THEOREM PROVING TECHNIQUES FOR FORMAL SYSTEM VERIFICATION

Hasan Krad, Ph.D.
Department of Computer Science and Engineering, Qatar University
hkrad@qu.edu.qa

ABSTRACT

The complexity and functionality of Hardware and Software systems are persistently growing. Therefore, the probability of subtle faults existence in these systems is also increasing. Some of these faults may result in a devastating loss in terms of money and time. However, one main goal of designing those systems is to construct better and more reliable systems, regardless of the level of their complexity. Formal methods seems to be a promising approach to specify such systems and be automated to verify them. In this paper, we introduce and show how we can use some of those formal methods techniques, Propositional Logic (PL) and First Order Logic (FOL), in specifying and verifying the correctness of some properties related to such systems.

KEYWORDS

Formal verification, theorem proving, problem solving, formal methods.

1 INTRODUCTION

The history of automated deduction started with the idea of mechanizing human thought, which was first found in the writings of Leibniz who proposed two ambitious projects, the first one was a calculus of reason and the second one was a universal language. He developed fragments of Boolean algebra. Two centuries later, a rational calculus, in Leibniz' sense was constructed by George Boole. The fact that Boole's work had mechanized logic was recognized by the economist and logician Stanley Jevons, who constructed a working machine for verifying Boolean identities in 1869. In 1920, Emil Post had clearly formulated, in his doctoral dissertation, a meta-mathematical program and carried out a careful meta-mathematical investigation of what is now called the propositional calculus. However, the key work for automated deduction was that of Skolem. He carried out a systematic study of the problem of the existence of an interpretation which will satisfy a given formula of the predicate calculus. He introduced in 1920 what is now known as skolem functions, and a full treatment of them came out in 1928. His technique is the underlying basis of most computer implemented theorem provers. With a modern digital computer becoming available in the 1950's, it is not surprising that interest began to develop in automating deduction, and two mathematical theorem provers were implemented. The first one was the Logic machine of Newell, Shaw, and Simon, which is a computer program to prove theorems in the propositional logic. Their interest was to simulate the behavior of a human problem-solver attempting the same task. The other early theorem prover was the implementation in 1954 by Davis of a decision procedure for the arithmetic of addition that had been given by Presburger. The computer used was JohnNiac of the Institute for Advanced Study. It is a vacuum tube machine with cathode ray tube memory. What they were trying to prove was that the sum of two even numbers is even[1]. Formal methods are very useful and effective in specifying and verifying software and hardware systems. Some formal methods such as Communicating Sequential Processes[2], a Calculus of Communicating Systems[3], Statecharts[4], Temporal logic[5, 6, 7], and

I/O automata[8] focus on specifying the behavior of concurrent systems, while others such as Larch[9], Z[10], and VDM[11] focus on specifying the behavior of sequential systems. Other formal methods work such as LOTOS[12] and RAISE[13] combine two different methods, one for handling rich state spaces and the other one for handling complexity due to concurrency. It is through the specification, the system developers will be able to better understand these systems and uncover design ambiguities, flows, incompleteness, and inconsistencies. Furthermore, the specification is an important communication mechanism between the customers and the designers, between the designers and implementers, and between the implementers the testers[14].

Artificial Intelligence science goal, for example, is to study all aspects of intelligence by *computational modeling*. Some of the aspects that have been studied include mathematical reasoning, the ability to coordinate hand and eye to manipulate objects, the ability to hold a conversation in what is called a *natural language*, the ability to diagnose an illness and prescribe a cure for it, etc. However, whatever aspect of intelligence we try to model in a computer program, the same needs arise over and over again, namely the need to have knowledge about the domain, the need to reason with that knowledge, and the need for knowledge about how to direct the reasoning. Thus, mathematical reasoning is a convenient domain for studying intelligence, simply because the knowledge can be clearly defined and the goals are clear and unambiguous[15].

2 PROPOSITIONAL LOGIC

In propositional logic, what we are interested in is a *proposition, which* is a declarative sentence that could be either *true* or *false,* but not both. By using *logical connectives,* $\neg$ *,* $\vee$, $\wedge$,$\rightarrow$ *, and* $\Leftrightarrow$, we can build what is called *Well-Formed Formulas,* compound propositions, which can be defined in the propositional logic as follows[16]:

1- An atom is a formula.
2- If P is a formula, then $\neg$P is a formula.
3- If P and Q are formulas, then (P $\wedge$ Q), (P $\vee$ Q), (P $\rightarrow$ Q), and (P $\Leftrightarrow$ Q) are formulas.
4- All formulas generated by the above three rules are also formulas.

If a propositional formula, P, has the atoms $A_1,\ldots, A_n$ then an *Interpretation* of P, I, is an assignment of truth values to A_1, . . . , A_n in which every A_i is assigned either *T* (True) or *F* (False)*,* but not both. The formula P then would be *T* under the interpretation I if and only if P is evaluated to *T* in I which is then called a *model* of P; otherwise, P is said to be *F* in I.

2.1 Applications of the Propositional Logic

Suppose that the stock prices go down if the prime interest rate goes up. Suppose that also most people are unhappy when stock prices go down. Assume that the prime interest rate does go up. Let us show that we can conclude by using the propositional logic that most people are unhappy. If we let

P :≡ prime interest rate goes up.
Q :≡ stock prices go down.
U :≡ most people are unhappy.

In this example, there are four statements:
1- If the prime interest rate goes up, stock prices go down.
2- If stock prices go down, most people are unhappy.
3- The prime interest rate goes up.
4- Most people are unhappy.

let us symbolize these statements:

1- P $\rightarrow$ Q
2- Q $\rightarrow$ *U*

3- P
4- U

We will show now that U is a logical consequence of the formula $(P \rightarrow Q) \wedge (Q \rightarrow U) \wedge P$. In other words, $((P \rightarrow Q) \wedge (Q \rightarrow U) \wedge P) \rightarrow U$ should be *valid*.

$((P \rightarrow Q) \wedge (Q \rightarrow U) \wedge P) \rightarrow U$
$= \neg((P \rightarrow Q) \wedge (Q \rightarrow U) \wedge P) \vee U$
$= \neg((\neg P \vee Q) \wedge (\neg Q \vee U) \wedge P) \vee U$
$= \neg(\neg P \vee Q) \vee \neg(\neg Q \vee U) \vee \neg P \vee U$
$= (\neg (\neg P \vee Q) \vee \neg P) \vee (\neg(\neg Q \vee U) \vee U)$
$= \neg((\neg P \vee Q) \wedge P) \vee \neg((\neg Q \vee U) \wedge \neg U)$
$= \neg(Q \wedge P) \vee \neg(\neg Q \wedge \neg U)$
$= \neg Q \vee \neg P \vee Q \vee U$
$= \mathbf{T} \vee \neg P \vee U$
$= \mathbf{T}$

We can also prove that by contradiction as follows:

$(P \rightarrow Q) \wedge (Q \rightarrow U) \wedge P \wedge \neg U$
$= (\neg P \vee Q) \wedge (\neg Q \vee U) \wedge P \wedge \neg U$
$= ((\neg P \wedge \neg Q) \vee (\neg P \wedge U) \vee (Q \wedge \neg Q) \vee (Q \wedge U)) \wedge P \wedge \neg U$
$= ((\neg P \wedge \neg Q \wedge P) \vee (\neg P \wedge U \wedge P) \vee (Q \wedge U \wedge P)) \wedge \neg U$
$= Q \wedge U \wedge P \wedge \neg U$
$= \mathbf{F}$

Another example might be the need to do the following chemical reactions:

$$MgO + H_2 \rightarrow Mg + H_2O$$
$$C + O_2 \rightarrow CO_2$$
$$CO_2 + H_2O \rightarrow H_2CO_3$$

If we assume that we have some quantities of *MgO, H_2, O_2,* and *C*. Let us see if we could conclude that we can make H_2CO_3. We can represent this case as follows:

$F_1 : (MgO \wedge H_2) \rightarrow (Mg \wedge H_2O)$
$F_2 : (C \wedge O_2) \rightarrow CO_2$
$F_3 : (CO_2 \wedge H_2O) \rightarrow H_2CO_3$
$F_4 : MgO$
$F_5 : H_2$
$F_6 : O_2$
$F_7 : C$

We would like to show that H_2CO_3 is a *logical consequence* of $F_1 \wedge \ldots \wedge F_7$ by showing that $F_1 \wedge \ldots \wedge F_7 \wedge \neg H_2CO_3$ is *inconsistent*.

$F_1 \wedge \ldots \wedge F_7 \wedge \neg H_2CO_3$
$= (\neg MgO \vee \neg H_2 \vee Mg) \wedge (\neg MgO \vee \neg H_2 \vee H_2O)$
$\wedge (\neg C \vee \neg O_2 \vee CO_2) \wedge (\neg CO_2 \vee \neg H_2O \vee H_2CO_3)$
$\wedge MgO \wedge H_2 \wedge O_2 \wedge C \wedge \neg H_2CO_3$
$= (\neg MgO \vee \neg H_2 \vee Mg) \wedge (\neg MgO \vee \neg H_2 \vee H_2O)$
$\wedge MgO \wedge H_2 \wedge (\neg C \vee \neg O_2 \vee CO_2) \wedge C \wedge O_2$
$\wedge (\neg CO_2 \vee \neg H_2O \vee H_2CO_3) \wedge \neg H_2CO_3$
$= Mg \wedge H_2O \wedge MgO \wedge H_2 \wedge CO_2 \wedge C \wedge O_2$
$\wedge (\neg CO_2 \vee \neg H_2O) \wedge \neg H_2CO_3$
$= (\neg CO_2 \vee \neg H_2O) \wedge H_2O \wedge CO_2 \wedge Mg \wedge MgO \wedge H_2 \wedge C \wedge O_2 \wedge \neg H_2CO_3$
$= \mathbf{F} \wedge Mg \wedge MgO \wedge H_2 \wedge C \wedge O_2 \wedge \neg H_2CO_3$
$= \mathbf{F}$

3 THE FIRST ORDER LOGIC

Propositional logic is not suitable to express certain situations using the propositional logic formulas. Thus, the *first order logic,* which has more logical notation such as *terms, predicates,* and *quantifiers*, can be used to express a wide variety of statements[15,16,17]. To start with, an atom could be constructed using individual symbols or constants, variable symbols, function symbols, and predicate symbols.

In the first order logic, terms are defined recursively as follows:
1- A constant is a term.
2- A variable is a term.
3- If g is an n-place function symbol, and $x_1, \ldots, x_n$ are terms, then $g(x_1, \ldots x_n)$ is a term.
4- All terms that are generated using the above rules are terms.

Note that if Q is an n-place predicate symbol, and $x_1, \ldots, x_n$ are terms, then $Q(x_1, \ldots, x_n)$ is an atom.
In the first order logic we can use the same five logical connectives we used in the propositional logic in order to build up formulas. However, since we introduce variables, we use two more special symbols to characterize variables, namely, the *universal* symbol, ∀, and the *existential* symbol, ∃.

Let us consider the following statements:
1- Every rational number is a real number.
2- There exists a number that is prime.
3- For every number N, there exists a number M, such that N < M.

to express these statements using the first order logic, let us denote:
Y *is a rational number* by R(Y)
Y *is a real number* by *Real(Y)*
X *is a prime number* by P(X)
N *is less than M* by LESS (N,M)

then the above statements could be written as follows:
1- (∀Y) (R(Y) → Real(Y)
2- (∃X) P(X)
3- (∀N) (∃M) LESS (N , M)

In the first order logic, *Well-Formed Formulas* are defined recursively as follows:

1- An atom is a formula.
2- If N is a formula, then ¬N *is* a formula.
3- If N and M are formulas, then (N ∨ M), (N ∧ M), (N → M), and (N ⇔ M) are formulas.
4- If N is a formula and X is a free variable in N, then (∀X)N and (∃X)N are formulas.
5- All formulas generated using the above rules are formulas.

As we saw in the propositional logic, an interpretation is an assignment of truth values to atoms. In the first order logic, however, we have to do more than that since we introduced variables. So, an interpretation of a formula, G, in the first order logic would be done by defining a nonempty domain, D, and assign values to each of the constants, function symbols, and predicate symbols occurring in G as follows:

1- We assign an element of D to each constant in G.
2- We assign a mapping from D^n to D to each n-place function symbol in G.
3- We assign a mapping from D^n to {T, F} to each n-place predicate symbol in G.

Now after defining the interpretation, given an interpretation, a formula, G, will be interpreted according to the following rules:

1- If the truth values of formulas M and N are evaluated, then the truth values of the formulas ¬M, (M ∧ N), (M ∨ N), (M → N), and(M ⇔ N) are evaluated as in the propositional logic.
2- (∀x)N is evaluated to T if the truth value of N is evaluated to T for every x in D; otherwise, it is evaluated to F.
3- (∃x)N is evaluated to T if the truth value of N is T for at least one x in D; otherwise, it is evaluated to F.

4 THE RESOLUTION PRINCIPLE

The resolution principle was introduced by Robinson (1965). It is an important inference rule that can be applied to a certain type of WFF called clauses. The basic idea of the resolution principle is to check whether S ,the set of clauses, contains the empty clause o. If it contains it, then S is inconsistent, otherwise, we check to see whether o can be derived from S [18]. Thus, in order to have efficient theorem proving procedures, we must prevent large numbers of clauses from being generated. This issue directs us to the consideration of refinements of the resolution principle. There are many refinements and every one of them has its own merit. Let us consider three important ones, namely: The *Semantic Resolution*, the *Lock Resolution*, and the *Linear Resolution*.

4.1 Semantic Resolution

It was proposed by Slagle (1967). It unifies Robinson's *hyper resolution* (1965), Meltzer's *renamable resolution* (1966), and the *set of support* strategy of Wos, Robinson, and Carson (1965).
Let us consider the following unsatisfiable set S of clauses [16]:

(1)	$\neg P \vee \neg Q \vee R$
(2)	$P \vee R$
(3)	$Q \vee R$
(4)	$\neg R$

If we use the *level-saturation* method to prove the unsatisfiability of S, then 38 clauses will be generated. However, suppose that we can divide S into S^1 and S^2, and allow only clauses from different sets to be resolved with each other, then we will cut down the number of clauses generated. The question now is how we divide S into S^1 and S^2. In semantic resolution, we use an interpretation, I, to divide S, that is why it is called *semantic* resolution. Also to prevent some more clauses from being generated we may impose some other restrictions on the resolution, such as ordering of predicate symbols. To show this let us consider our last example with:

An interpretation: $I = \{\neg P, \neg Q, \neg R\}$
An ordering: $\alpha = P > Q > R$

Note that I divides S into:

$S_1 = \{(1), (4)\}$ which is satisfied by I.
$S_2 = \{(2), (3)\}$ which is falsified by I.

Note that (1) and (4) cannot be resolved since they are in the same set. Furthermore, (2) and (3) can still be resolved with (4) since (2) and (3) are in different set from that of (4). By imposing an ordering α, we can block such resolution by requiring that the resolved literal in the first clause contain the largest predicate symbol in that clause. Doing this, neither (2) nor (3) can be resolved with (4). Therefore, the proof then would be as follows:

S_1 :	(1)	$\neg P \vee \neg Q \vee R$		
	(4)	$\neg R$		
S_2 :	(2)	$P \vee R$		
	(3)	$Q \vee R$		
	(5)	$\neg Q \vee R$	2,1	*(added to S_1)*
	(6)	$\neg P \vee R$	3,1	*(added to S_1)*
	(7)	R	2,6	*(added to S_2)*
	(8)	R	3,5	*(added to S_2)*
	(9)	**F**	7,4	

If we look to our example and examine the ways R was generated, we note that R was generated using clauses (1), (2), *and* (3). The only difference is the order in which they were used. Since only one of them is needed in the proof, it is wasteful to generate both of them. To solve this redundancy problem, let us introduce the idea of *semantic* clash, where we can generate R directly from (1), (2), *and* (3) without having to go through the intermediate clause (5) or (6). The set {(1), (2), (3)} then will be called a clash. Let us define the notion of a semantic clash.

If we let I to be an interpretation and α to be an ordering of predicate symbols. A finite set of clauses $\{E_1, \dots, E_q, N\}$, $q \geq 1$, is called a *semantic clash* with respect to α and I, if and only *if* $E_1, \dots, E_q$ and N satisfy the following condition:

1- $E_1, \dots, E_q$ are false in I.
2- Let R = N. For each $i = 1, \dots, q$,
there is a resolvent R_{i+1} of R_i and E_i.
3- The literal in E_i, which is resolved

upon, contains the largest predicate symbol in E_i, i = 1.... ,q.

4- R_{q+1} is false in I.

R_q+1 is called a αI-resolvent of the αI-*clash* $\{E_1,\ldots, E_q, N\}$. Going back to our example we will have:

$N = \neg P \vee \neg Q \vee R$
$E_1 = P \vee R$
$E_2 = Q \vee R$
$I = \{\neg P, \neg Q, \neg R\}$
$\alpha = P > Q > R$

Then $\{E_1, E_2, N\}$ *is* a αI-*clash*, and R is the αI-resolvent of this clash. Note that R is false in I. An interesting thing about semantic resolution is that we can use any ordering and any interpretation. Furthermore, the αI-resolution is complete; that is from any unsatisfiable set of clauses, we can always derive the empty clause, O, by using some αI-resolution. One thing to be mentioned here is that *Hyper resolution* and the *set of support* strategy are special cases of the semantic resolution.

4.2 Lock Resolution

It was introduced by Boyer (1971). In this type of resolution, we use indices to order literals of clauses in S. Then resolution is only permitted on literals of lowest index in each clause. The resolvent literals inherit their indices from their parent clauses, and if there is more than one possible inherited index for a literal then we assign the lowest index to that literal [16].

Example:

(1) $_1N \vee {}_2M$
(2) $_3\neg N \vee {}_4M$

The resolvent of (1) and (2) is:

(3) $_2M \vee {}_4M$
(4) $_2M$

now since $_2M$ and $_4M$ are the same literal M and the index 2 < index 4, so the resolvent of (1) and (2) will be (4) and is called a lock resolvent of clause (1) and clause (2). Note that if the literals of clause (2) were indexed as follows:

(2′) $_4\neg N \vee {}_3M$

then $_3M$ is the literal to be resolved upon, therefore there is no lock resolvent for (1) and *(2′)*

Thus, if we let S be a set of clauses, where every literal in S is indexed with an integer, then a deduction from S is called a *lock deduction* if and only if every clause in the deduction is either a clause in S or a lock resolvent.

Example:

(1) $_1P \vee {}_2Q$
(2) $_3P \vee {}_4\neg Q$
(3) $_6\neg P \vee {}_5Q$
(4) $_8\neg P \vee {}_7\neg Q$

This way, we can generate only one lock resolvent, namely:

(5) $_6\neg P$ 3,4

then we can get:

(6) $_2Q$ 1,5

(7) $_4\neg Q$ 2,5

and finally:

(8) **F** 6,7

Note that only three lock resolvents were generated for this proof, while in the ordinary resolution, 35 resolvents would be generated using the level-saturation method. Moreover, the lock resolution, although it is complete, is not compatible with most other resolution strategies. For example, the combination of the lock resolution and the deletion strategy is not complete. It is also not compatible with the set-of-support strategy. However, it is a very efficient inference rule as Boyer has shown.

4.3 Linear Resolution

It was independently proposed by Loveland *(1970)* and Luckham *(1970),* and was later strengthened by Anderson and Bledsoe *(1970),* Yates et al. *(1970),* Reiter *(1971),* Loveland *(1972),* and Kowalski and Kuehner *(1971).* It is similar to a chain reasoning where we start with a clause, resolve it with another clause to get a resolvent which is resolved again with some other clause until the empty clause is generated.

Since efficiency is important in mechanical theorem proving, sometimes we may like to trade completeness for efficiency, because of the fact that there may be some refinements of resolution that are efficient and powerful enough to prove a large class of theorems, even though they are not complete. For instant, let us consider one incomplete, but efficient resolution, namely, *the input resolution.* It is a subcase of linear resolution, equivalent to *Unit Resolution* which was intensively used by Wos, Carson, and Robinson (1964), more efficient, and easier to implement, but it is incomplete as we mentioned. The only difference between it and the linear resolution is that every side clause in the input resolution has to be an input clause.

5 PARAMODULATION

It is an inference rule for the equality relation, and was proposed by Robinson and Wos (1969). As we saw previously, O can be deduced from an unsatisfiable set of clauses by resolution. It has been shown that by using both resolution and paramodulation, we can always deduce O from *E-unsatisfiable* set of clauses. Let us consider the following example [15,16]:

$$\begin{array}{ll} C_1 : & P(a) \\ \underline{C_2 :} & \underline{a=b} \\ C_3 : & P(b) \end{array}$$

So, the equality substitution rule can be defined as follows : If a clause C_1 contains a term t and if a unit clause C_2 *is* $t = s$, then we can infer a clause by substituting s for one occurrence of t in C_1, and the *paramodulation is* actually an extension of this rule. It can be applied to any pair of clauses. for example:

$$\begin{array}{lll} \textit{If} & C_1 : & L[t] \lor C'_1 \\ \underline{\textit{and}} & \underline{C_2 :} & \underline{t=s \lor C'} \\ \textit{Then} & C_3 : & L[s] \lor C'_1 \lor C'_2 \end{array}$$

$C3$ is called a *paramodudant* of C_l and C_2. Paramodulation can be defined on general clauses, and then we may need to make *instantiation* before paramodulation can be applied. If we consider the following clauses:

$$\begin{array}{ll} C_1 : & P(x) \lor Q(b) \\ C_2 : & a=b \lor R(b) \end{array}$$

We can see that C_1 does not contain the term a. However, if x is replaced by a, then we can obtain an instance of C_1, namely,

$$C'_1: \quad P(a) \lor Q(b)$$

now, from C'_1 and C_2, we can get a paramodulant:

$$C'_3 : \quad P(b) \lor Q(b) \lor R(b)$$

So, we can define the paramodulation formally as follows : Let C_1 and C_2 to be two clauses with no variable in common. If $C_1 = L[t] \lor C'_1$ and $C_2 = (r=s) \lor C'_2$ and if t and r have a most general unifier σ, then we can infer the clause:

$$C_3 : \quad L\sigma[s\sigma] \lor C'_1\sigma \lor C'_2\sigma$$

Example:

$$\begin{array}{l} C_1 : P(g(f(x))) \lor Q(x) \\ C_2 : f(g(b)) = a \lor R(g(c)) \\ \sigma \ : \{g(b)/x\} \\ C_3 : P(g(a)) \lor Q(g(b)) \lor R(g(c)) \end{array}$$

So, we apply paramodulation from C_2 into C_1, and $P(g(f(x)))$ and $f(g(b))=a$ are the literals paramodulated upon. The combination of resolution and paramodulation is complete for E-unsatisfiable set of clauses. Furthermore, many refinements of paramodulation are still complete when used with resolution, examples of refined paramodulation are:

1- *Hyper* paramodulation.
2- *Input* paramodulation.
3- *Linear* paramodulation.

6 CONCLUSION

With the fact that software and hardware systems are growing fast in both functionality and complexity, an increase commercial pressure to design better quality and more reliable software and hardware systems is also increasing. Formal methods proof is very effective in demonstrating a great success in both specification and verification of commercial and safety-critical software and hardware systems as well as verification of protocol standards. In this paper, we introduce and show how we can use some of those formal methods, using Propositional Logic (PL) and First Order Logic (FOL), in specifying and verifying the correctness of related system aspects.

7 REFERENCES

1. Davis, M.: The Prehistory and Early History of Automated Deduction. In: Automation of Reasoning: Classical Papers on Computational Logic, J. Siekmann and G. Wrightson, eds., Vol. 1, pp. 1-28, Springer, Berlin, (1983).

2. Hoare, C. A. R.: Communicating Sequential Processes, Prentice-Hall International, Englewood Cliffs, NJ, (1985).

3. Milner, A.: A Calculus of Communicating Systems, Lecture Notes in Computer Science, Vol. 92, Springer-Verlag, (1980).

4. Harel, D.: Statecharts: A Visual Formalism for Complex Systems, Sci. Comput. Program, 8, 231-274, (1987).

5. Pnueli, A.: A Temporal Logic of Concurrent Programs, Theor. Comput. Sci., 13, 45-60, (1981).

6. Manna, Z. and A. Pnueli: The Temporal Logic of Reactive and Concurrent Systems, Springer-Verlag, New York, (1991).

7. Lamport, L.: The Temporal Logic of Actions, ACM Trans. Program. Lang. Syst., 872-923, (1984).

8. Lynch N. and M. Tuttle: Hierarchical Correctness Proofs for Distributed Algorithms. Technical Report, MIT Laboratory for Computer Science, Cambridge, MA, (1987).

9. Guttag, J. and J. Horning, Larch: Languages and Tools for Formal Specification, Springer-Verlag, (1993).

10. Spivey, J. M.: Introducing Z: a Specification Language and its Formal Semantics. Cambridge University Press, New York, (1988).

11. Jones, C. B.: Systematic Software Development Using VDM. Prentice-Hall International, New York, (1986).

12. ISO, Information Systems Processing – Open Systems International – LOTOS, Technical Report, International Standards Organization DIS 8807, April, (1987).

13. Nielsen, M., K. Havelund, K. Wagner, and C. George: The RAISE Language, Method, and Tools. Formal Aspects Comput., 1, 85-114, (1989).

14. Clarke, E. M., J. M. Wing, Et Al.: Formal Methods: State of the Art and Future Directions. ACM Computing Surveys, Vol. 28, No. 4, December (1996).

15. Bundy, A.: The Computer Modeling of Mathematical Reasoning, Academic Press, (1983).

16. Chang, C-L. and Lee, R. C-T: Symbolic Logic and Mathematical Theorem Proving, Academic Press, (1973).

17. Loveland, D.W.: Automated Theorem Proving - A Logical Basis, Fundamental Studies in Computer Science. Volume 6, North Holland, (1978).

18. Nilsson, Nils J.: Principles of Artificial Intelligence, Tioga, Palo Alto, CA, (1980).