**Proceedings of the IASTED International Conference**
**Applied Modelling and Simulation**
**September 1–3, 1999, Cairns, Australia**

# Solving Systems of Integer Linear Equations and Inequalities using Model Checking Techniques

Bernd Schütz
University of Hamburg, Dept. of Computer Science
Vogt-Kölln-Str. 30, 22527 Hamburg, Germany
Email: schuetz@informatik.uni-hamburg.de

*Abstract*—**During the last few years model checking has become a powerful tool in hardware verification. The number of states of the systems that can be verified using this technique have increased rapidly. This paper discusses the question whether model checking techniques practically can be used to solve systems of integer linear equations and inequalities.**

*Keywords*—**integer programming, model checking, arithmetics, modelling**

## I. Introduction

In the area of hardware verification model checking techniques have become well-established. Model checking is a technique that is based on building a finite model of the system to be verified and checking this model against a specification, e.g. some properties. The challenge in research was to find algorithms and data structures (e.g. to avoid the state explosion problem [14]) to manage the large search space. But model checkers today are able to handle systems with more than 100 state variables and more than $10^{20}$ reachable states [3]. Today, this technique is powerful enough that it is becoming widely used in industry to verify new developed designs [11].
Now it's an interesting question whether model checking

$$\sum_{j=1}^{n} a_{ij} \cdot X_j \quad \Diamond \quad b_i, \qquad i = 1, ..., m; \quad n \leq m,$$

$$a_{ij}, X_j, b_i \in \mathbb{Z}, \qquad \Diamond \in \{<, \leq, =, \geq, >\}$$

where all coefficients $a_{ij}$ and all $b_i$, the so called right hand side (RHS), are known integers and the $X_j, j \in \{1..n\}$, are feasible integers $(X_1, X_2, .., X_n)$

Fig. 1. Mathematical model of the problem

is powerful enough to be used in other application fields also. This paper discusses an application in the area of linear algebra or better in the area of integer linear programming (ILP): The solution of systems of linear integer equations and inequalities.
Integer linear programming is known to be NP-hard [9]. Most existing ILP-Solvers such as cutting plane e.g. [10] and branch and bound methods e.g. [7] are based on linear programming (LP) [8]. The difference of our problem definition (see Fig. 1) to ILP consists of the fact that we have involved no objective function.

## II. Basic Idea

In contrast to [12] where an OBDD (Ordered Binary Decision Diagram) based approach for 0-1 programming problems and [13], where an EVBDD (Edge Valued Binary Decision Diagram) based approach for ILP-tasks is presented, our approach does not rely on a special data structure.
The basic idea of our approach (see Fig. 2) is to trans-

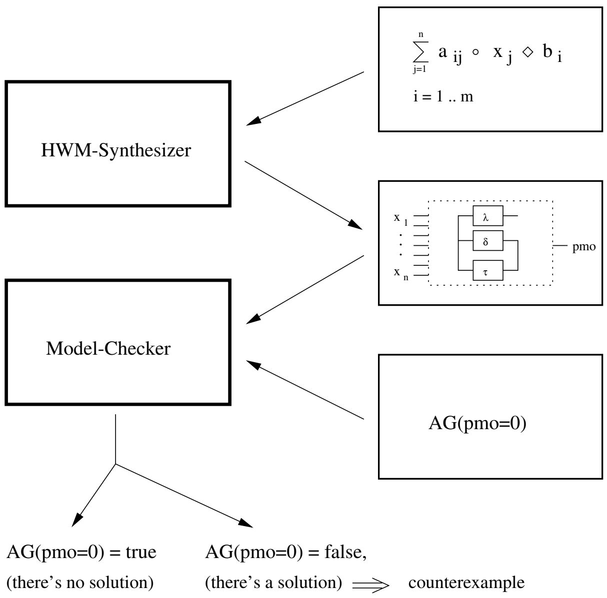


Fig. 2. System Overview

form a system of linear integer equalities and inequalities to some kind of hardware realization, which can be easily modeled as a finite state transition system. This task is done by the unit called HWM-Synthesizer shown in Figure 2. Now the model of the system has to be checked against a specification. The specification simply consists of one property, which denotes: *The system has no solution*.
Checking whether the model of the system is a model of

the specification is the task of a symbolic model checker, a well known tool in community of hardware verification. Either the model checker certifies the conformity of the model of the system and the specification, or it proves the non-conformity. Conformity means, that the system of linear integer equations and inequalities does not possess any solution. Non-conformity means, the system has at least one solution. The counterexample [4], supplied by the model checker, is the desired solution.

The advantage of our approach presented in this paper is in contrast to [12], [13], that we are not closely tightened to a underlying data structure such as OBDDs or EVBDDs. We have only to prepare a hardware model and the actually evaluation tool is a state of the art model checker. In this way we can take advantage of the progress in model checking techniques and the development of new more powerful model checkers by simply using a new version of the model checking tool.

## III. Model Checking

The model checking technique was introduced by Clarke and Emerson [5] and independently by Queille and Sifakis [17] and represents an approach to verification quite different from proof-based methods, where the validity of a logical formula for all models has to be proven. Model checking means to decide whether a given specific finite model satisfies a formula in a given logic.

Since [16] temporal logic became a wide spread formalism for reactive systems and a model in this semantics is usually referred to as a Kripke model. To decide whether a formula is satisfied by such a model the model checker has to traverse the entire Kripke structure. And this leads to the *state explosion problem* [14]. One method, that avoids this problem predominantly is the representation of the Kripke structure symbolically by characterizing the set of states and the transition relation by boolean formulas. Then Ordered Binary Decision Diagrams or OBDDs [2] provide an efficient tool for manipulating these boolean formulas. Verification tools that perform temporal logic model checking and which represent the Kripke structure symbolically (commonly by OBDDs) are generally referred to symbolic model checkers.

## IV. Number Representation

We represent numbers $X \in \mathbb{Z}$ in a binary representation with $k$ bits

$$X = \sum_{i=0}^{k} x_i \cdot 2^i, x_i \in \{0,1\} \qquad \text{(IV.1)}$$

using 2's complement for negative numbers. Because of the bit-serial realization of the hardware structure generated by our tool called *HWM-Synthesizer* (hardware-model-synthesizer, see the above sec. II), which realizes the system of equations and inequalities, we encode the numbers as vectors of bits $\overline{x} = (x_1, x_2, .., x_k), x_i \in \{0,1\}, i \in \{1..n\}$. The least significant bit comes first. Because of the underlying principle our HWM-Synthesizer constructs the hardware model of the system of equations and inequalities (bitwise serial computation, see also sec. VI) there is no limitation in the length $(k)$ of the numbers. In case of unequal length of the binary input vectors of the numbers $(X_1..X_n)$, they have to be aligned by sign extension.

## V. Automata Structure

The hardware-model-synthesizer (see Figure 2, HWM-Synthesizer) gets the mathematical description of the problem (see Figure 1) and generates a specialized hardware-architecture; or better a hardware-realization of the special system of linear integer equations and inequalities. The hardware description delivered by the HWM-Tool is generated in BLIF (Berkeley Logic Interchange Format) [18]. We made this choice, because the model checker we used, offered a BLIF entry. But other HDLs (Hardware Description Languages) like VHDL (Very High Speed Integrated Circuit Hardware Description Language) or Verilog are also possible.

The inputs of this synthesized unit are the variables $X_1..X_n$ (see Figure 1) in a binary representation. The output of the unit goes to logically true only and only if the input pattern (the actual values of the variables) satisfy the system. Figure 3 gives a closer look inside the unit. It consists of a number of at least $m$ automata. Each of these automata $A_i$, where $i \in \{1..m\}$, realizes one of the $m$ equations/inequalities, we can find in our mathematical model (Fig. 1). The automata $A_i$ are supplied with a subset $S_i \subset \{X_1..X_n\}$ of the variables, such that automaton $A_i$ generates the truth value of $f_i(S_i)$.

$$f_i(S_i) = \begin{cases} true & \sum_{j=1}^{n} a_{ij} \cdot X_j \quad \Diamond \quad b_i \\ & a_{ij}, X_j, b_i \in \mathbb{Z}, \\ & \Diamond \in \{<, \leq, =, \geq, >\} \\ & \\ false & \text{otherwise} \end{cases}$$

On other words: If the constraint $f_i, i \in \{1..m\}$ is satisfied by the sequence of inputs at time $t$, the output will become true at time $t$, too. These automata $A_{(1..m)}$ are connected via a logical AND, forming some kind of product machine. And the product machine output $(pmo)$ will show logically true at time $t$ if all sub-automata $A_{(1..m)}$ will generate at time $t$ simultaneously a high level at their outputs. This situation happens if the input sequence of the variables satisfies all $1..m$ restrictions simultaneously. This means that the actual input vectors of $X_1..X_n$ satisfy the system of linear integer equations and inequalities.

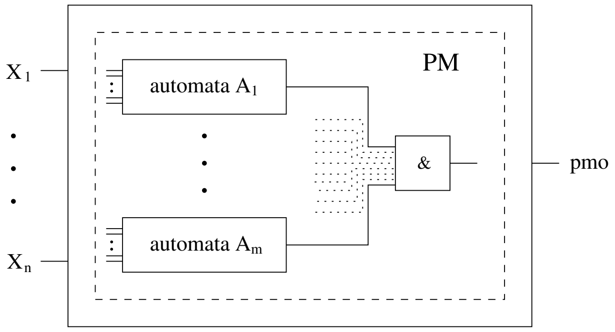


Fig. 3. Automata Structure

## A. Example

This section gives an example to demonstrate the basic procedure. Suppose we have the following system of inequalities :

$$5X_1 + 2X_2 \geq 11 \tag{V.1}$$

$$2X_1 < 7 \tag{V.2}$$

The first step is, to find a hardware realization for this system with one primary output. This is automatically done by our tool called HWM-Synthesizer.
The primary output *pmo* (see Figure 4) goes to logically one only and only if the input vectors satisfy the system. The inside of the box of Figure 4 shows a possible realization of our example described by the inequalities (V.1) and (V.2). As shown above each restriction is represented by an automaton. Automaton $A_1$, which realizes restriction (V.1) is shown by the upper dashed box of Fig. 4 and automaton $A_2$ representing inequality (V.2) is shown by the lower dashed box. The small boxes inside automaton $A_1$ and $A_2$ denote synchronous bit-serial sub-automata. These automata realize the required mathematical operations. For our example we have to synthesize multiplication with a constant, addition and integer comparison. Some overhead for synchronization aids and to ensure that no bit-vectors smaller than the required minimal number of bits are accepted has to be added (counter). The corresponding bits of the input vectors are read simultaneously.

This problem specific hardware description is fed in an automata like representation into an usual model checker, which is normally used for hardware verification. Generally the model checker checks a model of a (hardware) system against a specification to determine whether or not its behavior conforms to that of the specification. Exactly this task we have to assign to the model checker. In contrast to typical design verification tasks our specification simply consists of only one property, namely

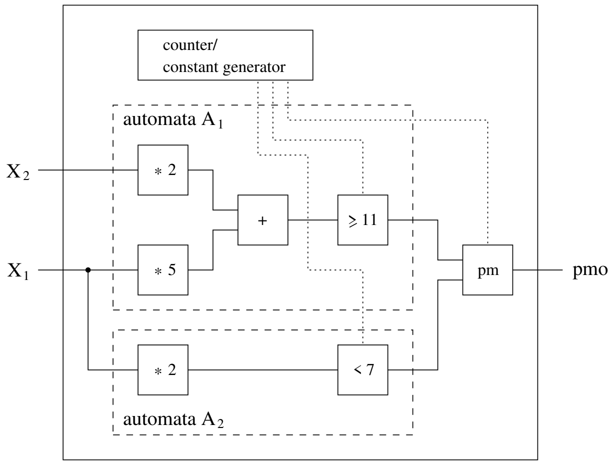


Fig. 4. Synthesized hardware

*AG(pmo=0)*[1].
If the model checker proves this property, there will exist no valid integer solution to our system of inequalities because of the construction of our model (see Fig. 4). If this property can not be proved, the model checker will stop with a counterexample. This counterexample is nothing else than a valid integer solution of the system.

## VI. Realized Operations

We started to develop the circuits needed for systems of linear integer equations and inequalities as shown in Fig. 1 in a first step only for numbers $X \in \mathbb{Z}^+$. This was done only for simplification.
So we realized the typical arithmetic operations used for systems of linear integer equations and inequalities, namely addition, subtraction, multiplication with a constant and the circuits for integer comparison with a constant, namely equal, greater, smaller, and so on.
The decision, to realize the automata as bit-serial circuits was not intended only by pure simplification, but to manage the circuit's complexity. This was done also regarding the characteristics/requirements of the verification tool. The smaller the state space the better is the performance of the tool. Generally bit-serial circuits are ones with the lowest complexity at hardware level. Considering these facts we designed bit-serial automata performing the desired mathematical operations.
Because of the bit-serial realization of the units their inputs $X_1..X_n$ must be provided as bit-serial input vectors $(\overline{x_1}..\overline{x_n})$. The corresponding bits of the input vectors $\overline{x_1}..\overline{x_n}$ have to be supplied to and to be read by the automata simultaneously. This fact gives one reason

[1] *AGf* is a typical CTL or Computation Tree Logic formula [6] and consist of the path quantifier *A* (for all computation path), the operator *G* (globally) and a formula *f*. *AGf* is true in a state if for all path starting from that state *f* is always true.

that the implementation of our automata is not in every case completely independent of the word length of the operands. This happens for example, if bit $j$ of a variable $X$ is supplied at time $t$ (note: corresponding bits of all variables are supplied simultaneously), but is needed by an automaton at time $t + k$ (may be a multiplication unit that performs $2^k \cdot X$), To overcome this gap the automaton has to store $k$ bits of that variable $X$.

In the following subsections we will give some examples of our sub-automata implementing arithmetic operations.

### A. Multiply by 2

From computer arithmetics we know the multiplication of one operand by a factor of $2^n$ as simply $n$-shifts. In this

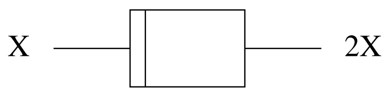


Fig. 5. Serial multiplication by 2

manner the circuit of Fig. 5 performs the multiplication of the bit-serial operand $X$ by a factor of 2. A multiplication by 16 for example will result in a register chain with four stages.

### B. Adder

Fig. 6 shows the implementation of the bit-serial adder. The architecture of this unit is standard stuff and can be found in literature everywhere [15]. A full-adder gets both

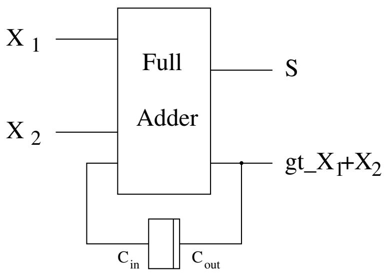


Fig. 6. Serial adder

operands $X_1, X_2$ and a carry $C_{in}$ as input and generates the sum $S$ and the carry $C_{out}$, which has to be stored in order to be supplied as $C_{in}$ ($C_{in} = 2\,C_{out}$) for the next pair of bits.
The actual $C_{out}$ additionally feeds a special output of the unit labeled in Fig. 6 as $gt\_X_1 + X_2$ because it has to be made available to other units, particularly compare-units. This output tells other units receiving the result $S$ that it is already known at time $t$ that the result $S$ will increase at time $t + 1$. And this increase is forced by the actual input sequence computed up to time $t$.

### C. Multiplication with a Constant

The architecture of the unit performing the multiplication with a constant is not invariant to the word length of it's operands. Fig. 7 shows the hardware synthesized for the special task $(9 \cdot X)$. This unit gets only one input, namely $X$. The second operand, the constant 9, which is already known at compile-time, will be hard-coded in the structure of the unit and this gives a direct dependency of structure and word length.
The other possibility of using an universal multiplier for two variables and to feed one operand of the unit by the input variable $X$ and the second operand by the integer constant via a constant generator is substantially worse because of the complexity of the multiplication of a variable with a variable. This will lead to an exponential growth of the resulting data structure of the verification tool (see section VIII).
Fig. 7 demonstrates a further particularity by it's second

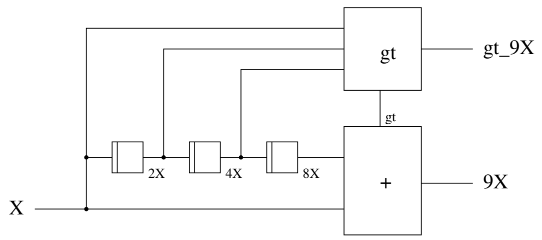


Fig. 7. Serial multiplier with a constant

primary output labeled $gt$. We need this output because possibly a compare unit gets the result of the multiplier and may decide its input sequence denoted by the actual output bits of the multiplier gives the desired result. But we know at this moment if one of the registers in the register chain of the multiplication unit generating $2X, 4X, 8X$ is set (in case of a positive operand) the result derived from the actual supplied input sequence will be greater than the primary output $9X$ tells us at this moment. This fact denotes that it is already known at time $t$, that the result of the multiplication unit will increase at time $(t + 1)..(t + n)$, with $n$ the length of register-chain. And this increase is forced exclusive by the actual input sequence computed up to time $t$. Finally the output of the subunit *gt* will rise if we will extend all inputs of the unit by sign extension and nevertheless the result will not remain constant but will increase in the future. Therefore the compare unit has to be supplied by the $gt$-output, too.

## D. Integer Comparison

This section shows exemplarily an unit, which tests a variable against an integer constant on equality. We realized also the operations for greater, smaller, greater-equal, etc. The integer constant to be compared against

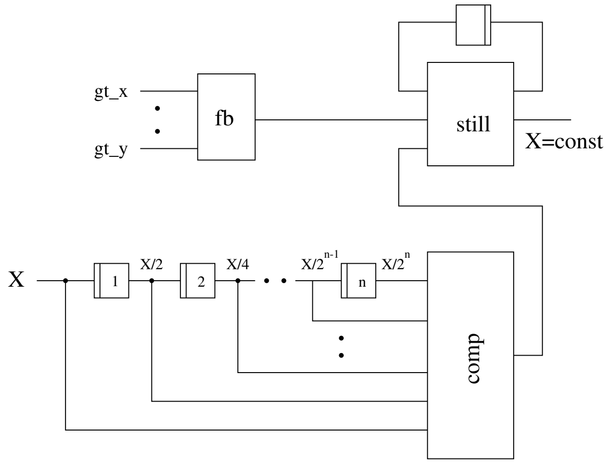


Fig. 8. Integer comparison

the input variable $X$ is already known at compile-time. As previously discussed in context of the multiplication with a constant in this automaton the integer constant is also hard-coded in the structure of the subunit labeled *comp* (see Fig. 8). This unit tests the recent $n$ bits ($n = \lceil log_2(const) \rceil$) against the desired value of the constant. The subunit *still* calculates the final result over the complete history of bits fed into the input $X$. The subunit *fb* is a particularity of the compare unit. We need it, because the input is usually not a primary one but comes from a network of adders and multipliers. And these units deliver not only a result but additional information (e.g. $gt\_add$, $gt\_mul$) about the known future of their results.

After first encouraging tests with the previous introduced automata we extended the circuits for numbers $X \in \mathbb{Z}$. Obviously this step increased the complexity of the units. And as a direct consequence the number of nodes of the resulting BDDs (the underlying data structure of the model checker) for most operations increased approximately by a factor of two. Especially the circuits for integer comparison became more complex.

## VII. Implementation

In the previous sections we showed exemplarily some realizations of the operations. The appropriate pictures and especially Fig. 3 suggest to the reader, that the automata and sub-automata are implemented completely separately. But this methodology will lead to unnecessary complex circuits. Especially in the context of systems of linear equations and inequalities there is a high probability of the multiple occurances of equivalent sub-structures. The example V-A demonstrates this fact by the term $2X_1$ of the restriction ( V.1). Figure 4 introduces for this operation an additional multiplier, but ignoring that the sub-unit performing $5X_1$ of automaton $A_1$, which represents restriction ( V.2), will already calculate this sub-term on its way calculating $5X_1$ (see Fig. 10).

Our implementation takes advantage of this fact. Every

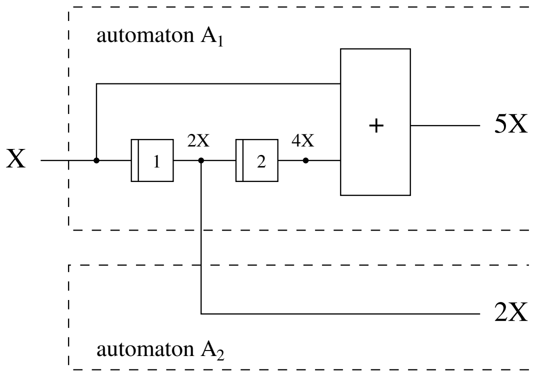


Fig. 10. Equivalent substructures

time an internal node of the automaton or sub-automaton shall be created while synthesizing the system, we will look for it in a special table. If the node already can be found, the task will be done. If not, we have to create the necessary hardware and the node is entered also under an unique name into that special table. To reduce the seek-time we defined a relation over the names of the nodes. This mechanism is quite similar to the unique tables introduced by [1] to make their BDD-packages much more efficient.

## VIII. Experimental Results

The Tabular of Fig. 9 shows some examples and demonstrates the development over the number of nodes of the resulting BDDs generated by the model checker. The items 1..5 of the tabular show the results of some systems of linear integer equalities and inequalities. Our example introduced in section V-A is the first item of the table and has got the name *p.lp*. The items 6..9 are the results of some of our native prime tests.

Regarding the tabular of Fig. 9 it should be considered, that the items 1..4 and 6..8 demonstrate some kind of worst case behavior. This fact is obvious regarding our native prime test. To verify that a number is a prime means to explore the whole state-space and you have to expand every node. The items 8 and 9 demonstrate this effect. *Prime.257* (item 8) tests against a prime and *prime.256* (item 9) tests not against a prime.

| | name | # variables | # of bits (word length) | # constraints | # nodes |
|---|---|---|---|---|---|
| 1 | p.lp | 2 | 4 | 2 | 672 |
| 2 | sk.lp | 4 | 4 | 2 | 2102 |
| 3 | b0.lp | 3 | 6 | 4 | 18774 |
| 4 | g1.lp | 9 | 12 | 15 | 196608 |
| 5 | g11.lp | 9 | 12 | 15 | 15283 |
| 6 | prime.11 | 2 | 4 | 1 | 131072 |
| 7 | prime.111 | 2 | 7 | 1 | 655360 |
| 8 | prime.257 | 2 | 9 | 1 | 2818048 |
| 9 | prime.256 | 2 | 9 | 1 | 70333 |

Fig. 9. Some examples and their number of BDD-nodes

Nevertheless it should be avoided to extend our mathematical model by the multiplication of a variable with a variable. The implementation of this operation leads to BDDs with an exponential growth in the length of the input numbers as Bryant already showed in 1986 [2]. And according to this fact a native prime test (e. g. $A \cdot B = 257$) was only computable for small numbers ($< 512$) in an acceptable time (see Fig. 9).

## IX. Conclusion

All operations required for systems of linear integer equations and inequalities (see Fig. 1) result in a manageable size of BDDs. Our presented approach demonstrates, that symbolic model checking techniques are also an appropriate tool for integer problems in linear algebra.
Further research will be done to analyze whether the efficiency of the model checking tool can be improved by supplying circuits, which can be represented by more compact OBDDs. We will analyze different architectures of machines alternative to that ones we choose, especially regarding their OBDD representations.

## Acknowledgments

Special thanks to J.W. Lohse at Siemens Corporate Technology, for his support and valuable contributions.

## References

[1] K. S. Brace, R. L. Rudell, and R. E. Bryant. Efficient Implementation of a BDD Package. In *Proceedings of the 27th ACM/IEEE Design Automation Conference*, pages 40–45, Orlando, FL, June 1990. ACM Press.

[2] R. E. Bryant. Graph-based Algorithms for Boolean Function Manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, August 1986.

[3] J.R. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill, and L.J. Hwang. Symbolic Model Checking: $10^{20}$ States and Beyond. In *Fifth Annual IEEE Symposium on Logic in Computer Scienc*, pages 428–439, Philadelphia, Pennsylvania, June 1990. IEEE Computer Society Press.

[4] E. Clarke, O. Grumber, K. McMillan, and X. Zhao. Efficient Generation of Counterexamples and Witnesses in Symbolic Model Checking. In *SIGDA Publications on CD-ROM : DAC 95*, San Francisco, CA, June 1995. SIGDA.

[5] E. M. Clarke and E. A. Emerson. Synthesis of Synchronization Sekletons for Branching Time Temporal Logic. In Dexter Kozen, editor, *Lecture Notes in Computer Science*, volume 131, pages 52–71, Yorktown Heights, NY, May 1981. Springer-Verlag.

[6] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic Verification of Finite-state Concurrent Systems using Temporal Logic Specifications. *ACM Transactions on Programming Languages and Systems*, 8(2):244–263, 1986.

[7] R. Dakin. A Tree Search Algorithm for Mixed Integer Programming Problems. *The Computer Journal*, 8:250–255, 1965.

[8] G. B. Dantzig. *Linear Programming and Extensions*. Princeton University Press, Princeton, NJ, 1963.

[9] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. Freeman, San Francisco, 1979.

[10] R. E. Gomory. Solving Linear Programming Problems in Integers. In R.E. Bellman and Jr M. Hall, editors, *American Mathematical Society*, pages 211–216, 1960.

[11] R. Herrmann, F. Korf, and R. Kumar. Speeding Up ASIC Design Using Formal Verification. In *Design, Automation and Test in Europe*, pages 279–283, Paris, France, February 1998. IEEE Computer Society Press.

[12] S-W. Jeong and F. Somenzi. A New Algorithm for 0-1 Programming Based on Binary decision diagrams. In T. Sasao, editor, *Logic Synthesis and Optimization*, pages 145–165, Boston, MA, January 1993. Kluwer Academic Publishers.

[13] Y-T. Lai, T Pedram, and S.B. Vrudhula. EVBDD-based Algorithms for Linear Integer Programming, Spectral Transformation and Function Decomposition. *IEEE Trans on CAD*, 13(8):959–975, August 1994.

[14] K. L. McMillan. *Symbolic Model Checking: An Approach to the State Explosion Problem*. PhD thesis, Carnegie Mellon University, Pittsburgh, Pennsylvania, USA, May 1992.

[15] Amos R. Omondi. *Computer Arithmetic Systems : Algorithms, Architecture and Implementations*. Prentice-Hall International (UK) Limited, Hemel Hempstead, Herfordshire, 1994.

[16] Amir Pnueli. Applications of Temporal Logic to the Specification and Verification of Reactive Systems: A Survey of Current Trends. In J. W. de Bakker, W. P. de Roever, and G. Rosenberg, editors, *Current Trends in Concurreny: Overviews and Tutorials*, volume 224, pages 510–584. Springer-Verlag, 1986. Lecture Notes in Computer Science.

[17] J. P. Queille and J. Sifakis. Specification and Verification of Concurrent Systems in CESAR. In *Proceedings of the 5th International Symposium in Programming*, volume 137, pages 337–351, Turin, Italy, April 1982. Springer-Verlag. Lecture Notes in Computer Science.

[18] E. M. Sentovich, K. J. Singh, L. Lavagno, C. Moon, R. Murgai, A. Saldanha, H. Savoj, P. R. Stephan, R. K. Brayton, and A. Sangiovanni-Vincentelli. *SIS: A System for Sequential Circuti Synthesis*. University of California, Berkeley, CA 94720, May 1992.