Functional Extension of Symbolic Model Checking

Thomas Filkorn
Siemens AG, Corporate Laboratories for Information Technology, ZFE IS INF 2
Otto-Hahn-Ring 6, D-8000 Munich 83, F.R.G.

Abstract

Burch, Clarke, McMillan, Dill and Hwang describe in [4] a symbolic model checking procedure for μ-calculus formulas. The algorithm is based on the representation of relations by binary decision diagrams (BDDs) [1]. In the area of synchronous digital circuits a functional instead of a relational representation results in more compact BDDs. This is the reason for extending the μ-calculus and the symbolic model checking procedure with functions.

1 Introduction

Errors in the design phase of systems, like communication protocols or digital circuits, are a major reason for unexpected delays, costs and lack of reliability. Verification is today performed by techniques based on simulation and testing. However these are far away from being exhaustive and hence correctness can not be guaranteed. This has stimulated interest in formal verification techniques which can guarantee correctness with respect to the verified properties.

The behaviour of many systems can be modeled adequately as finite-state systems and verification of them can often be performed automatically by examining their state-graphs. Based on this a number of methods, e.g., testing for various equivalences or model checking on finite-state systems, have been proposed and are further researched. Since all of the methods rely on an explicit representation of the state-graph in a table or something similar they are limited to systems with at most approximately 10^6 states. A principal,problem in the application to larger realistic examples is the so called *state explosion problem*, that is the number of states grows exponentially with the number of components in the system. One approach to avoid the state explosion problem is to represent the state space symbolically.

One kind of symbolic representation are binary decision diagrams (BDDs) [1]. BDDs are a canonical representation of boolean formulas by directed acyclic graphs and Bryant described in [1] efficient algorithms for manipulating them. Based on BDDs Burch, Clarke et al. described in [3] a model checking algorithm for a branching time temporal logic, CTL, and generalized the idea in [4] to a powerful version of the μ-calculus. Their model checking algorithm is restricted to relations. From a theoretical point of view this is not really a restriction, since every function $f : A \rightarrow B$ can be seen as a relation $r_f \subseteq A \times B$. But, for a compact representation, BDDs exploit regularities in the structure of a function and often these regularities can not be exploited by BDDs in the corresponding relation. From our experiences in the area of digital circuits the BDD representations for the functions of circuits are in general more compact than the representation of the corresponding relations.

This is the reason why I extended the μ-calculus presented by Burch, Clarke et al. in [4] with functions. Section 2 describes the extended μ-calculus and in Section 3 the BDD based symbolic evaluation algorithm is presented. Section 4 will give results about the practical examples, including the simple pipeline design from [3], where the functional BDD representation is more efficient than the BDD representation of the model by relations.

2 The Extended μ-Calculus

The semantic model of the extended μ-calculus formulas will be vectors and functions over vectors. Since functions are normally defined only for vectors of a certain length, we have to introduce simple typing in the calculus in order to interpret the formulas in the semantic model. The set of basic types is Γ. Let X be a set of variable symbols, where each $x \in X$ has a basic type $\tau \in \Gamma$. F is a set of function symbols and every $f \in F$ has a type $\tau_1 \times ... \times \tau_n \rightarrow \tau_{n+1}$ with basic types $\tau_i \in \Gamma$. There are two syntactical categories, individual terms and functional terms, both typed and inductively defined as follows.

individual terms

x	where $x \in X$. The type of this individual term is the type of the variable symbol x.
$g(t_1, ..., t_n)$	where t_i are individual terms with types τ_i and g must be a functional term with type $\tau_1 \times ... \times \tau_n \rightarrow \tau$. The type of this individual term is τ.
$\forall x\ t$	where t is an individual term with type τ and $x \in X$. The resulting type is τ.

functional terms

f	where $f \in F$. The type of this functional term is the type of the functional symbol f.
$\lambda x_1, ..., x_n\ t$	where t is an individual term and $x_1, ..., x_n \in X$. The type of this functional term is $\tau_1 \times ... \times \tau_n \rightarrow \tau$, when τ_i is the type of x_i and τ the type of t.
rec$f.g$	where g is a functional term and $f \in F$, both with type $\tau_1 \times ... \times \tau_n \rightarrow \tau$, which is also the resulting type.

The individual and functional terms are interpreted with respect to a semantic structure $\mathcal{M} = (D, I_\Gamma, I_X, I_F)$. The *domain* D is a finite, non-empty, totally ordered set. I_Γ gives an interpretation of the basic types $\tau \in \Gamma$ as sets of vectors over D, $I_\Gamma(\tau) = D^{n_\tau}$. Individual variables $x \in X$ with basic type τ are mapped by the variable interpretation I_X to vectors over D, $I_X(x) \in I_\Gamma(\tau) = D^{n_\tau}$. In the same way function symbols $f \in F$ are interpreted by the functional variable interpretation I_F as functions over D-vectors. Let $\tau_1 \times ... \times \tau_n \rightarrow \tau$ be the type of f, then $I_F(f) \in (I_\Gamma(\tau_1) \times ... \times I_\Gamma(\tau_n) \rightarrow I_\Gamma(\tau))$.

The semantic interpretation $I_{(I_X, I_F)}$ for a semantic structure (D, I_Γ, I_X, I_F) maps individual terms t to vectors over D, $I_{(I_X,I_F)}(t) \in D^{n_t}$, and functional terms g to functions over D-vectors, $I_{(I_X,I_F)}(g) \in (D^{n_1} \times ... \times D^{n_k} \rightarrow D^{n_{k+1}})$. $I_{(I_X,I_F)}$ is inductively defined on the syntactic structure of individual and functional terms. In the following x is a variable, f a functional symbol, g a functional term, and $t_1, ..., t_n, t$ are individual terms. $\tau(x_i)$ is the type of a variable symbol x_i. The definition of $I_{(I_X,I_F)}$ on individual terms is given by the following equations:

$$\begin{aligned}
I_{(I_X,I_F)}(x) &= I_X(x) \\
I_{(I_X,I_F)}(g(t_1, ..., t_n)) &= I_{(I_X,I_F)}(g)(I_{(I_X,I_F)}(t_1), ..., I_{(I_X,I_F)}(t_n)) \\
I_{(I_X,I_F)}(\forall x\ t) &= min(\{I_{(I_X\langle x \leftarrow e\rangle, I_F)}(t) \mid e \in I_\Gamma(x)\})
\end{aligned}$$

min for a set of vectors over D is defined as a vector, in which each component is the minimal value of all the values occuring in the corresponding component of all vectors in the set. The minimal value is determined with respect to the total ordering on D. The interpretation of functional terms is also defined equationally:

$$\begin{array}{lcl}
I_{(I_X,I_F)}(f) & = & I_F(f) \\
I_{(I_X,I_F)}(\lambda x_1,...,x_n\ t) & = & h : I_\Gamma(\tau(x_1)) \times ... \times I_\Gamma(\tau(x_n)) \rightarrow I_\Gamma(\tau(t)) \\
 & & h(e_1,...,e_n) \overset{\mathrm{def}}{=} I_{(I_{X(x_1 \leftarrow e_1,...,x_n \leftarrow e_n)},I_F)}(t) \\
 & & e_i \in I_\Gamma(\tau(x_i)) \\
I_{(I_X,I_F)}(\mathrm{rec} f.g) & = & \mathrm{lfp}\ h \in I'_F(f)\ .\ \ I_{(I_X,I_{F(f \leftarrow h)})}(g)
\end{array}$$

$I'_F(f)$ stands as an abreviaton for $I_\Gamma(\tau_1) \times ... \times I_\Gamma(\tau_n) \rightarrow I_\Gamma(\tau)$ when f has the type $\tau_1 \times ... \times \tau_n \rightarrow \tau$. This are all possible functions to which the function symbol f can be mapped by a semantic interpretation. lfp $h.g$ denotes the least fixpoint of the functional g with respect to the partial ordering $\sqsubseteq$ on functions, defined in the following. On the domain D a partial order $\sqsubseteq$ is defined by: $a \sqsubseteq b$ iff $a = b$ or $a = \perp$. $\perp$ denotes the minimal value of D with respect to the total ordering on D. This extends to vectors of D by: $\langle a_1,...,a_n \rangle \sqsubseteq \langle b_1,...,b_n \rangle$ iff $\forall i : a_i \sqsubseteq b_i$. The partial order $\sqsubseteq$ can further be extended to functions $f_1, f_2 \in (D^n \rightarrow D^m)$ in the usual way: $f_1 \sqsubseteq f_2$ iff $\forall x \in D^n : f_1(x) \sqsubseteq f_2(x)$. A functional g is monotone, iff $f_1 \sqsubseteq f_2$ implies $g(f_1) \sqsubseteq g(f_2)$. A least fixpoint need not exist for every functional, but for monotone functionals over a finite domain it exists and is uniquely defined. So $I_{(I_X,I_F)}$ is only well defined for functional terms rec$f.g$ where g is a monotone functional.

In this paragraph I want to outline briefly how the μ-calculus used by Burch, Clarke et al. in [4] is contained in the extended calculus. For this I assume the boolean domain, $D = \{0,1\}$, with the ordering $0 < 1$, thus 0 serving as the bottom element $\perp$. Any relation $r \subseteq D^n$ can be represented by its characteristic function $f_r : D^n \rightarrow D$ with $f(x) = 1 \Leftrightarrow x \in r$. For characteristic functions f_r the ordering $\sqsubseteq$ is exactly the set inclusion ordering on the corresponding relations r and so the rec operator is identical to the μ operator. From the previous it is clear that the relational terms defined in Section 3 of [4] are a subset of the functional terms used in the calculus here. Also the formulas of [4] are special cases of the individual terms described here, if the boolean operators $\vee, \neg, =$ are available with their usual interpretation in I_F.

By using a finite domain D and the ordering $\sqsubseteq$ we have a general calculus in which also e.g., 3-valued logic or recursively defined functions can be expressed directly. However the examples of Section 4 will only use the boolean domain $D = \{0,1\}$.

3 Symbolic Evaluation

Evaluation of a term t means computing the semantic interpretation $I_{(I_X,I_F)}(t)$ with respect to a semantic structure (D, I_Γ, I_X, I_F). An explicit representation of functions $g : D^{n_1+\cdots+n_k} \rightarrow D^m$ by tables, would implicate the state explosion problem, as mentioned in the introduction. To avoid this problem BDDs are used here as a symbolical representation of functions.

In [1] Bryant described binary decision diagrams (BDDs) as a normal form representation for boolean functions and efficient algorithms for manipulating them. BDDs are directed acyclic graphs with internal nodes labeled by variables $x_1,...,x_n$ and encode the truth table of a boolean function by exploiting some regularities in the function. For a given variable ordering a boolean function has a unique BDD. In most cases the ordering of the variables is very critical for the size of the BDD. For certain boolean functions (e.g., integer multiplication [1]) the size of a BDD grows exponentially in the number of variables for every variable ordering, which is not surprising since the NP-complete satisfiability problem can be solved with BDDs. However from our experience the sizes of the BDDs representing boolean functions realized by digital circuits are small in most cases. The extension of BDDs to functions $f : D^n \rightarrow D^m$, where D is a finite set is straigthforward.

The symbolic evaluation algorithm described in the next part is based on a few operations on BDDs. *BDD_var* maps variable symbols $x \in X$ to vectors of BDD variables. Since BDDs are normal forms for functions the equivalecnce check *BDD_equal* is a trivial operation. *BDD_forall* gets two arguments, a set of BDD variables and a BDD-vector, and evaluates a BDD-vector according to the semantic given in Section 2. The basic operation during the evaluation process is *BDD_compose*. Given the BDD-vectors for $g(x_1,...,x_n), f_1,...,f_n$ it computes the BDD-vector of the composite function $g|_{(x_1=f_1,...,x_n=f_n)}$. The apply operation, as described in [1], can be seen as a special case of function composition.

Based on BDDs, symbolic evaluation of individual terms and functional terms is performed by the routines *eval_it* and *eval_ft* as defined in the Figure below. The definitions of the semantic interpretation $I_{(I_X,I_F)}$ from Section 2 are directly computed, but for all possible variable interpretations instead of only a specific one.

Hence, the result of the routine *eval_it* for an individual term t is not a value from D^n, but a BDD-vector t_{BDD} with variables from X. For a specific variable interpretation I_X the interpretation $I_{(I_X,I_F)}(t)$ can be obtained from the BDD-vector t_{BDD} by substituting all the BDD variables in t_{BDD} with their values according to I_X, resulting in a vector over D. $I_{(I_X,I_F)}(t) = t_{BDD}|_{x_i=I_X(x_i)}$. In the first case of *eval_it* a variable is just mapped to its corresponding vector of BDD variables. In the two other cases the parts of the individual term are evaluated first and afterwards the BDD operation, realizing the semantic (see Section 2), is applied.

```
eval_it($x, I_F$)                 = BDD_var($x$)
eval_it($f(t_1,...,t_n), I_F$)    = BDD_compose(eval_ft($f, I_F$),
                                         eval_it($t_1, I_F$), ..., eval_it($t_n, I_F$))
eval_it($\forall x\ t, I_F$)      = BDD_forall(BDD_var($x$), eval_it($t, I_F$))
```

The result of the routine *eval_ft* for a functional term g is a BDD-vector g_{BDD} and a vector of variables, indicating that these variables are serving as placeholders for the functions arguments. Again the interpretation $I_{(I_X,I_F)}(g)$ for a variable interpretation I_X is obtained from g_{BDD} by substituting all variables not marked as placeholders by their values, according to I_X. The resulting BDD-vector, containinig only placeholder variables, is a representation for the function $I_{(I_X,I_F)}(g)$. When g is a function symbol its BDD representation is obtained by a simple table lookup in I_F. In the second case the individual term t is evaluated first, resulting in a BDD t_{BDD} with variables of X. For the lambda abstraction it is sufficient to mark the variables $x_1,...,x_n$ as placeholders without affecting the BDD-vector t_{BDD}. The least fixpoint rec$f.g$ of a monotone functional g is calculated by the standard fixpoint iteration, starting with the bottom element $\vec{\bot}$. $\vec{\bot}$ is a vector of $\bot$'s, when $\bot$ is the minimal element of the domain D, with respect to the total ordering on D. The next function h_{i+1} of the iteration is calculated from h_i and g by evaluating the functional term g with the interpretation of the function symbol f set to h_i. Monotonicity of g and finiteness of D guarantees termination of the iteration with the least fixpoint lfp $f.g = h_{fp+1} = h_{fp}$.

```
eval_ft($f, I_F$)                    = $I_F(f)$
eval_ft($\lambda x_1,...,x_n\ t, I_F$) = $\langle$eval_it($t, I_F$), $\langle$BDD_var($x_1$), ..., BDD_var($x_n$)$\rangle\rangle$
eval_ft(rec$f.g, I_F$)               = $h := \vec{\bot}$;
                                       do
                                           $h_{old} := h$;
                                           $h :=$ eval_ft($g, I_F\langle f \leftarrow h_{old}\rangle$);
                                       until BDD_equal($h, h_{old}$);
                                       return $h$
```

4 Empirical Results

Using BDDs is only efficient in a heuristic sense, and so it is difficult to give estimates for the sizes of the BDDs. Therefore empirical results from practical circuits are needed in order to evaluate the method and to compare it with other approaches. For this reason two examples of synchronous digital circuits, reported previously in the literature, were considered: The MinMax circuit [8], which is a small signal processor proposed by IMEC as a benchmark for formal system design methods, and the simple pipeline used by Burch, Clarke et al. in [3].

4.1 MinMax Circuit

This Section shows how equivalence of functionally represented automata can be expressed as a term in the extended μ-calculus. So a symbolic comparison of the automata can be performed by evaluating that term with the symbolic evaluation algorithm of Section 3. Specification and implementation of synchronous digital circuits like the MinMax example can be modeled by finite-state systems, more precisely Mealy automata. A Mealy automaton is a tuple $(S, \Sigma, \Delta, \delta, \lambda, r)$, where S is the set of states, Σ the input alphabet, Δ the output alphabet and $r \in S$ the initial state. The behaviour of a Mealy automaton is defined by the transition function $\delta : S \times \Sigma \to S$ and the output function $\lambda : S \times \Sigma \to \Delta$. In the case of digital circuits the domain of the states, inputs, and outputs are bitvectors.

Behavioural equivalence of two Mealy automata $\mathcal{M}_{spec}$ and $\mathcal{M}_{impl}$ can be defined with respect to their initial states: $r_{spec} \approx r_{impl}$. Two states are behavioural equivalent iff for every input sequence the generated output sequences are equal. The relation $\approx \subseteq S_{spec} \times S_{impl}$ of behavioural equivalent states is defined inductively as the largest relation with the following property:

$$s_1 \approx s_2 \text{ iff } \forall \sigma \in \Sigma : \ (\lambda_{spec}(s_1, \sigma) = \lambda_{impl}(s_2, \sigma) \ \text{ and } \ \delta_{spec}(s_1, \sigma) \approx \delta_{impl}(s_2, \sigma)).$$

The largest fixpoint $\nu f.g(f)$ of an recursive definition, as above, can also be expressed by a least fixpoint: $\neg \mu f. \neg g(\neg f)$. With this simple syntactic transformation the above equivalence definition for Mealy automata can be expressed directly in our calculus by the individual term $\neg \not\approx (r_{spec}, r_{impl})$, where $\not\approx$ stands as an abreviation for the following functional term:

$$rec \not\approx . \ \lambda x_1, x_2 \neg \forall \sigma (\lambda_{spec}(s_1, \sigma) = \lambda_{impl}(s_2, \sigma) \wedge \neg \not\approx (\delta_{spec}(s_1, \sigma), \delta_{impl}(s_2, \sigma)))$$

This formula was evaluated for different bitsizes of the MinMax circuit and the results are listed in Table 1. The first five columns give some characteristics about the circuit. *width* is the width of the data path, *states* is an approximation for the number of possible states, these are those reachable from initial states. The execution times, measured in minutes, of column *time* have been obtained on a Sun 3/60 workstation with the described method, implemented in a Prolog extended by unification in finite algebras [6]. As a comparison, the times in column *time*[7] are the ones Berthet, Coudert and Madre obtained with their approach for BDD-based automata equivalence checking [5] on a DPX5000 mini computer (about twice as fast as a VAX/780).

width	*states*	*time*	*time* [7]
8	$2.8 * 10^6$	3	1.5
9	$2.5 * 10^7$	4	5
10	$1.8 * 10^8$	6	23
16	$4.4 * 10^{13}$	20	
32	$9.3 * 10^{27}$	109	

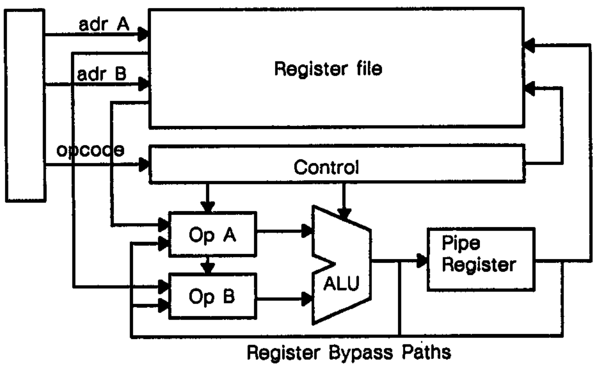
Table 1: MinMax Empirical Results

The time for evaluating the above term, expressing automata equivalence, grows polynomially with the bitsize N, about $N^{2.5}$, whereas execution times in [7] seem to grow exponentially, about 4^N.

4.2 Synchronous Pipeline

This section gives an example for the verification of a simple pipeline design. The circuit was first described in [3] and also verified with the symbolic model checker of [3]. However Burch, Clarke et al. used relations for representing the circuit. In contrast to them in the following approach functions are used for the same task.

The pipeline performs simple arithmetic and logical operations on a register file, according to an instruction register. The instruction register contains the source register addresses, the destination register address, the operation code, and a special stall bit. The operations are performed in three stages. In the first step the operands are read from the register file. In the next step an ALU operation is performed and in the third step the result is written back to the register file. Since the result of an operation can be used immediately in the next step as an operand there are register bypass paths. If the stall bit in the instruction register is set a "no-operation" is propagated through the pipe. A simple block diagram of the pipeline is shown below.



The pipeline can be modeled as a finite-state system. The state is composed of the register file state, the pipe registers and the state of the control part. A transition function δ gives for each state and input, the state in the next step. From Table 2 the great difference in the representation by a transition relation, as done in [3], and the functional representation is evident.

A specification of the pipeline can be obtained by taking into account the latency of the pipe which is three clock cycles. The result of an operation will not affect the register file until three cycles in the future, and the inputs of the operation should correspond to the state of the register file two cycles in the future. *reg* is a simple projection, giving the register file part of a pipeline state. The function *select*, selects the i-th part of an array of values. $reg_2(i)$, $reg_3(i)$ are used as abreviations for the value of the i-th register two, three steps in the future.

$$\begin{aligned} reg_2(i) &= select(reg(\delta(\delta(s,\sigma_1),\sigma_2),i) \\ reg_3(i) &= select(reg(\delta(\delta(\delta(s,\sigma_1),\sigma_2),\sigma_3),i) \end{aligned}$$

The function $aluop(op,x,y)$ gives the result of the operation op applied on the arguments x and y. With this abreviations the whole specification can be expressed by the following term in our calculus.

$$\begin{array}{lll} \forall s,\sigma_1,\sigma_2,\sigma_3,x & \neg stall & \Rightarrow \; reg_3(c) = aluop(op, reg_2(a), reg_2(b)) \\ & \wedge & \\ & stall \vee x \neq c & \Rightarrow \; reg_3(x) = reg_2(x) \end{array}$$

The above formula was evaluated for different versions of the pipeline. The pipeline performed addition and exor operations and was used with various register widths and numbers of registers. The first column *width* gives the width of the registers in bits. In the following columns the results for register files with 4, 8, and 16 registers are listed. *BDD* gives the number of nodes in the BDD-vector for representing the transition function δ. The execution times, measured in seconds, of column *time* have been obtained on a Sun 3/60 workstation with the described method, implemented in a Prolog extended by unification in finite algebras [6]. As a comparison, the columns *BDD*[3] and *time*[3] are the results Burch, Clarke et al. obtained with their approach for the same task also on a Sun 3.

width	4 registers				8 registers		16 registers	
	BDD	*BDD* [3]	*time*	*time* [3]	*BDD*	*time*	*BDD*	*time*
2	161	18429	21	188	355	141	757	1178
4	329	53924	66	1706	715	433	1517	
8	665		308		1435	1683	3037	
16	1337		1905		2875		6077	
32	2681				5755		12157.0	

Table 2: Pipeline Empirical Results

In a new paper Burch, Clarke and Long described in [2] a method for representing circuits more efficiently in symbolic model checking. The key idea is to express a relation by conjunctions or disjunctions of relations, each with a compact BDD representation. Using this method the results have been improved considerably and seem to be in the same order of magnitude then the ones achieved with the functional extension.

The functional BDD representation of the pipelines behaviour is much more compact than the pure relational one, by a factor of more than 100. It grows linearly with the width and the number of registers. The relational representation grows only linearly with the width, but cubically with the number of registers from our experience. The time needed for verification grows between quadratically and cubically with the width and the number of registers. The most time consuming operation during verification consists in computing the function reg_3, which encodes all possible operand, i.e., register, combinations. Because in the BDD for reg_3 nearly no sharing is possible between two different combinations, the BDD grows at least quadratically with the number of registers.

References

[1] R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions Computer*, C-35(12):1035–1044, 1986.

[2] J.R. Burch, E.M. Clarke, and D.E. Long. Representing circuits more efficiently in symbolic model checking. In *ACM/IEEE Design Automation Conference*, 1991.

[3] J.R. Burch, E.M. Clarke, K.L. McMillan, and David L. Dill. Sequential circuit verification using symbolic model checking. In *ACM/IEEE Design Automation Conference*, 1990.

[4] J.R. Burch, E.M. Clarke, K.L. McMillan, David L. Dill, and L.J. Hwang. Symbolic model checking: 10^{20} states and beyond. In *LICS*, 1990.

[5] Olivier Coudert, Christian Berthet, and Jean-Christophe Madre. Verification of sequential machines using boolean functional vectors. In *IMEC-IFIP International Workshop on Applied Formal Methods For Correct VLSI Design*, 1989.

[6] Thomas Filkorn. Unifikation in endlichen Algebren und ihre Integration in Prolog. Master's thesis, Technical University Munich, 1988.

[7] Jean-Christophe Madre, Olivier Coudert, Michel Currat, Alain Debreil, and Christian Berthet. The formal verification chain at BULL. In *EURO ASIC 90*, 1990.

[8] Diederik Verkest, Luc Claesen, and Hugo De Man. Special benchmark session on formal system design. In *IMEC-IFIP International Workshop on Applied Formal Methods For Correct VLSI Design*, 1989.