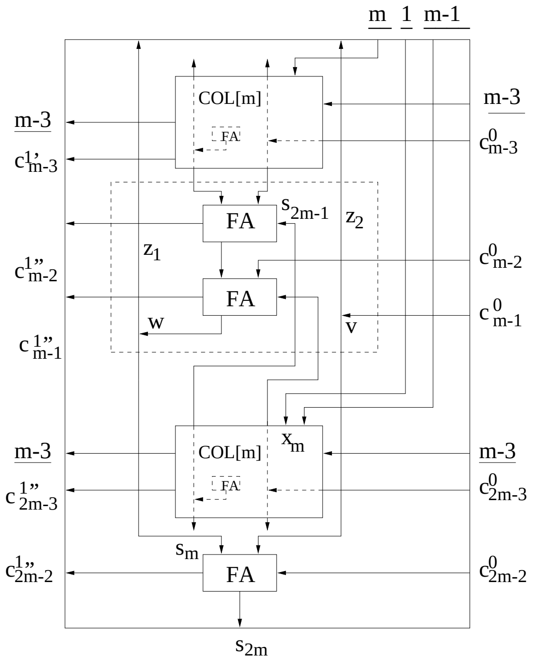


Figure 34:

8 Conclusions and Further Work

It is an efficient approach to verify parameterized recursive hardwares using semantics-preserving transformations of nets. We have successfully proved the correctness of some arithmetic circuits by using this method. The verifying procedures rely only on several simple transformation rules. The results show that the structure-proof-based theorem-proving method is feasible to verify parameterized recursive hardwares.

Rules of semantics-preserving transformations of nets can be described graphically by using the 2dL language. In verification procedures, these rules can be used directly as graphical modules by human-machine interaction. This makes the interactive process much simpler than that in other theorem provers. The user is not required to learn many reasoning skills. Furthermore, using information of structural equivalence in theorem-proving methods is an effective approach to simplify verification procedures.

The key for applying our method to practice lies in whether an efficient interactive tool can be provided. Our further work will be the development of an interactive verification tool under the CADIC framework to support the practical applications of the new method. This tool will also be a significant extension of the CADIC system. It will include a library of rules of semantics-preserving transformations of nets and an interactive environment. Similarly to the set of theorems in theorem-proving methods, the library of transformation rules will continually be expanded by deriving new semantics-preserving transformation rules in the practice. In order to build an efficient interactive environment, we must solve two problems. The first one is how to guide the choice of SPTNs rules so as to enhance the structural similarity between the transformed circuit and the circuit to be verified. The second one is how automatically to check the structural similarity. The excellent abilities of navigation and visualization in the CADIC system will be helpful to solve the former, and an automatic checker of structural equivalence will be developed to solve the latter.

References

[BBCC95] G.Bezzi, M.Bombana, P.Cavalloro, S.Conigliaro, G.Zaza, "Quantitative Evaluation of Formal Based Synthesis in ASIC Design", *Proc. of TPCD'94, LNCS901*, 1995, pp.286-291

[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

[Beck87] B.Becker, "An Easily Testable Optimal Time VLSI-Multiplier", *Acta Informatica, Vol.24, 1987*, pp.363-380

[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

[BoMo84] R.S.Boyer and J.S.Moore, "Proof-checking, Theorem-proving and Program Verification", *Contemporary Mathematics, vol. 29*, 1984, pp.119-132

[BoMo88] R.S.Boyer and J.S.Moore, "*A Computational Logic Handbook*", Academic Press, Boston, 1988

[Bühl86] H.Bühler, *Korrectheitsbeweise von rekursiv beschriebenen logisch-topologischen Netzen*, Master thesis, Fachbereich Informatik, Universität des Saarlandes, Germany, 1986

[BuHZ95] T.Burch, G.Hotz and B.Zhu, *CADIC: A Top Down VLSI Design System – User's Guide*, Fachbereich 14 - Informatik, SFB 124, Universität des Saarlandes, Germany, September 1995

[Burc95] T.Burch, *Eine graphische Arbeitsumgebung für den parametrisierten Entwurf integrierter Schaltkreise*, PhD thesis, Fachbereich Informatik, Universität des Saarlandes, Germany, 1995, pp.53-60

[Busc90] H.Busch, "Proof-based Transformation of Formal Hardware Models", *in Designing Correct Circuits, G.Jones and M.Sheeran (Eds.)*, 1990, pp.271-296

[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

[CaGM86] A.Camilleri, M.Gordon and T.Melham, "Hardware Verification using Higher-Order Logic", *Technical Report, Computer Laboratory, University of Cambridge*, 1986

[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

[ClFe86] E.Clarke and Y.Feng, "Escher - A Geometrical Layout System For Recursively Defined Circuits", *Proc. of DAC'86*, 1986, pp.650-653

[CySr95] D.A.Cyrluk and M.K.Srivas, "Theorem Proving: Not an Esoteric Diversion, but the Unifying Framework for Industrial Verification", *Proc. of ICCD'95*, 1995, pp.538-544

[DrWa92] R.Drefenstadt and T.Walle, "Parametric ASIC-Design by CADIC", *Proc. of EDAC'92*, 1992

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

[EmNa96] E.A.Emerson and K.S.Namjoshi, "Automatic Verification of Parameterized Synchronous Systems", *Proc. of CAV'96, LNCS1102*, 1996, pp.87-98

[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

[Geus92] A.J.de Geus, "High Level Design: A Design Vision for the 90's", *Proc. of ICCD'92*, 1992, pp.8

[GeWa85] S.M.German and Y.Wang, "Formal Verification of Parameterized Hardware Designs", *Proc. of ICCD'85*, 1985, pp.549-552

[GuFi93a] A.Gupta and A.L.Fisher, "Parametric Circuit Representation Using Inductive Boolean Functions", *Proc. of CAV'93*, 1993, pp.15-28

[GuFi93b] A.Gupta and A.L.Fisher, "Representation and Symbolic Manipulation of Linearly Inductive Boolean Functions", *Proc. of ICCAD'93*, 1993, pp.192-199

[HaDa92] F.K.Hanna and N.Daeche, "Dependent Types and Formal Synthesis", *Phil. Trans. R. Soc. London., Vol.339*, 1992, pp.121-135

[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

[HFFM93] R.B.Hughes, M.D.Francis, S.P.Finn, G.Musgrave, "Formal Tools for Tri-State Design in Busses", *Proc. of Higher Order Logic Theorem Proving and Its Applications, L.J.M.Claesen and M.J.C.Gordon (Eds.), Elsevier Publishers, Vol.A-20*, 1993, pp.459-474

[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*, 1965, pp.185-205, 209-231

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

[HoWu95] G.Hotz and H-Z.Wu, "On the Arrangement Complexity of Uniform Trees", *Technical Report 05/1995, FB 14, SHB 124, Universität des Saarlandes, Germany*, 1995

[Hwan79] K.Hwang, *Computer Arithmetic: Principles, Architecture, and Design*, John Wiley & Sons, 1979

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

[KaSu96] D.Kapur and M.Subramaniam, "Mechanically Verifying a Family of Multiplier Circuits", *Proc. of CAV96, LNCS1102*, 1996, pp.135-146

[KePa95] J.Keller and W.J.Paul, *Hardware Design*, B.G.Teubner, 1995

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

[KuBE96] R.Kumar, C.Blumenröhr, D.Eisenbiegler and D.Schmid, "Formal Synthesis in Circuit Design - A Classification and Survey", *Proc. of the First International Conference on Formal Methods in Computer-Aided Design, November 1996, FMCAD'96, LNCS1166*, 1996, pp.294-309

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

[LuVu83] W.K.Luk and J.Vuillemin, "Recursive Implementation of Optimal Time VLSI integer Multipliers", in *VLSI'83, F.Anceau, E.J.Aas(Eds.), North Holland:Elsevier, 1983*, pp.155-168

[MaRS:86] H.D.Man, J.Rabaey, P.Six and L.Claesen, "Cathedral-II: A Silicon Compiler for Digital Signal Processing", *IEEE Design and Test of Computers, vol.3, No.6*, December 1986, pp.73-85

[Melh93] T.Melham, *Higher Order Logic and Hardware Verification*, Cambridge University Press, 1993

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

[Palm97] G.D.Palma, "What's Next in Formal Verification", Technical Report, Lucent Technologies Inc., 1997

[RhSo92] J.K.Rho and F.Somenzi, "Inductive Verification of Iterative Systems", *Proc. of DAC'92*, 1992, pp.628-633

[RhSo93] J.K.Rho and F.Somenzi, "Automatic Generation of Network Invariants for the Verification of Iterative Sequential Systems", *Proc. of CAV'93, LNCS663*, 1993, pp.123-137

[Ritt97] J.Ritter, *Flexible Visualisierung von Entwurfsdaten in CADIC*, Diplomarbeit, Fachbereich 14 - Informatik, Universität des Saarlandes, Germany, 1997

[Ross90] L.Rossen, "Ruby Algebra", *in Designing Correct Circuits, G.Jones and M.Sheeran (Eds.)*, 1990, pp.297-312

[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

[Shee88] M.Sheeran, "Retiming and Slowdown in Ruby", *The Fusion of Hardware Design and Verification, G.J.Milne(Ed.), North-Holland*, 1988, pp.245-259

[ShRa93] R.Sharp, O.Rasmussen, "Transformational Rewriting with Ruby", *Proc. of Computer Hardware Description Languages and their Applications (CHDL'93)*, 1993, pp.243-260

[ShRa95] R.Sharp, O.Rasmussen, "The T-Ruby Design System", *Proc. of Computer Hardware Description Languages and their Applications (CHDL'95)*, 1995, pp.587-596

[VeCM90] D.Verkest, L.Claesen and H.D.Man, "Correctness Proofs of Parameterized Hardware Modules in The Cathedral-II Synthesis Environment", *Proc. of EDAC'90*, 1990, pp.62-66