A Synthesized Framework for Formal Verification of Computing Systems

Nikola Bogunovic, Igor Grudenic, Edgar Pek
Faculty of Electrical Engineering and Computing, University of Zagreb,
Zagreb, 10000, Croatia

ABSTRACT

Design process of computing systems gradually evolved to a level that encompasses formal verification techniques. However, the integration of formal verification techniques into a methodical design procedure has many inherent miscomprehensions and problems. The paper explicates the discrepancy between the real system implementation and the abstracted model that is actually used in the formal verification procedure. Particular attention is paid to the seamless integration of all phases of the verification procedure that encompasses definition of the specification language and denotation and execution of conformance relation between the abstracted model and its intended behavior. The concealed obstacles are exposed, computationally expensive steps identified and possible improvements proposed.

Keywords: Formal methods, System engineering, System modeling, System verification.

1. INTRODUCTION

Analysis and design of computer-based systems is essentially still based on the process that encompasses diverse informal steps. More specifically, these steps are the formulation of requirements (what the system should and should not do), specification and analysis, design (how would the systems achieve its goals), coding (the actual programming), unit testing, integration and systems testing. Correctness is regularly validated by simulation and testing. However, such an approach can only prove the presence of bug, but never its absence.

Formal methods applied in developing computer-based systems are mathematical procedures for describing and verifying system properties and its behavior. Hence, formal verification is the process of ensuring that formal model of the design (*Imp* - implementation) satisfies a formal specification (*Spec*) with mathematical certainty, or plainly that an implementation conforms to specification. The possible conformance relations are equivalencies, various simulation relations, (logical) satisfaction, (logical) implication, refinement, etc. Formal verification guarantees beyond any doubt a correct relation between mathematical objects *Imp* and *Spec*. However, it does not establish the correctness of a real implementation abstracted by *Imp*.

In this paper we will formally define the abstracted model of the examined computer-based system (*Imp*) by adopting state machine descriptions from the automata theory. The concept will be restricted to finite (but extensive) state systems. Since real implementation procedures differ from this formal description, a mapping from some design (programming) language to state machine description is needed. Next, we will define the specification language (*Spec*) that, best to our knowledge, captures the intended behavior of the finite state system. The possible adopted conformance relation will be restricted to formal logical concepts of satisfaction and implication. The goal of this paper is to show the feasibility of a seamless integration of real system implementation into the verification process that encompasses transformation to a formal model (*Imp*), possible definition of interesting system behavior in terms of formal specification (*Spec*), and finally the conformance between *Imp* and *Spec*. The outlined synthesized process has some notable and concealed obstacles with computationally very expensive steps that will be identified and possible improvements proposed.

2. IMPLEMENTATION MODEL

A state machine (*SM*) is a simple mathematical model, fundamental and ubiquitous in computer-based systems. A programming language is one way to describe a state machine. The execution of a program corresponds to an execution of the state machine it describes. A software system is a complex state machine with a huge state space. Notable examples of such systems are reactive programs that maintain an ongoing interaction with their environment. In addition to input/output relations, such programs must be specified and verified in terms of their behavior.

A state system (machine) *SM*, as defined in this paper, is a 6-touple (S, I, O, N, S_0, Y), where S is the set of states, I the set of input bit-patterns, O the set of output bit-patterns, N the next state relation (often reduced to a function), $S_0 \subseteq S$ is the non-empty set of initial states and Y the output function. If S is finite, then *SM* is a finite state machine. Having N to be a relation, we can more easily model nondeterminism. An execution fragment is a finite or infinite sequence $(s_0, s_1, \ldots)$ of states such that *for all* i (s_i, s_{i+1}) is a step of *SM*. An execution (trace) is an execution fragment starting with an initial state of *SM*. A state is *reachable* if it is a state of an execution. The behavior of the system *SM* is the set of all traces of *SM*. With a model that has both finite and infinite traces, whenever one proves some property about the behavior it describes, one usually have to structure the proof into two parts, one to handle the finite trace case and one to handle the infinite trace case. If one cannot see the infinite traces one cannot talk about certain system properties like deadlock or fairness. To do so requires adding an additional structure (e.g. fairness constraints) to trace the behavior of the sate system. The paper concentrates primarily on the finite-trace models that encompass (possibly infinite) collection of finite traces.

Given a state machine model of a system, one can formally reason about properties of the model (not the real system itself).

The most important is the property of a truthfulness of associated expressions in any reachable state (or a set of states) along the trace of a system (ex. "Starting from the state s_i, can the system ever reach a state for which a deadlock is imminent?"). Hence, with each state there is an associated set of expressions given in some formal logic. The simplest kinds of expressions are atomic formulae in the propositional logic, defined from a set of valid propositions (*AP*). A labeling function $L{:}S \rightarrow 2^{AP}$ is introduced to associate with each state (member of *S*) an interpretation of the atomic propositions from the set *AP*, i.e. through *L* we know for each state which atomic propositions are assigned true. As the system evolves with time in discrete steps (change states) one can reason about its behavior. The aggregated model of a system implementation (or only a part of it) can be called *state machine with labeling* (*SML*). It denotes the usual Kripke structure [1] or the equivalent automaton.

A Kripke structure directly corresponds to an ω-regular automaton, where all the states are accepting. Specifically, a Kripke structure (S, S_0, R, L), where R is a total transition relation, can be transformed into an automaton $A=(\Sigma, S\cup\{l\}, \Delta, \{l\}, S\cup\{l\})$. In this notation the input alphabet is denoted by $\Sigma=2^{AP}$, *S* is the set of states and Δ is the transition relation. Fig.1 depicts a particular Kripke structure and the corresponding finite automaton [2]. In the sequel we will use the term *SML* to denote the class of analogues transition structures (with labeling on arcs or states).

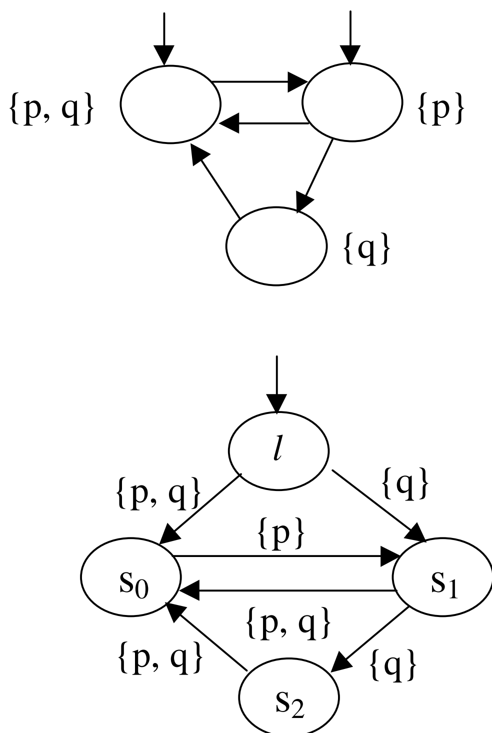

Figure 1. Kripke structure (top) and the equivalent automaton (bottom).

3. FORMAL SPECIFICATION LANGUAGE

A formal specification language *SL* provides a notation (syntactic domain), a universe of objects (semantic domain), and a precise rule definitions that specify which objects satisfy each specification. *SL* is a triple $<Syn, Sem, Sat>$, where *Syn* and *Sem* are sets and $Sat \subseteq Syn \times Sem$ is a relation between them [3]. The more explicit specification language implies the better-defined formal method. Specification language that is fundamentally advantageous in specifying system behavior should belong to the family of temporal logic. In this paper we concentrate on the branching discrete time temporal logic, i.e. computation tree logic (CTL) formulae [4].

Formula in CTL temporal logic can be true for some states of the underpinning model (denoted by the *SML*) and false for other states. Formulae my change their truth values as the system evolves from state to state. CTL formulae are built from atomic prepositions, standard Boolean operators, and temporal operators. Atomic propositions express state properties of the system (e.g. signal "valid_user" is set to true in the particular state). Temporal operators describe when these properties have to be satisfied. Each temporal operator consists of a path quantifier and a temporal modality. The path quantifier **A** indicates that a property is true for all computations (traces, paths), and **E** denotes that it is true for some computation (trace, path). The temporal modality describes ordering of events in time of computation, and can be one of the following: **F** (in some future and possibly current state), **G** (in every state), **X** (in the next state), **U** (strong until). In CTL one can easily express properties such as: "For any state, if a valid_user occurs (requesting validation), it will eventually be answered":

$$AG\ [\ (valid_user=true) \Rightarrow AF\ (answer=true)\ ]$$

Any CTL formula φ could be interpreted within the given *SML* structure as specifying a set of states, namely those states *Q* for which $(SML, s \models \varphi)$ holds, or formally:

$$Q(\varphi) \subseteq S \text{ is a set of states such that } Q(\varphi) = \{s \mid SML, s \models \varphi\}.$$

Usually we require that the initial state is included, i.e. $s_0 \in Q(\varphi)$. Clearly, the verification procedure encompasses exploration of states as the system evolves from the initial state s_0 in discrete time steps.

4. DEVELOPING *SML* MODEL

In hardware engineering, the introduction of description languages (HDL), like Verilog [5], has made it simple to design systems in terms of their behavior as well as their implementation. In addition, using behavioral descriptions makes it easy to write abstractions of designs and environments. Abstractions, which can be used to reduce system complexity, are an important aspect of hierarchical synthesis/verification of large systems. Most HDLs are designed for simulation and their semantics are either defined in terms of simulation results or left undefined. The lack of formal semantics makes it hard to apply advanced verification techniques to existing designs written in these HDLs, since it is hard to guarantee the same behavior of the synthesized and verified circuits. To make it possible to utilize verification algorithms one needs to bridge the gap between HDLs and the formal models (such as *SML*). One distinct solution to this problem is given in [6].

By using timed finite state machines to model behaviors of Verilog programs, authors separated the problem of determining whether the program is "implementable" from the problem of

deciding if a program is "synthesizable". The extracted timed finite state machines (or *SML*s) can model almost arbitrary programs, built from the defined subset of Verilog. However, the extracted timed *SML*s are not optimal in any sense, rather they only closely emulate Verilog program. Control flow graphs (*CFG*) represent the execution of a Verilog program. *CFG* are defined as multi graphs $G = (V_p + V_c, E)$, where V_p is the set of all distinct pauses, V_c is the set of all conditional statements and E is the set of edges, $e=(v_1, v_2) \in E$ *iff* $v_2 \in V_p + V_c$.
Conceptually, the product of two sets of machines; timing machines and untimed machines logically models Verilog process. Timing machines determine how long a program can stay in a certain state, using values of logical expressions in conditional statements. Untimed machines use program context and transitions of timing machines to compute the next state variables and resolve contention among variable updates.

Software engineering deals with the development of large and often complex information processing systems. The development process can hardly be carried out with modeling and description techniques that are based solely on programming languages [7]. As an aftereffect special modeling languages like SDL [8] and UML [9] have become popular. All these approaches are too syntax oriented and lack a proper and simple semantic foundation. Consequently, the tool support for design validation and verification remains shallow, even though the theoretical foundations are today quite powerful. What is needed however, is a mathematical theory that will give semantics to the description techniques. Unfortunately, a lot of work in the future is needed to achieve this goal.

Prevailing practices in attempting to verify software designs follow two approaches. In automated reasoning about a program fragment, a software tool carries out symbolic execution of a program or verifies that a program has certain required properties. Such a software verification tool must have a proper representation of an interested program fragment. Reasoning tools based on Automated Theorem Proving (ATP) require the program fragment to be represented as the set of logical clauses. Logical representation of even a very small program fragment requires many input clauses that must be generated by hand. Hence, what the user is trying to verify is not an implemented program fragment, but rather its image generated by an informal method.

The other approach exploit a special modeling language (e.g. SMV, [10]) that has a synchronous data-flow semantics and can verify programs in this language with respect to temporal logic formulas (*Spec*). CTL class of the specification language is given in the previous section. The input language of SMV is designed to allow the description of finite state systems and supports parallel assignment syntax. The semantics of assignment is similar to that of single assignment data flow languages. A program fragment can be viewed as a system of simultaneous equations, whose solutions determine the next state. Again, a verification process is carried out not on the actual programming language implementation, but on the specific SMV representation, developed informally and resembling a program fragment. What is needed here is an automatic, possibly sound and complete transformation of the SMV program into a final implementable language, or a transformation of a software design (given in the particular programming or specification language) into the SMV description (analogous to the development of a finite state machine model from the Verilog description in hardware engineering).

5. DEDUCTIVE VERIFICATION

Theorem proving is a technique where both the system and its desired properties are expressed as formulas in some mathematical logic. This logic is given by a *formal system*, which defines a set of axioms and a set of inference rules. Theorem proving is the process of finding a proof of a property from the axioms of the system. There are many references covering various types of proof systems such as: natural deduction, axiomatic systems, tableau, etc. We hypothesize that the procedural semantics of proof systems is essential to the verification process, and therefore our focus is on the resolution-based deduction, explicated by the following well-known algorithm (for the predicate logic):

```
Deduction {
Define_implementation_as_wff_formulas_{Ea};
Define_property_as_wff_formulas_[Hb};
Form_the_set_{Fc}={Ea}∪ ¬{Hb};
Convert_{Fc}_to_clauses_{Ki};
repeat
   Chose_clauses_k1_and_k2_from_{Ki};
   New-k := resolve(k1, k2);
   New-k' := simplify(New-k, {Ki});
   {Ki} := simplify({Ki}, New-k');
until
   (empty_clause_found OR
   no_more_deductions);
if
   (empty_clause) return "proved";
else
   return "no_contradiction";}
```

There are many difficulties in executing this algorithm, but the most notable are clausal representation, selection strategy and resolution with equalities. In the sequel we will briefly explain each of them.

Clausal representation of a computing system ranges from an apparent low-level hardware description, to a detailed symbolic description of a program. Even though theoretically sound, representing program state (*PS*) of a software design as a logic predicate is very tedious and frustrating. When defining terms of the predicate *PS*, in addition to the current instruction symbol (*current_inst*), one needs to define functions to construct a list of names and values of variables in that state (*var_vals*):

$$PS(current_inst, var_vals)$$

A huge additional step in presenting a program to an automated theorem prover is to write the clausal representation for every statement in the program. A simple assignment statement (*AS*) may have the predicate form:

$$AS(name, as_var, as_exp, next_inst)$$

Where *name* defines the statement, *as_var* is the variable assigned to, *as_exp* defines the expression to be evaluated, and *next_inst* is the next instruction to be executed. General axiom that says: "in state *p* applying the assignment *a* leads to the state *q*", may have the predicate logic well formed form:

$$(PS(p1, p2) \wedge AS(a1,a2, a3,a4)) \Rightarrow PS(q1, q2)$$

or in the normalized clause form:

$$\neg PS(p1, p2) \vee \neg AS(a1,a2, a3,a4)) \vee PS(q1, q2)$$

A selection strategy for *k1* and *k2* is chosen according to some heuristics on the nature of the application (ad hoc approach).

Finally, equality is a special relation (reflexive, symmetric and transitive), very important to symbolize many theorems in a

natural way. To properly use the equality relation in a theorem prover its features must be specified with a set of axioms, leading to an extension of the formula set. Moreover, during the execution of the proving process this method generates numerous useless clauses. Other approaches employ the *paramodulation* inference rule, or some form of *rewriting*, [11]. Paramodulation rule is an extension to non-unit of the equality substitution which says that if a clause C contains term t and a unit clause is $s=t$ then infer C with s replacing one single occurrence of t. There are problems of over-generation of paramodulants, resulting again in an uncontrolled growth of the size of terms. There is a need to restrict the applicability of the paramodulation. A partial solution comes from the theory of rewriting systems. The basic idea of rewriting systems is to orient equations into rewrite rules with a partial order defined on first order terms, i.e. allow substitutions of equal terms only in some directions. In this way an equation $s=t$ is used as a rewrite rule $s \Rightarrow t$ that can be used only to rewrite instances of s with instances of t (and not vice versa).

An advantage of deductive verification though, is that it can be used for reasoning about infinite state systems. The task can be automated to a limited extent. In the process of finding a proof, the human user often gains invaluable insight into the system and the property being proved. However, because of the undecidability of a general mathematical logic, no limit can be placed on the amount of time or memory that may be needed in order to find proof. In particular, the theory of computability shows that there cannot be an algorithm that decides whether an arbitrary computer program terminates. This limits what can be verified automatically. Nevertheless, theorem provers are increasingly being used today in the mechanical verification of safety critical properties of hardware and software designs.

6. VERIFICATION BY MODEL CHECKING

In the classical proof-based verification, the system description is a set of formulas Γ (in a suitable logic) and the specification is another formula φ. The verification method consists of trying to find a proof that $\Gamma \vdash \varphi$ (i.e. by applying allowable rules to the set Γ, derive φ). In a model-based approach [2], [10], [12], a finite model M in an appropriate logic represents the system. The specification property is again represented by formula φ. The verification method consists of computing whether a model M logically satisfies φ (i.e. $M \models \varphi$). The model-based approach is potentially simpler that the proof-based approach, for it is based on a single model M, rather than a possibly infinite class of them. In model checking one is not concerned with semantic entailment ($\Gamma \models \varphi$), or with proof theory ($\Gamma \vdash \varphi$), but rather with the notion of *satisfaction*, i.e. the satisfaction relation between the model and a formula ($M \models \varphi$).

The model M is a transition system and the properties φ are formulas in temporal logic. Since computing processes are predominantly state transition systems, they can be appropriately modeled by the *SML* formalism. Clearly, the model checking technique requires an exploration of states as the system evolves from the initial state s_0 in discrete time steps. All possible successor states are given by the relation R. We use the notation $R(s)$ to denote the set $\{ t \in S \mid (s,t) \in R\}$. From this, it is easy to derive the Boolean immediate neighboring function, as given in [12]:

$$\eta: S \times S \rightarrow B, \text{ with } \eta(s, t) = TRUE \text{ if } (s, t) \in R,$$

where B={0, 1}. The function η clearly gives the set of neighboring states H(S):

$$H(S) = \{t \mid \exists_{s \in S} \ \eta(s, t) = TRUE\} \quad (1)$$

Assuming the above formal preliminaries, we can compute the set of all reachable states RC, given the set of initial states S_0. Let us denote the set RC to comprise recursively all the initial sets (S_0) united with all the states that are immediate successors of so far reachable states. The traditional breadth-first computation to derive the set of reachable states is given in the following algorithm:

```
find_reach (S0){
  RC:=∅; New:=S0;
  do {
      RC:= RC ∪ New;
      Next := H(New);
      New := Next\RC;
      } while (New ≠ ∅);
  return RC;}
```

The presented algorithm is expressed in terms of operations on the sets of states. Finite states can conveniently be implemented using bit vectors data types. Assuming that all bit vector operations needed for the above algorithm take $O(|Q|)$ time, where $Q \subseteq S$, and that in the worst case only one state is discovered "reachable" per iteration, the overall complexity is $O(|Q|^2)$. However, the number of reachable states is in the worst case exponential in the number of register signals (bits), rendering this approach to state space exploration unrealizable.

A much more efficient computer representation and manipulation of *SML* structure is achieved by Reduced Ordered Binary Decision Diagrams (ROBDD), or BDD for short. BDDs were first studied as a useful representation of boolean expressions. Binary decision diagrams have their roots in the decomposition of Boolean functions given by the Shannon expansion with respect to a variable:

$$f = x_i f_{xi} + x_i' f_{xi'} \quad (2)$$

with cofactors:

$$f_{xi}(x_1, \ldots, x_n) = f(x_1, \ldots, x_{i-1}, 1, x_{i+1}, \ldots, x_n)$$
$$f_{xi'}(x_1, \ldots, x_n) = f(x_1, \ldots, x_{i-1}, 0, x_{i+1}, \ldots, x_n).$$

The expansion formula can be represented by a binary tree with root node x_i, and left branch as the cofactor f_{xi} , and right branch as the cofactor $f_{xi'}$. Recursive application of the Shannon expansion theorem, while keeping strict variable ordering, with identification of isomorphic sub-graphs and removal of redundant nodes, concludes the building of a BDD. The power of BDD representation is that it does not explicitly enumerate Boolean function values, but rather values are given by tracing all paths to the terminal node (rendering exponential number of paths by a linear number of nodes). Such a Boolean function representation is canonical, but its size can critically depend on the variable ordering.

The key to exploiting the power of BDD representation is to express a problem in a form where all mathematical objects are represented as Boolean functions. Let B = {0, 1} and let $S \subseteq B^n$ be a set of points of the Boolean space. It is possible to define a function $\chi_S : B^n \rightarrow B$, called the *characteristic function* of S, that has the value of 1 exactly for the points of B^n that are in S:

$$\forall x \in B^n, \ x \in S \Leftrightarrow \chi_S(x) = 1 \quad (3)$$

The definition of the characteristic function can be extended to subsets of an arbitrary finite set Q, by providing an injective mapping $E : Q \rightarrow B^n$. Such a mapping is called a binary encoding of Q. In many applications, the domains have a natural encoding, e.g. the binary encoding of finite integers.

The set of states in a computing systems are not represented explicitly, but rather sets are represented by their characteristic functions and subsequently by BDDs. This approach is known as *symbolic* or implicit analysis. The key point is that the BDD representing a set of states may be quit small. For the correct interpretation of a set as a BDD, we should a-priori fix a set of encoding (placeholder) variables $\{d_i \mid 0 \leq i < n \}$, and assume all characteristic functions be expressed in them.

Likewise, for a binary relation $R \subseteq Q_1 \times Q_2$, we define its characteristic function: $\chi_R : Q_1 \times Q_2 \rightarrow B$. Given different sets of variables to identify the elements of Q_1 and Q_2, the characteristic function is:

$$\chi_R(x_1, \ldots, x_n, y_1, \ldots, y_m) \leq \chi_{Q1}(x_1, \ldots, x_n) . \chi_{Q2}(y_1, \ldots, y_m) \quad (4)$$

This generalizes to n-ary relations.

It is now straightforward to give the calculation of a set of next states given the set of present states Q_1 and the transition relation *R*:

$$Next(Q_1) = \exists_{xi} [R(x_i , y_j) \wedge Q_1(x_i)] \quad (5)$$

By performing an existential abstraction, we extract only the second elements of pairs. The above operation is performed with BDDs (representing characteristic functions):

$$\chi_{next} = \text{ True if } \exists_{xi} [\chi_R \wedge \chi_{Q1}]$$

7. METHODS OF IMAGE COMPUTATION

In the previous section it was shown that the model checking technique requires an exploration of states as the system evolve from the initial state s_0 in discrete time steps. Given the *SML* structure by the set of states and the transition relation *R*, one can (in principle) easily compute the set of states *Q* that satisfy basic temporal properties, namely EX, EG, EU, [12]:

$$Q(EX\ f) = R^{-1} (Q(f))$$

Computing of Q(EG f) is given as an algorithm:

```
2^S EG (CTL f){
 k:=0; Zk:=S;
 do {
  Zk+1:=Q( f )∩ R^-1 ( Zk );
  if (Zk+1=Zk) return Zk;
  k++;
 } forever;
}
```

Computing of Q(EU f, g) is given by another algorithm:

```
2^S EU (CTL f, CTL g){
 k:=0; Zk:=∅;
 do {
  Zk+1:=Q(g)∪(Q( f )∩R^-1 (Zk) );
  if (Zk+1=Zk) return Zk;
  k++;
 } forever;
}
```

While examining above algorithms, it is observed that basic operations include set operations such as union and intersection. These operations can be computed in time proportional to the product of number of BDD nodes representing the sets. It is obvious that the largest set to be represented is the set given by the inverse transition relation R^{-1}. Although the presented algorithms are constructed using the inverse transition relation, the same can be done using forward transition relation, termed *image computation*. This paper will address the latter.

Transition function method

The Transition function, introduced in [13] was the first one used in the image computation process. Image computation is defined by the following expression:

$$\exists x((\bigwedge_{1 \leq i \leq n} y_i \Leftrightarrow f_i(x)) \wedge S(x)) \quad (6)$$

where x are all present state variables ($x_1, x_2 \ldots x_n$), y_i is one of the next state variables, f_i is the next state function as given before, and *S(x)* is the set of states for which image computation is taken. *S(x)* and f_i are represented as BDDs, and the same goes for the result of the image computation. For the simplicity we shall mark the following expression as T_i(bit relation): $T_i = y_i \Leftrightarrow f_i(x)$.

Since the product of the T_i (Relational product) is very large when represented as BDD and the result of the image computation is not, it is necessary to avoid its size explosion. To prevent size explosion, the constraint operator ($\downarrow$) is used which can reduce the amount of memory needed. Another two methods for minimizing the amount of used memory are Domain partitioning and Co-domain partitioning, both based on recursive division of the computation.

The Domain partitioning is a method in which division of the problem given by (6) is made regarding current state variables as shown in following expression:

$$(6) = \exists x(\bigwedge_{1 \leq i \leq n} (T_i)_v \wedge S_v) \vee \exists x(\bigwedge_{1 \leq i \leq n} (T_i)_{\neg v} \wedge S_{\neg v})$$

where v is one of the current state variables(x_j). Index *v* in the expression denotes Shannon cofactor with respect to the variable *v*. The Co-domain partitioning is the division method made on next state variables in the same manner as the domain partitioning:

$$(6) = \left[v \wedge \exists x(\bigwedge_{1 \leq i \leq n} (T_i)_v \wedge S) \right] \vee \left[v \wedge \exists x(\bigwedge_{1 \leq i \leq n} (T_i)_{\neg v} \wedge S) \right]$$

where v is one of the next state variables. It was shown by [13] that Domain partitioning outperforms Co-domain partitioning in most of the cases. However there are cases in which one of the methods runs for extremely long time (didn't finish) while the other gives the results in reasonable time.

Transition relation method

The Transition relation method is based on calculation of conjunctions in the expression (6), which regularly might result in BDD size explosion. To avoid the explosion, the following theorem (distribution of existential quantification over conjunction) is used:

$$\exists x[f(x, y) \wedge g(y)] = \exists x[f(x, y)] \wedge g(y) \quad (7)$$

If the theorem (7) is applied to the expression (6), it can be observed that (6) can be simplified during the computation. If

conjunctions are taken in such an order that results of conjunctions can be existentially quantified as soon as possible, the intermediate results during the image computation would be smaller in size. There have been a number of implemented algorithms that address quantification schedule due to many possible criterions of picking conjuncts. The criterion should be as simple as possible, should be independent on any other optimization criteria (such as variable ordering) and should perform well for any circuit and any given current state [14].

Hybrid method

Even by using early and efficient quantification schedulers there is still a class of systems that can't be traversed with transition relation method.

A dependence matrix *D* is defined as a matrix in which rows represent bit relations *Ti* and current state set *S*, while columns represent current state variables. Elements of the matrix are defined as *D(i,j)=1* if the function bounded by *i*-th row depends on variable bounded by *j*-the column and *D(i,j)=0* otherwise. Systems that can't be traversed with transition relation method have dependence matrix close to full since there is no "smart" schedule for the case in which every bit relation depends on all variables.

The solution lies in division of the matrix into two submatrices. The division is made as in the Domain partitioning manner [15], that is one of the submatrices is calculated by setting current state variable *v* to *true*, and the other one by setting variable *v* to false. The resultant matrices were shorter for the column that was bounded to splitting variable *v* and possibly not as full as the matrix before the division. In the case of full resulting matrices, the division process can run iteratively. When the resulting submatrices come close to the triangular form, the Transition relation method is being applied. In order to decide whether the conjunction should be made or not, a Normalized average lifetime (NAL) is being used.

8. CONCLUSION

In this paper we have presented a feasibility analysis of formal verification methods in the design of computing systems that can be abstracted by state transition models. The analysis can be summarized in two directions: (a) along the computing system to be verified (hardware or software) and (b) along the employed verification method (deductive or model checking).

It is essential to note that the verification is employed not on the implemented system itself, but on the abstracted *SML* model. In this respect hardware systems are in better position since the introduction of description languages (HDL) has made it simple to design systems in terms of their behavior as well as their implementation. Behavioral descriptions are the basis for writing transition abstractions of designs and environments.

Software systems are much more difficult to verify. There is no automatic method for abstracting a common programming language fragment into a formal model. A user is compelled to manually generate the logical (clausal) description of the interested program fragment (in the case of deductive verification), or carry out a coexisting programming in a language that can be automatically abstracted to *SML* (in the case of model checking verification).

Focusing on the deductive verification method, the most difficult steps that require careful, precise and ever diverse approach are the clausal symbolic representation, selection strategy, and resolution with equalities.

In the model checking approach we have identified the image computation as potentially expensive operation (even with BDDs). We have presented three methods from the literature and examined their performance. Image computation by the Hybrid method shows a potential for improvement when carried out on a distributed system, since it can be made at the level that is higher than the BDD package level. Because the Hybrid method does the splitting as the first step, it is possible to compute each of the subtasks on different machines.

9. REFERENCES

[1] P.Wolper, "Temporal Logic Can Be More Expressive", *Information and Control*, No. 56, 1983, pp. 72-99.

[2] Clarke, E. et al, *Model Checking*, The MIT Press, 1999.

[3] J.V.Guttag et al, "Some Remarks on Putting Formal Specification to Productive Use", *Science of Computer Programming*, Vol. 2, No. 1, Oct. 1982, pp. 53-68.

[4] E. A. Emerson, "Temporal and modal logic", in J.Van Leeuwen (ed.), *Handbook of Theoretical Computer Science*, Volume B, Formal Models and Semantics, The MIT Press,1990, pp.995-1072.

[5] D.E. Thomas, P.R. Moorby, "The Verilog Hardware Description Language", Kluwer Academic Publishers, 1991.

[6] S-T. Cheng et al, "Compiling Verilog into Timed Finite State Machine", *Proc. of the 4th IEEE International Verilog HDL Conference, IVC'95*, Santa Clara, 1995, p.32-39.

[7] M.Broy, Toward a Mathematical Foundation of Software Engineering Methods, *IEEE Trans. On Software Engineering*, Vol. 27, No.1, January 2001, pp.42-57.

[8] "Specification and Description Language (SDL), Recommendation Z.100", Technical report, CCITT, 1988.

[9] G. Booch et al, "The Unified Modeling Language", Addison-Wesley, 1999.

[10] K.L.McMillan, The SMV System, Carnegie-Mellon University, 1992.

[11] Newborn, M., Newborn, M., *Automated Theorem Proving: Theory and Practice*, Springer Verlag, 2001.

[12] Janssen, L.J.M, *Logics for Digital Circuit Verification*, Ph.D. Thesis, Eindhoven Tech. University, 1999.

[13] Coudert, O. and Madre, J.C.. "A unified framework for the formal verification of sequential circuits", *Proceedings of the IEEE International Conference on Computer Aided Design*, pp. 126-129, November 1990.

[14] D. Geist and I. Beer: "Efficient model checking by automated ordering of transition relation partitions.", *Computer Aided Verification, 6th International Conference, CAV'94 Proceedings*, vol. 818 of Lecture Notes in Computer Science, pp. 299--310, 1994.

[15] Moon IH. Et al, "To split or to conjoin: the question in image computation", Proc. of the *Design automaton conference*, pp. 23-28, 2000.