VERIFYING INTERACTING FINITE STATE MACHINES: COMPLEXITY ISSUES

by

Adnan Aziz, Vigyan Singhal, and Robert K. Brayton

Memorandum No. UCB/ERL M93/52

1 July 1993

(Revised 24 October 1994)

cover page

VERIFYING INTERACTING FINITE STATE MACHINES: COMPLEXITY ISSUES

by

Adnan Aziz, Vigyan Singhal, and Robert K. Brayton

Memorandum No. UCB/ERL M93/52

1 July 1993

(Revised 24 October 1994)

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

Verifying Interacting Finite State Machines : Complexity Issues

Adnan Aziz Vigyan Singhal
Robert K. Brayton *
Email: {adnan,vigyan,brayton}@cs.berkeley.edu
Fax: 1 (510) 643-5052
Department of Electrical Engineering and Computer Sciences
University of California, Berkeley, CA 94720, USA

Abstract. In this report we carry out a computational complexity analysis of a simple model of concurrency consisting of interacting finite state machines with fairness constraints (IFSMs). This model is based on specification languages used for system specification by actual formal verification tools, and it allows compact representation of complex systems. We categorize the complexity of two problems arising in this model that are of fundamental importance:

Formal verification Given a property (expressed as a formula in the logic CTL), deciding if it holds of a system of IFSMs is PSPACE-complete.

Trace universality Given a system of IFSMs, deciding if the set of output traces generated by the system is universal is EXPSPACE-complete.

For a single machines the verification and trace universality are decidable in polytime, and complete for PSPACE respectively. Thus our results demonstrate a tradeoff between the ability to compactly describe systems using concurrency, and the increased complexity of analyzing such systems.

* Supported by SRC Grant 93-DC-008 and NSF/DARPA Grant MIP-8719546

1 Introduction

The verification of sequential systems is an area of active research in VLSI. Verification of a design is typically done by modelling the design as a finite state machine. Properties to be verified can be specified by formulae in a linear time or branching time temporal logic[15, 4], or by the set of output traces accepted by a task automaton[12].

Verification algorithms proceed by performing some form of traversal of the state transition graph[5, 3]. Thus, given a system explicitly described by its state transition graph (STG), verification can proceed efficiently i.e., in time that is polynomial in the number of states. In particular, properties expressed in the logic CTL [4] can be checked in time proportional to the size of the STG times the length of the formula. Similarly, when the task automaton can be efficiently complemented, verification proceeds in time proportional to the product of the number of states in the system and the number of states in the task.

In practice, large systems are designed in a hierarchical fashion. Thus they are made up of small interconnected components. Composition of the system leads to the state explosion problem[3]. Informally, this refers to the fact that given n Finite State Machines (FSMs) the number of states in the product machine is the product of the number of states in each individual machine. As a result algorithms that explicitly operate on the state space of the product machine have exponential time and/or space complexity. Various techniques have been proposed for dealing with the complexity introduced by concurrency. For example, McMillan [16] uses the binary decision diagram data structure to compactly represent product machines. Other approaches proceed incrementally forming the product and minimizing the intermediate products with respect to some equivalence on the states [8, 20]. Abstraction techniques are used to scale datapath [14].

In this paper we define a formal model of concurrency, consisting of interacting finite state machines with fairness constraints (IFSMs). We argue that it is well suited to hardware verification; indeed it is based on specification languages used for system specification by the formal verification tools SMV [16] and HSIS [1]. We categorize the complexity of the two following problems on this model that are of fundamental interest :

- *Formal verification* - Given a property (expressed as a formula in the logic CTL), deciding if it holds of a system of IFSMs is PSPACE-complete.
- *Trace universality* - Given a system of IFSMs, deciding if the set of output traces generated by the system is universal is EXPSPACE-complete.

Our proofs have several interesting corollaries. For example, the high complexity of verification can occur in very simple cases where all machines are identical. Furthermore, the fact that model checking the logic CTL on interacting finite state machines is in PSPACE is actually a positive result – given the branching nature of CTL one have expected the problem to be EXPTIME-complete.

We have come across various remarks in the literature alluding to the PSPACE hardness of the reachability problem (defined in section 3) for various models which allow compact notation, e.g. [13, 10]. However we have not come across any published proofs of this fact for models similar to the ones we deal with. Furthermore we have not found any proofs of the PSPACE completeness of the verification problem. Similarly, lower bounds on universality exist for other models of concurrency, e.g. the LTS model [17]. However these models are not immediately seen to be interpretable in IFSMs, and so our result on universality is also novel.

In section 2 we define the notions of a finite state machine and weak fairness. The semantics of interaction are made precise, and we motivate the definition of IFSMs and show how they can be used to model hardware. We also define the syntax and semantics of CTL on a models defined by IFSMs. In section 3 we prove complexity theoretic lower bounds for verification and trace universality. We conclude by going over the consequences of our results to issues in practical verification and discuss future work.

2 Definitions

Definition 1 Finite State Machine. A *finite state machine* $\mathcal{M}$ is a 5-tuple (S, r, I, O, T) where

- S is a finite set of states
- r is the *initial state*
- I is a finite set of *inputs*
- O is a finite set of *outputs*
- $T \subset S \times I \times O \times S$ is the *transition relation*

An FSM can be represented in terms of a state transition graph (STG) or a register with a table, as depicted in figure 1. Informally, $(s, i, o, t) \in T$ means that from state s on input i, there is a transition to state t, while the output is o. Given state s and input sequence $\alpha = (a_0, a_1, \ldots)$, a **path** is a finite or infinite sequence of states starting at s consistent with the transition relation. For the

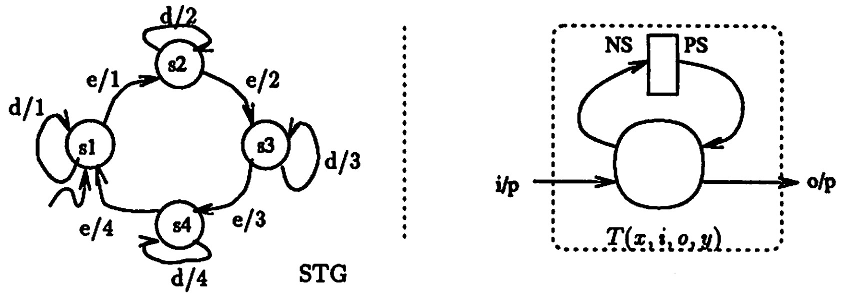

Fig. 1. Visual representations of FSMs

machine in figure 1, (s_1, s_2, s_2, s_3) is a path starting at state s_1, corresponding to input (e, d, e).

Fairness constraints [6] are restrictions on infinitary behavior of FSM used to model system, environment, and properties. An infinite path is *fair* iff it satisfies the constraint - behavior is restricted to sequences of fair paths As an example consider the system described in 2. The fairness constraint is "visit state START infinitely often". Thus the only output sequences corresponding to fair paths are those in which every *count* is eventually followed by a *start*.

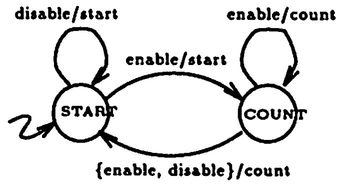

Fig. 2. Fairness condition - state "START" visited infinitely often

Definition 2. A *Büchi* (*weak*) fairness condition on a machine $M = (S, r, I, O, T)$ is characterized by a subset of the state space, i.e. by some $B \subset S$. The path σ is *fair* if and only if some state from B occurs infinitely often on σ.

Various other formalisms exist for expressing fairness constraints. In particular, Streett [18] (*strong*) conditions have the property of being succinct while still being efficient to compute with. All our results continue to hold for systems with Streett fairness conditions.

Complex designs typically arise as the composition of smaller interacting machines, where the input to a machine can be the output of another machine.

The entire system defines a finite state machine, referred to as the product machine, defined below.

Definition 3 Product Machine. Given a collection of interconnected FSMs $\{M_1, M_2, \ldots, M_n\}$, the *product machine* $M = M_1 \times \ldots \times M_n$ is an FSM on the product state space $S_1 \times S_2 \times \ldots \times S_n$.

- The initial state is the Cartesian product of the initial state in each component.
- The inputs to the product machine are those inputs to $\{M_1, M_2, \ldots, M_n\}$ **which are not outputs of other machines.**
- The output corresponds to some some subset of all outputs of the components; it is a subset rather than the entire set since not all the intermediate outputs may correspond to observable outputs.
- The transition relation of the product is defined by the requirement that the transition in each component M_i satisfies T_i, and that the input and output assignments be consistent; it is the conjunction of the component transition relations in an appropriate boolean algebra [12].

Given fairness conditions $B_1, B_2, \ldots, B_n$ on the component machines, the fairness condition on the product is the set of states $B_1 \times B_2 \times \cdots \times B_n$. Thus a path through the product machine is fair if the path followed in each component is fair.

Intuitively, we are dealing with interacting Mealy machines. The definition of the product machine is motivated by the physical basis for hardware: (1) wire values must be consistent, (2) registers are triggered by common clock, and so all machines change state synchronously. An example of composition of machines M_1 and M_2 is shown in figure 3. A *closed system* of interacting FSMs is one to which there are no external inputs. Any open system can be closed by adding machines that simulate external inputs.

A 2-input NAND gate can be emulated by an FSM as shown in figure 4. Thus *Boolean logic gates* can be embedded in IFSMs. This feature allows compact representation of dynamics on the product space that can be very complex. Any function computed in polytime can be computed by a polysize circuit. Hence any formalism for representing a transition structure on a product space that is efficiently computable can be translated to a compact representation in the interacting FSM model.

SMV [16] and HSIS [1] are two tools used for formal verification of hardware designs. The formal models derived from the specification languages for both

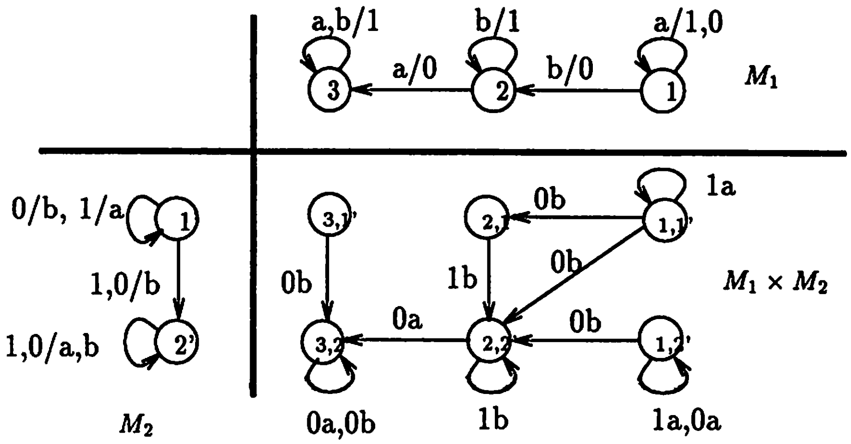

Fig. 3. Product of M_1 and M_2 yields M

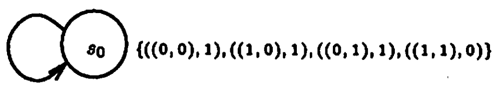

Fig. 4. A one state FSM that emulates a NAND gate - State space = $\{s_0\}$, Input space = $\{(0,0),(1,0),(0,1),(1,1)\}$, Output space = $\{0,1\}$

systems are efficiently interpretable as a product machine with fairness conditions; furthermore a system of IFSMs with fairness conditions can efficiently be specified in SMV and HSIS. This provided the basis for our desire to analyze complexity issues related to IFSMs with fairness conditions.

2.1 Formal verification

Consider a closed system of interacting mahines $M = M_1 \times M_2 \times \ldots \times M_n$. Let the state spaces of each machine be $S_1 = \{s^1_1, s^1_2, \ldots, s^1_{|S_1|}\}$, $S_2 = \{s^2_1, s^2_2, \ldots, s^2_{|S_2|}\}$, $\ldots$, $S_n = \{s^n_1, s^n_2, \ldots, s^n_{|S_n|}\}$. The syntax and semantics of CTL are defined as follows:

Definition 4 CTL Syntax. The set of *atomic propositions* of the system is the set of symbols $\{\underline{s^j_i} \mid s^j_i \in S_j\}$, i.e. there is a symbol for each state of a component machine. The syntax of CTL is defined inductively below:

- p is a CTL formula, where p is an atomic proposition
- if ψ_1 and ψ_2 are CTL formulae, then so are $\neg\psi_1$, $\psi_1 \vee \psi_2$, $\exists X \psi_1$, $\exists G \psi_1$, and $\exists[\psi_1 U \psi_2]$

The expression $\exists F\phi$ abbreviates the CTL formula $\exists(\text{TRUE}\, U\, \phi)$. CTL formulae express properties of the system that are true or false at a state. Intuitively, the formula $\exists X\psi_1$ asserts the property that there is a next state in which ψ_1 holds. Similarly $\exists G\psi_1$ asserts there exists a path along which every state satisfies ψ_1; $\exists[\psi_1 U \psi_2]$ asserts that there is a path to a state where ψ_2 holds, and at each prior state ψ_1 holds.

The formal semantics of CTL on a closed product machine $M = M_1 \times \ldots M_n$ with fairness conditions are given below. Given an infinite path π, $[\pi]_i$ refers to the i-th state on the path.

Definition 5 CTL Semantics. Given a state $s \in S_1 \times S_2 \times \ldots \times S_n$ and a formula ϕ from the logic CTL, we define the satisfaction predicate $s \models \phi$ inductively as follows

1. $\phi = p$ where $p = \underline{s_j^i}$ is an atomic proposition: $s \models \phi$ if and only if there exists a fair path starting at state s, and $[s]_i = s_j^i$
2. $\phi = \neg\psi_1$: $s \models \phi$ if and only if $s \not\models \psi_1$
3. $\phi = \psi_1 \vee \psi_2$: $s \models \phi$ if and only if $s \models \psi_1$ or $s \models \psi_2$
4. $\phi = \exists X\psi_1$: $s \models \phi$ if and only if there exists a fair path π starting at s such that $[\pi]_1 \models \psi_1$
5. $\phi = \exists G\psi_1$: $s \models \phi$ if and only if there exists a fair path π starting at s such that $\forall i [\pi]_i \models \psi_1$
6. $\phi = \exists[\psi_1 U \psi_2]$: if and only if there exists a fair path π starting at s such that $\exists i[[\pi]_i \models \psi_2 \wedge \forall j < i [\pi]_j \models \psi_1]$

3 Complexity Issues

3.1 Formal verification

In this section we shall prove that it is PSPACE-complete to model check CTL formula on a system of interacting machines with fairness conditions.

To do so we first demonstrate that simply determining if there is a path from one state to another state in a product machine is PSPACE-complete.

Definition 6. Given a product machine M and two states $s = [s_1 \ldots s_n], t = [t_1 \ldots t_n]$ in M, t is said to be *reachable* from s if there exists a of states in the product machines $< u_0, u_1, \ldots, u_k >$ such that $u_0 = s$, $u_k = t$, and $\forall l < k$ there is an input i_l under which there is a transition from state u_l to u_{l+1}.

Lemma 7. It is PSPACE-complete to decide reachability.

First we categorize the complexity of determining if there exists a transition from one state to another in the product machine.

Definition 8. Given a product machine M and two states $s = [s_1 \dots s_n], t = [t_1 \dots t_n]$ in M, t is said to be *one step reachable* from s if there is a transition from s to t under some input.

Lemma 9. It is NP-complete to decide one step reachability.

Proof. Assignments to the inputs and outputs can be generated nondeterministically; it can be checked in linear time that the assignments are consistent (i.e. inputs of machines that are outputs of other machines are assigned mutual values). It can also be checked in linear time that the assignment leads to transitions from $[s]_i$ to $[t]_i$ in each component machine M_i. Hence one step reachability is in NP.

Since FSMs can be used to model Boolean logic gates, a Boolean logic network can be efficiently simulated as a network of interacting FSMs. The output of this network can be used to enable a transition between states s and t; hence t will be one step reachable from s if and only if the logic network is satisfiable. Satisfiability of logic networks is NP-complete [7]; ergo one step reachability is NP-hard. ■

We are now ready to prove lemma 7

Proof. Membership in PSPACE is direct: a path $(s_1 \dots s_n) \rightarrow (a_1 \dots a_n) \rightsquigarrow \dots \rightsquigarrow (t_1 \dots t_n)$ can be non-deterministically generated and checked at each step (by lemma 9 this check can be carried out in NP). Observe that only successive transitions need to be stored. Hence the procedure runs in non-deterministic PSPACE. By Savitch's theorem [9], NPSPACE = PSPACE, and so reachability is in PSPACE.

PSPACE-hardness follows from a generic Turing machine reduction. Let $\mathcal{P}$ be an arbitrary problem in PSPACE. There is a Turing Machine and polynomial $p(n)$ such that machine decides all instances of $\mathcal{P}$ of size n using less than $p(n)$ tape. We will show how to simulate runs of the Turing machine on problem of size n can be simulated by $p(n)$ FSMs

We recall the definition of a Turing machine as given in [7]. A Turing machine is characterized by

- Finite set Γ of tape *symbols*, including a subset $\Sigma \subset \Gamma$ of *input symbols* and a distinguished *blank symbol* $b \in \Gamma - \Sigma$
- Finite set Q of states, including *start state* q_0, *final* states q_Y and q_N
- *Transition function* $\delta : (Q - \{q_Y, q_N\}) \times \Gamma \rightarrow Q \times \Gamma \times \{1, +1\}$

The machine can be schematically represented as in figure 5.

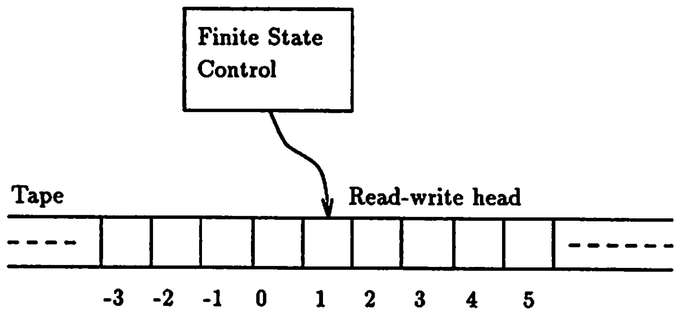

Fig. 5. Turing machine schematic

The input to the Turing machine is a string $x \in \Sigma^*$ placed in tape squares 1 to $| x |$; the remaining squares are marked with blanks. The machine starts at state q_0 with the read-write head scanning square 1, and goes through a sequence of computations defined by δ: the read-write head moves back and forth and contents of tape head are updated based on the value computed by δ; the computation ends on entering states q_Y, q_N.

A Turing machine on input of size n can be simulated by $p(n)$ interacting FSMs arranged in a linear array with bidirectional communication between neighbors, as depicted in figure 6. Each machine's state identifies the the contents of the corresponding tape square; the machine has a copy of the state transition graph for the controller. Only one machine is active at any time; control is passed across neighboring machines depending on head movement. The input-output arrangement, as well a portion of the state transition diagram is described in figure 7. From the construction it follows that the Turing machine

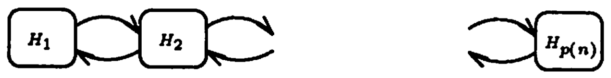

Fig. 6. Array of $p(n)$ FSMs simulating the TM on input of size n

will accept the input iff the FSM array can reach an accepting final state. Furthermore the array can be constructed in time $p(n)$ and so the reduction is polytime. Thus reachability is PSPACE-complete. ∎

Note that the construction above yields a very simple systems of FSMs – all the machines are identical, the transition structure of each machine is determin-

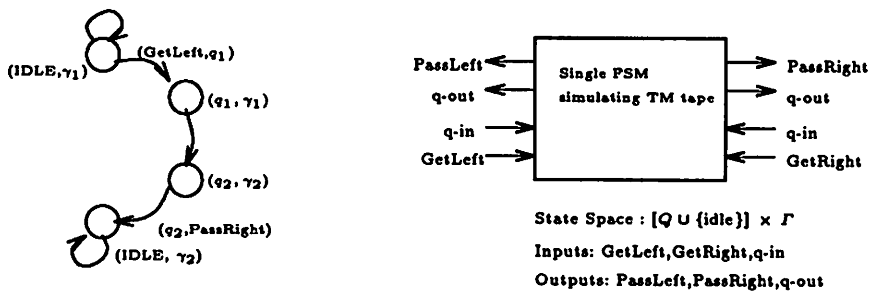
(IDLE,γ1)
(GetLeft,q1)
(q1, γ1)
(q2, γ2)
(q2,PassRight)
(IDLE, γ2)
PassLeft
q-out
q-in
GetLeft
Single FSM simulating TM tape
PassRight
q-out
q-in
GetRight
State Space : [Q ∪ {idle}] × Γ
Inputs: GetLeft,GetRight,q-in
Outputs: PassLeft,PassRight,q-out

Fig. 7. FSM used to simulate Turing Machine tape square; each FSM state is the product of the set of tape symbols with the controller state. On the left is a fragment of the STG, on the right are the input/output signals. Only one machine is active at anytime. Control is passed across the machines based on the computed movement of the read-write head. The inputs *PassLeft, GetRight* and outputs *GetLeft, PassRight* are for handshaking the transfer of control. The next state is passed through *q-in, q-out*.

istic, and communication is only with adjacent machines. It is a surprising and important fact that even such simple systems can have high complexity.

Theorem 10. *CTL model checking a system of interacting finite state machines is PSPACE complete.*

Proof. Observe that state s can reach state t in the product machine if and only if s models the CTL formula $\exists F[\underline{[t]_1} \wedge \underline{[t]_2} \wedge \ldots \wedge \underline{[t]_n}]$; thus reachability reduces CTL model checking a system of interacting machines. From lemma 7 it follows that model checking is PSPACE-hard.
We now demonstrate membership in PSPACE. First consider model checking atomic propositions. $s \models \underline{s^i_j}$ if and only if $[s]_i = s^i_j$ and there exists a fair path starting at state s. Checking $[s]^i = s^i_j$ is trivial. For the second part, i.e. checking that there exists a fair path starting at s, note that there such a path exists if and only if there is a path $(s_1 \ldots s_n) \leadsto \cdots \leadsto (t_1 \ldots t_n)$ to a cycle $(t_1 \ldots t_n) \rightarrow (a_1 \ldots a_n) \leadsto \cdots \leadsto (t_1 \ldots t_n)$ which is fair. The path and cycle can be non-deterministically generated. At each stage it can be checked that the transition is legitimate. Furthermore, the set of infinitary states in each component occurring in the cycle can be stored and used to check that the cycle is fair. This procedure is in NPSPACE; invoking Savitch's theorem it follows that model checking atomic propositions is in PSPACE.

Checking formulae of the form $\neg\psi_1$, or $\psi_1 \vee \psi_2$ requires only constant space over the formulas used to check ψ_1, ψ_2. For existential path formulae i.e. those of the form $\exists X\psi_1$, $\exists G\psi_1$, and $\exists[\psi_1 U \psi_2]$, a path to a fair cycle can be nondeterministically generated as above; each state can be recursively checked for ψ_1 and ψ_2. The additional space used to generate the path is no more than the sum of the number of states in each component. Thus composite formula can be model checked recursively to yield a procedure that runs in NPSPACE. A detailed analysis invoking Savitch's theorem yields a deterministic space bound of $O(|\psi|^2 \cdot (|M_1| + \cdots + |M_n|)^2)$. ■

In some ways the result of theorem 10 is actually a positive one – given the branching nature of CTL one might have expected this problem to be EXPTIME–complete. However, the algorithm for model checking in the proof uses nondeterministic PSPACE; there does not appear to be a direct deterministic algorithm which runs in PSPACE for this problem.

3.2 Language universality

Terminology: Given an alphabet Σ, the class of $*$-languages over Σ is the set of all sets of finite strings over Σ; the class of ω-languages over Σ is the set of all sets of infinite strings over Σ.

Definition 11. Given a product machine M with no external inputs and Büchi fairness conditions on the components, the *language* $\mathcal{L}_M$ of the system is defined to be the set of infinite sequences of symbols from the output space that correspond to fair runs starting at the initial state. The language $\mathcal{L}_M$ will be referred to as *universal* if it consists of <u>all</u> infinite sequences of outputs.

Universality is of fundamental significance – it lower bounds the complexity of *equivalence* (do two product machines generate exactly the same set of output sequences) since universality is a special case of equivalence to a trivial universal machine. It also lower bounds the complexity of *containment* (is the set of output sequences of one machine contained in the set of output sequences of another machine) since universality can be reduced to checking containment of a universal machine.

We will show that deciding universality of language $\mathcal{L}_M$ is EXPSPACE-complete. Membership in EXPSPACE is direct – the product machine contains an exponential number of states; it follows from the work of Sistla, Vardi, and Wolper [19] that the checking a Büchi automaton for universality can be performed in space polynomial in the number of states. EXPSPACE–hardness follows by a reduction from a *word problem* which is known to be EXPSPACE–complete. First we define regular expressions with exponentiation.

Definition 12. Given a finite alphabet $\Sigma = \{a_1, a_2, \ldots, a_n\}$ not containing the symbols $0, 1, \uparrow$, a *regular expression with exponentiation* is a formula derived from the following syntax: ϕ ; ϵ ; $\mathbf{a_i}$; $(r + s)$; $(r \cdot s)$; (r^*) ; $(r \uparrow k)$ where k is a positive integer expressed in binary. Given a regular expression r, the set of finite strings $\mathcal{L}(r)$ defined by r is derived in the usual way; the expression $(r \uparrow k)$ defines the set of all strings that are the concatenation of k elements from the set defined by r.

Stockmeyer [9] proved the following theorem.

Theorem 13. *The problem whether a regular expression with exponentiation denotes all strings over its alphabet is complete for exponential space with respect to polynomial time reduction*

Given a regular expression with exponentiation, one can construct a system of interacting FSMs with fairness conditions, such that the set of output traces of the system is universal if and only if the regular expression is universal. Furthermore the reduction takes time polynomial in the length of the expression. The interesting step in the construction is simulating the exponentiation operator efficiently. This is achieved by using a counter to keep track of the number of concatenations; the key observation is that an n-state counter can be built with $O(\log(n))$ constant sized FSMs.

Theorem 14. *Given a closed system of interacting finite state machines with Büchi fairness conditions, deciding if if the set of output strings generated by the system is universal is complete for EXPSPACE.*

Proof.(Sketch) Membership in EXPSPACE is direct: the transition graph of the product machine is exponential in the number of components, and can be constructed in EXPSPACE. The product is a Büchi automaton over the output alphabet; It follows by the results of Sistla, Vardi, and Wolper [19] that universality for Büchi automaton can be decided in space that is polynomial in the number of states of the automaton (since they demonstrate that complementation of Büchi automata can be performed in PSPACE). Thus universality for interacting finite state machines can be decided in EXPSPACE.

Let r be a regular expression with exponentiation over the alphabet $\Sigma = \{a_1, \ldots, a_n\}$. To demonstrate EXPSPACE-hardness, we will outline an polynomial time construction for a product machine M_r^ω with Büchi fairness conditions on the components such that $\mathcal{L}_{M_r^\omega}$ is universal if an only if $\mathcal{L}(r)$ is universal.

Rather than define M_r^ω immediately, we will first show an efficient construction for a product machine M_r^* on inputs drawn from Σ which defines the set

$\mathcal{L}(r)$ in the following way: the product machine will have a single state designated the *start* state q_s, and a single state designated the *final state* q_f; the start state will have no states making transitions to it, and the final state will have no transitions to other states. A finite string has an accepting run in such a system if it causes a valid sequence of transitions starting at the start state and ending at the final state. The set of accepting strings will be denoted by $\mathcal{L}_{M_r^*}$; we will show this set is precisely $\mathcal{L}(r)$. The systems generated by such a construction are essentially compact representations of non-deterministic finite state automaton [9].

The following is an outline of the construction:

- $r = \mathbf{a_i}$: A machine on two states q_s and q_f with a transition from q_s to q_f on input a_i suffices.
- $r = (r_1 \cdot r_2)$: Let $M_{r_1}^*$ and $M_{r_2}^*$ be machines defining $\mathcal{L}(r_1)$ and $\mathcal{L}(r_2)$; let their start/final states be q_s^1, q_f^1 and q_s^2, q_f^2 respectively. A machine M_r^* for $\mathcal{L}(r)$ can be derived from M_1 and M_2 by running them concurrently and having M_1 idle (i.e. always remain in the same state) after reaching its final state, and M_2 idle at its start state until M_1 reaches its final state. The start/final states of M_r^* are $(q_s^1), q_s^2)$ and (q_f^1, q_f^2).
- $r = (r_1^*)$: Similar to the previous case.
- $r = (r_1 + r_2)$: Let $M_{r_1}^*$ and $M_{r_2}^*$ be machines defining $\mathcal{L}(r_1)$; let their start/final states be q_s^1, q_f^1 and q_s^2, q_f^2 respectively. A machine M_r^* for $\mathcal{L}(r)$ can be derived from M_1 and M_2 in the following way: Add new states p_1 and p_2 to $M_{r_1}^*$ and $M_{r_2}^*$ respectively; at p_1 and p_2 on any input the next states allowed are p_1 or q_f^1, and p_2 or q_f^2 respectively; outputs are arbitrary. At the initial state, nondeterministically make exactly one of $M_{r_1}^*$ or $M_{r_2}^*$ transition to the the corresponding idle state; the rest of the transition structure is unchanged. The start/final states of M_r^* are (q_s^1, q_s^2) and (q_s^1, q_s^2) and (q_f^1, q_f^2).
- $r = (r_1 \uparrow n)$: Let $M_{r_1}^*$ be a machine defining $\mathcal{L}(r_1)$. Run an n state counter in parallel with $M_{r_1}^*$; transitions to the final state of M_1 are allowed only when the counter has counted down n steps. Each time a transition can be made to the final state of $M_{r_1}^*$, add a transition returning to the start state of M_1 and decrement the counter. This construction is outlined in figure 8. Observe that an n-state counter can be built using $O(\log(n))$ constant sized FSMs. Thus the composition of the counter with $M_{r_1}^*$ yields a compact product machine which defines the set $\mathcal{L}(r)$.

It is clear that the set of strings having runs starting at the start state and ending at the final state will be precisely strings in $\mathcal{L}(r)$. Furthermore the

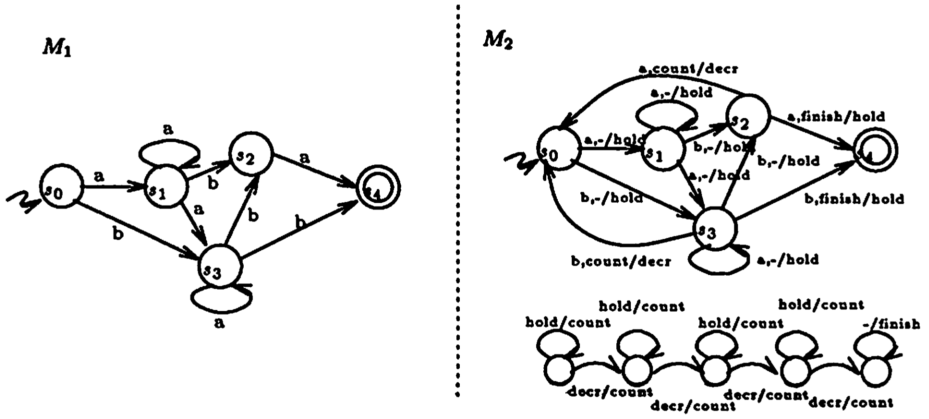

Fig. 8. Use of counter to simulate exponentiation: $\mathcal{L}(M_2) = (\mathcal{L}(M_1)) \uparrow 5$

reduction is in polynomial time.

The above argument indicates that deciding universality for interacting machines on finite strings is EXPSPACE-hard. To push this result over to interacting machines on ω-strings, we use the following observation of Sistla, Vardi, and Wolper [19].

Let $\mathcal{L}$ be a $*$-language on the alphabet $A = \{a_1, \ldots, a_n\}$. Define the new alphabets $A_1 = \{a_1^1, \ldots, a_n^1\}$ and $A_2 = \{a_1^2, \ldots, a_n^2\}$. Let language $\mathcal{L}_1$ be the set of strings of $\mathcal{L}$ with a_k substituted by a_k^1; similarly define $\mathcal{L}_2$. Then the $*$-language $\mathcal{L}$ is universal if and only if the ω-language $\mathcal{L}_\omega$ defined below is universal.

$$\begin{aligned}\mathcal{L}_\omega = \mathcal{L}_1 \cdot \mathcal{L}_2^\omega \,\cup\, \mathcal{L}_2 \cdot \mathcal{L}_1^\omega \,\cup\, (\mathcal{L}_1 \cdot \mathcal{L}_2)^* \cdot \mathcal{L}_1^\omega \,\cup \\ (\mathcal{L}_1 \cdot \mathcal{L}_2)^* \cdot \mathcal{L}_2^\omega \,\cup\, (\mathcal{L}_2 \cdot \mathcal{L}_1)^* \cdot \mathcal{L}_1^\omega \,\cup\, (\mathcal{L}_2 \cdot \mathcal{L}_1)^* \cdot \mathcal{L}_2^\omega \qquad (1)\end{aligned}$$

We can define ω-regular expressions with exponentiation over infinite strings in a manner analogous to the regular expressions with exponentiation in definition 12. By the observation of equation 1, deciding the universality of regular expression with exponentiation can be reduced to deciding the universality of ω-regular expressions with exponentiation. A polynomial time construction analogous similar to the one given for regular expressions with exponentiation yields a system of interacting finite state machines, with Büchi fairness conditions on the components, which defines the language $\mathcal{L}^\omega(r)$. Thus the universality problem for regular expressions with exponentiation can be polytime reduced to universality for interacting FSMs with fairness, demonstrating the EXPSPACE-hardness of universality for interacting FSMs with fairness. ∎

Part of the motivation for formal verification is that it allows designer to check their designs for bugs at an early stage. As an example, consider the top down design methodology proposed by Kurshan [11] as illustrated in figure 9. The design process is a series of refinements, with a set of properties verified at each stage. Properties are specified by task automaton; the language accepted by task automaton defines the set of correct output behaviors.

The refinement step must preserve the properties originally proved; in Kurshan's paradigm, this means that the language of the implementation is contained in the language of the specification. Only a subclass of implementations are allowed, namely those for which a short proof of containment exists. This proof takes the form of structural containment between specification and implementation in the form of language homomorphisms and simulation relations. Since the containment check is EXPSPACE-complete, this methodology is incomplete - there are interesting implementations that cannot be derived via the allowed refinement mechanism. Therefor it will not be possible to find the best implementation for a given specification via this mechanism.

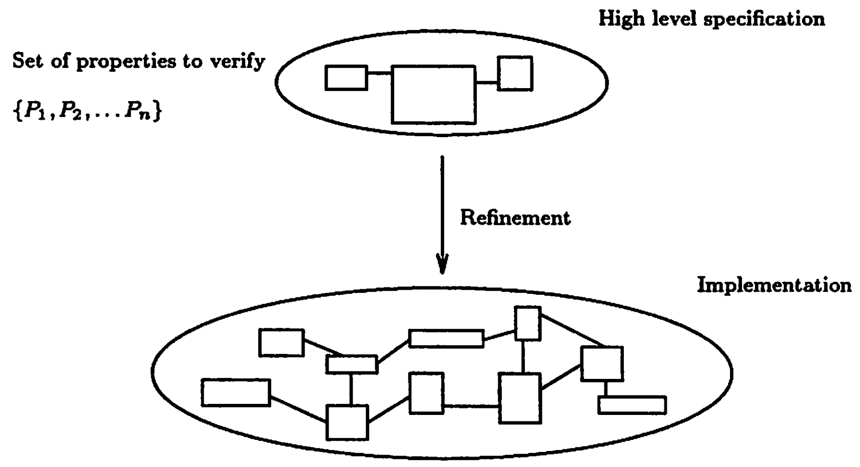

Fig. 9. Top down design methodology - at each stage the designer checks a set of properties; the refinement process preserves all properties previously checked.

4 Conclusion and Future Work

We have defined a simple model of concurrency that is suited to describing synchronous hardware. The complexity of verifying such systems in the CTL paradigm was shown to be PSPACE-complete; for single machines represented in terms of a transition graph verification can be performed in polynomial time. The complexity of deciding trace universality was shown to be EXPSPACE-complete; for single machines universality is PSPACE-complete. Thus our results demonstrate a tradeoff between the ability to compactly describe systems using concurrency, and the increased complexity of analyzing such systems. In the future we plan to study other other models of computation, and compare them in terms of succinctness, expressiveness, and complexity to the interacting FSM model.

References

1. A. Aziz, F. Balarin, R. K. Brayton, S.-T. Cheng, R. Hojati, T. Kam, S. C. Krishnan, R. K. Ranjan, A. L. Sangiovanni-Vincentelli, T. R. Shiple, V. Singhal, S. Tasiran, and H.-Y. Wang. HSIS: A BDD-Based Environment for Formal Verification. In *Proc. of the Design Automation Conf.*, June 1994.
2. A. Aziz and R. K. Brayton. Verifying Interacting Finite State Machines. Technical Report UCB/ERL M93/52, Electronics Research Lab, Univ. of California, Berkeley, CA 94720, July 1993.
3. J. R. Burch, E. M. Clarke, K. L. McMillan, and D. L. Dill. Symbolic Model Checking: 10^{20} States and Beyond. *Information and Computation*, 98(2):142–170, 1992.
4. E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic Verification of Finite-State Concurrent Systems Using Temporal Logic Specifications. *ACM Trans. Prog. Lang. Syst.*, 8(2):244–263, 1986.
5. D. L. Dill, A. J. Hu, and H. Wong-Toi. Checking for Language Inclusion Using Simulation Preorder. In *Proc. of the Third Workshop on Computer-Aided Verification*, 1991.
6. E. A. Emerson. Temporal and Modal Logic. In J. van Leeuwen, editor, *Formal Models and Semantics*, volume B of *Handbook of Theoretical Computer Science*, pages 996–1072. Elsevier Science, 1990.
7. M. R. Garey and D. S. Johnson. *Computers and Intractability.* W. H. Freeman and Co., 1979.
8. O. Grümberg and D. E. Long. Model Checking and Modular Verification. In J. C. M. Baeten and J. F. Groote, editors, *Proc. of CONCUR '91: 2nd Inter. Conf. on Concurrency Theory*, volume 527 of *Lecture Notes in Computer Science.* Springer-Verlag, Aug. 1991.
9. J. E. Hopcroft and J. D. Ullman. *Introduction to Automata Theory, Languages and Computation.* Addison-Wesley, 1979.
10. N. D. Jones, L. H. Landweber, and Y. E. Lien. Complexity of some problems in Petri nets. *Theoretical Computer Science*, 4:277–299, 1977.

11. R. P. Kurshan. Reducibility in Analysis of Coordination. In *Discrete Event Systems: Models and Applications*, volume 103 of *LNCIS*, pages 19–39. Springer-Verlag, 1987.
12. R. P. Kurshan. *Automata-Theoretic Verification of Coordinating Processes*. Princeton University Press, 1993. To appear.
13. D. Lee and M. Yannakakis. Online Minimization of Transition Systems. In *ACM Symposium on the Theory of Computation*, pages 264–274, May 1992.
14. D. E. Long. *Model Checking, Abstraction and Compositional Verification* . PhD thesis, Carnegie Mellon University, Pittsburgh, PA, July 1993.
15. Z. Manna and A. Pneuli. Verification of Concurrent Programs: The Temporal Framework. In R. S. Boyer and J. S. Moore, editors, *The Correctness Problem in Computer Science*, Int. Lecture Series in Computer Science, pages 215–273. Academic Press, London, 1981.
16. K. L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, 1993.
17. A. Rabinovich. Checking Equivalences Between Concurrent Systems of Finite Agents. In W. Kuich, editor, *Proc. Intl. Colloquium on Automata, Languages and Programming (ICALP)*, volume 623 of *Lecture Notes in Computer Science*, pages 696–707. Springer Verlag, July 1992.
18. S. Safra. *Complexity of Automata on Infinite Objects*. PhD thesis, The Weizmann Institute of Science, Rehovot, Israel, Mar. 1989.
19. A. P. Sistla, M. Y. Vardi, and P. L. Wolper. The Complementation Problem for Büchi Automata, with Applications to Temporal Logic. *Theoretical Computer Science*, 49:217–237, 1987.
20. R. J. van Glabbeek. *Comparative Concurrency Sematics and Refinement of Actions*. PhD thesis, Centrum voor Wiskunde en Informatica, Vrije Universiteit te Amsterdam, Amsterdam, May 1990.

This article was processed using the LaTeX macro package with LLNCS style