MINIMIZING INTERACTING FINITE STATE MACHINES

by

Adnan Aziz, Vigyan Singhal, Gitanjali M. Swamy,
and Robert K. Brayton

Memorandum No. UCB/ERL M93/68

9 September 1993

MINIMIZING INTERACTING FINITE STATE MACHINES

by

Adnan Aziz, Vigyan Singhal, Gitanjali M. Swamy,
and Robert K. Brayton

Memorandum No. UCB/ERL M93/68

9 September 1993

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

Minimizing Interacting Finite State Machines

Adnan Aziz Vigyan Singhal Gitanjali M. Swamy
Robert K. Brayton *
Department of Electrical Engineering and Computer Sciences
University of California, Berkeley, CA 94720, USA

Abstract

We address the problem of minimizing collections of interacting finite state machines that arise in the context of formal verification. Typically much of the behavior of the system is redundant with respect to a given property being verified, and so the system can be replaced by substantially simpler representations. These redundancies can be captured by a series of equivalence relations on the state space. Directly minimizing the system requires forming the complete product machine which can be very large. We describe hierarchical procedures that minimize the system with respect to explicit and implicit representations. We present experimental results on some standard verification examples to show that our algorithms allow the product machine to be represented by very small implicit or explicit representations. We conclude with some further directions.

*This research was supported by SRC grant 93-DC-008 and NSF/DARPA Grant MIP-8719546

1 Introduction

We are concerned with the problem of formally verifying properties of a system of interacting finite state machines. There are two popular approaches— model checking using temporal logic [1, 2] and language containment [3, 4]. It has been shown these two approaches are complementary in nature and both have certain advantages[5].

Large designs arising in practice are invariably the product of small interacting finite state machines. Industrial experience indicates that few components, even in large designs, have more than a hundred states[6]. However, taking the product of these components to form the product machine leads to the state explosion problem[7, 8]. Informally, this refers to the fact that given n Finite State Machines (FSMs) $\{M_1, M_2, \ldots, M_n\}$, 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 may have exponential time and space complexity. Coping with the explosion of states has been at the forefront of synthesis and verification. Binary Decision Diagrams [9] are extensively used to implicitly represent states and transition relations. BDD-based methods are widely used for formal verification [10, 11]. However, on designs with many components these standard BDD-based techniques also reach their limit.

The complexity of systems of interacting finite state machines has been investigated [12, 13]. It was proved in [13] that such systems are substantially more difficult to synthesize and verify than single machines. As an example, simply determining if a state is reachable from a given starting state is P-space complete for a product machine, whereas the corresponding problem for a single machine can be solved in linear time.

We propose strategies to cope with this problem of explosion of state space. We multiply the components of the designs iteratively and at each iteration we compute an equivalence relationship on the resulting state state. Using this equivalence relation we reduce the current product. Applying these minimizations iteratively we are able to construct a smaller final product machine and also the intermediate product machines are smaller than observed without these minimizations. We also use the current properties being verified to give us more flexibility in the equivalence relation, and thus smaller product machines. Our approach allows us to utilize information about the environment, to achieve greater minimization. We formulate these computations both for model checking and language containment.

We tested some of our algorithms on some studied examples in literature as well as a few real industrial examples. The results demonstrate the effectiveness of the method.

In section 2 we define finite state machines, and assign semantics to their interaction. In section 3, we describe our method for hierarchical minimization in detail. In particular, we define various

equivalence relations on the state space, and discuss their complexity. Issues related to partitioning sequential systems and using equivalent and reached states to minimize BDD representations of a transition system are discussed. We report our experimental results in section 4, and conclude with future work in section 5.

2 Definitions

In this section we define finite state machines and the semantics of their interaction. The definitions are motivated by the desire to model hardware designs. At the early stages of VLSI design, components may be incompletely specified, and the wires and states may not be encoded. Our definition allows this flexibility. Non-determinism is commonly used to abstract the environment or parts of the designs, and is reflected in the use of transition relations rather than functions to represent the state dynamics.

Definition 1 *A* **finite state machine** *with state space S, inputs I, and output alphabet O is characterized by its transition relation $T \subset S \times I \times S$ and output relation $\Theta \subset S \times O$, as illustrated in Figure 1.*

We allow machines to be non-deterministic and incompletely specified.

At the early stages of a design, it is necessary to abstract the behavior of the some of the components or the environment. Fairness conditions, defined below, are used to disallow portions of the system's infinitary behavior that are inconsistent with the eventual design. A single machine may have more than one fairness constraint.

Definition 2 *A* **Buchi Fairness condition** *on a finite state machine M is a subset of the state space $F \subset S$. Given a state $s_0 \in S$, and an input sequence $i \in I^\omega$, the sequence of states $s \in S^\omega$ is said to be a* **run** *of i if*

- $[s]_0 = s_0$
- $\forall k\, T([s]_k, [i]_k, [s]_{k+1})$

The run is said to be **fair** *if $inf(s) \cap F \neq \phi$, where $inf(s)$ is the set of states in the sequence s that occur infinitely often. An output sequence corresponding to a run s is simply any sequence o such that $\forall k\, \Theta([s]_k[o]_k)$. Basically this definition constrains fair runs to be those which visit certain subsets of the state space infinitely often.*

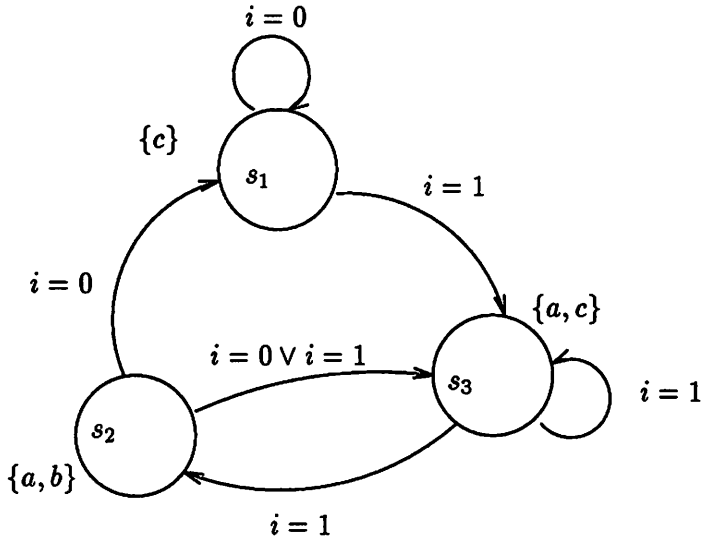

Figure 1: An FSM on states s_1, s_2, s_3, with inputs $0, 1$, and outputs a, b, c.

The fairness constraints imposed above are very simple. However, any ω-regular set can be described by such machines, although there may be other ways of forming the fairness constraints that yield more compact representations. We shall expand on this when equivalences are defined in the presence of Street-Rabin fairness constraints.

Systems are described as a collections of hardware units, communicating through a set of wires, and driven in lockstep by a single clock; this is the basis for the definition of product machine. Consider a system of n interacting machines $M_1, M_2 \ldots, M_n$. We assume there are no external inputs. Any external inputs I_{ext} can be modeled by adding a one state FSM such that this FSM non-deterministically outputs any of the inputs from I_{ext}. Each component machine M_i has present state s^{M_i}, next state variable t^{M_i}, and takes as input $\vec{o}_{ext,i}$ (some subset of outputs of the other machines). Basically, we are dealing with non-deterministic Moore machines with transition predicates on the edges. A run through the product machine is fair if it is fair in each component machine; thus the fair subsets of the product machine are the fair subsets of the component machines lifted to the product state space.

Definition 3 *The* **product machine** $M = M_1 \otimes M_2 \otimes \ldots \otimes M_n$ *is the machine on state space* $S = S^{M_1} \times S^{M_2} \times \ldots \times S^{M_n}$ *and output space* $O = O^{M_1} \times O^{M_2} \times \ldots \times O^{M_n}$, *characterized by*

- *Output relation* $\Theta^M(x^M, o^M) = \prod_{i=1}^{n} \Theta^{M_i}(x^{M_i}, o^{M_i})$
- *Transition relation* $T^M(x^M, y^M) = (\exists \vec{i})[\prod_{k=1}^{n} T^{M_k}(x^{M_k}, i, y^{M_k}) \cdot \Theta^{M_k}(x^{M_k}, o^{M_k}) \cdot (i = [o^{M_1} \ldots o^{M_n}])]$

where the present state variable $x^M = [x^{M_1} x^{M_2} \ldots x^{M_n}]$, *and the next state variable* $y^M = [y^{M_1} y^{M_2} \ldots y^{M_n}]$. *We assume the sets* O^{M_i}, O^{M_j} *are disjoint for all* i, j.

Given an initial state s_0 the FSM as described defines a mapping from input sequences to output sequences:

Definition 4 *The* **behavior** *of the FSM at state* s_0 *is a relation* $\mathcal{B}^{s_0} \subset I^\omega \times O^\omega$, *where the output sequence* σ *is related to* ι *if there is a fair run of the machine starting at state* s_0 *on input* ι *which corresponds to* σ

Observe that a given input sequence may generate no output sequences, or multiple output sequences.

3 Hierarchical Verification Using State Minimization

The intractability results proved in [13] imply the need for heuristics for dealing with the state explosion problem. One heuristic is to incrementally form the product and minimize the incremental machines generated. For example, it is known that states that are bisimulation equivalent are exactly those that satisfy the same set of CTL formulae [14]. Thus one procedure for minimizing the component machines prior to performing CTL model checking would be to compute the bisimulation relation on each component machine and minimize the component state space with respect to this equivalence. Similarly trace equivalence could be used to minimize component machines prior to language containment.

In general the minimization can be hierarchical, i.e. the minimized components can be clustered, and minimization could be performed again. It may be possible to answer the verification problem posed before the product is completely formed, in which case the procedure would return the result. A generic procedure for this form of verification is described below in figure 2

Remark 1: Observe that bisimulation and trace equivalence preserve all CTL and language properties. Thus the equivalence that we get is finer than that defined with respect to a specific property. In fact much of the system dynamics will be irrelevant with respect to the property being checked, and so by using the property to define equivalences on the state space one can get much coarser equivalences. We elaborate on this in section 3.1.

Remark 2: As pointed out in the introduction, large sequential designs sometimes arise as the product of interacting FSMs. Identifying these components in large sequential netlists is a significant problem. A natural way of identifying the components is by partitioning the design (and hence the state space) into interacting machines based on minimizing some cost function. This is discussed further in section 3.2.

```
function incrementally_minimize_and_verify(M_0, ..., M_{n-1}, Property)
        M'_1, ..., M'_l = cluster_and_partition(M_0, ..., M_{n-1}, Property);
        for( i = 0; i < l ; i++ )
                M*_i = minimize(M'_i, Property);
        if fast_pass( M*_1, ..., M*_l )
                return PASS;
        if fast_fail( M*_1, ..., M*_l )
                return FAIL;
return( incrementally_minimize_and_verify(M*_1, ..., M*_l ) );
```

Figure 2: A generic procedure for hierarchical minimization

Remark 3: When designs are represented explicitly, the minimum state machine is the minimum representation for the design. In the case of implicit representations, such as BDDs, there is no correlation between the number of states and the size of the BDD for the transition relation. As a result, it is very difficult to pick a compatible representation. Methods for deriving a small BDD are described in section 3.3.

3.1 State Equivalences

The notion of state equivalence comes up in various contexts. The classical Myhill-Nerode procedure for minimizing DFAs proceeds by computing equivalent states; it then forms the quotient machine which is proved to be the minimum state DFA for the language. Bisimulation equivalence, described in [15], is used in verification to show an implementation satisfies its specification. State equivalence in sequential logic networks is used as don't care information to minimize the associated combinational logic for the next state and output functions.

We will define various equivalences on the state spaces of component machines. Equivalences need to take into account fairness constraints, the nature of the environment, and the property being verified. Coarser equivalences are harder to calculate than finer ones; there is a trade-off between the more succinct representations derived from coarser equivalences, and the added cost of computing them.

In section 3.1.1 we describe equivalent state calculations in the context of language containment for systems of interacting machines; in section 3.1.2 we do the same for model checking.

3.1.1 Equivalence with respect to Output Traces

In this section we develop heuristics for performing language containment. Specifically, given a system of interacting machines $M_1, \ldots, M_n$, possibly with fairness constraints, and a set of outputs $O^{obs} \subset \{o_1, \ldots o_n\}$ being observed, we address the problem of finding small state machines that have equivalent behavior.

Given a machine M and an equivalence relation $\mathcal{E}(s_1, s_2) \subset S^M \times S^M$ on the state space of M, we define the notion of a quotient machine and a quotient state space.

Definition 5 *Given a component machine M, a formula ϕ, and the equivalence relation $\mathcal{E}^\phi(x, y)$, a* **quotient state space** *is a maximal subset of S^M (the state space of M) such no two elements are equivalent modulo $\mathcal{E}^\phi$.*

We will abuse notation and refer to the quotient state space since all such spaces are isomorphic under the natural map derived from $\mathcal{E}^\phi(x, y)$.

Definition 6 *Given a component machine M, a formula ϕ, and the equivalence relation $\mathcal{E}^\phi(x, y)$, a* **quotient machine** *is a machine defined on the quotient state space with transitions from s_1 to s_2 under predicate p iff there is a transition from some $t_1 \in EQ^\phi(s_1)$ to some $t_2 \in EQ^\phi(s_2)$ under predicate p.*

We will abuse notation and refer to the quotient machine since all such machines are isomorphic under the natural map derived from $\mathcal{E}^\phi(x, y)$.

One can define equivalence on the state space of a machine in various ways. The following definition preserves behavior, ie all input/output traces.

Definition 7 *$\mathcal{E}^\beta(s_1, s_2)$ if $\mathcal{B}^{s_1} = \mathcal{B}^{s_2}$*

Lemma 3.1 *Given a machine M and the equivalence relation $\mathcal{E}^\beta(s_1, s_2)$, the quotient machine M' defined on the quotient space $S_{\mathcal{E}^\beta}$ is the minimum state machine that behavior compatible with M*

Proof: If not then there would be a compatible machine M'' with fewer states than there are equivalence classes in $\mathcal{E}^\beta(s_1, s_2)$. By the pigeon-hole principle, two non equivalent states would fall into the same state, contradicting the compatibility of M''. ∎

Lemma 3.2 *Given a machine M and two states s_1, s_2, it it P-space complete to decide if $\mathcal{E}^\beta(s_1, s_2)$*

Proof: Hardness follows from the fact that the Universality problem (that all strings are accepted) for Buchi automaton is P-space complete. The reduction from an instance of universality is straightforward: Assume the Buchi automaton $\mathcal{B}$ has a single start state s_0 (non-deterministic start states can be modelled by adding a single start state which non-deterministically goes to any start state). View the automaton $\mathcal{B}$ as a machine $\mathcal{M}$ with inputs from the alphabet of the Buchi automaton. Take the the accepting states of the Buchi automaton to be a fairness condition on the machine, as described in definition 2. By definition, a string is accepted by the Buchi automaton $\mathcal{B}$ iff there is a run through the automaton which visits accepting states infinitely often. Such a run corresponds to a fair run in the machine $\mathcal{M}$. Thus a string is accepted by the Buchi automaton iff there is a fair run of the string in the corresponding machine. Add a state s_1 which has a self loop on all inputs, which is a fair state. Then the starting state s_0 is equivalent under $\mathcal{E}^\beta$ to s_1 iff the language of the automaton is the universe.

Membership in P-space is analagous to that of Buchi Universality: guess an output string σ and an input string ι show σ is generated by ι at s_1, show that it cannot correspond to any run of s_2 on ι. This procedure requires only polynomial space. ∎

Given an initial state s_0, one can define two states to be equivalent if they can be merged without affecting the behavior at s_0. One can generalize this to multiple initial states by requiring the behavior to be equivalent at all initial states.

Definition 8 *$\mathcal{E}^{\beta,s_0}(s_1,s_2)$ if the machine $M' = (S', I', O', \Theta', T')$ defined below*

- *State space* $S' = (S - \{s_1, s_2\}) \cup \{s_c\}$
- *Input space* $I' = I$
- *Output space* $O' = O$
- *Output relation* $\Theta'(s,o) = \Theta(s,o) \vee ((s = s_c) \wedge (\Theta(s_1,o) \vee \Theta(s_2,o)))$
- *Transition relation*

$$\begin{aligned} T'(x,i,y) \;=\; & T(x,i,y) \vee \\ & ((x = s_c) \wedge (T(s_1,i,y) \vee T(s_2,i,y))) \vee \\ & ((y = s_c) \wedge (T(x,i,s_1) \vee T(x,,is_2))) \end{aligned}$$

has the same behavior at state s_0 as the machine M has at state s_0. The state s_c may taken to be fair to make the behaviors equivalent.

Lemma 3.3 *Given a machine M, a state s_0, and the equivalence relation $\mathcal{E}^{\beta,s_0}$, the quotient machine M' defined on the quotient space $S_{\mathcal{E}}^{\beta,s_0}$ is a minimal state machine which is behavior compatible with M, s_0.*

Proof: Follows from the construction. ∎

Since in general it is hard to minimize the component machines using the equivalence relations defined above, one can use a finer equivalence such as bisimulation to minimize the state space. Care has to be taken to preserve fairness. If only a certain subset of the outputs are observable, one can get coarser bisimulation relations. Bisimulation equivalence can be calculated in polynomial time. Depending on the way the fairness conditions are treated, one gets different bisimulation relations. In [16] simulation relations are defined in the presence of Buchi fairness conditions. These relations match fair states in various ways, and can easily be extended to bisimulation equivalence.

In the definitions below, let O_{Obs} refer to the subset of outputs that are observable.

Definition 9 (Bisim-aa, Büchi fairness) *Let M be a machine. Define states s_1 and s_2 to be bisimulation-aa equivalent ($\mathcal{E}^{bis-aa}(s_1, s_2)$) if*

- $\forall o \in O_{Obs}$, $(\Theta(s_1, o) \leftrightarrow \Theta(s_2, o))$ *and*
- $\forall i, z\,[T(s_1, i, z) \rightarrow \exists z'\,(T(s_2, i, z') \wedge \mathcal{E}^{bis}(z, z'))]$ *and*
- $\forall i, z'\,[T(s_2, i, z') \rightarrow \exists z\,(T(s_1, i, z) \wedge \mathcal{E}^{bis}(z, z'))]$ *and*
- *For each fairness constraint F_k : $s_1 \in F_k \leftrightarrow s_2 \in F_k$, where F_k are the fair subsets.*

Definition 10 (Bisim-lc, Büchi fairness) *Let M be a machine. Define states s_1 and s_2 to be bisimulation-lc equivalent ($\mathcal{E}^{bis-lc}(s_1, s_2)$) if*

- $\forall o \in O_{Obs}$, $(\Theta(s_1, o) \leftrightarrow \Theta(y, o))$ *and*
- $\forall i, z\,[T(s_1, i, z) \rightarrow \exists z'\,(T(s_2, i, z') \wedge \mathcal{E}^{bis}(z, z'))]$ *and*
- $\forall i, z'\,[T(s_2, i, z') \rightarrow \exists z\,(T(s_1, i, z) \wedge \mathcal{E}^{bis}(z, z'))]$ *and*
- *For each fairness constraint F_k : Every cycle through s_1 contains a state from the fair subset F_k $\leftrightarrow$ Every cycle through s_2 contains an fair state from F_k*

Both the above bisimulation relations can be calculated in polynomial time. One can extend the bisimulation relations developed above to machines with fairness constraints expressed as cycle sets, recur edges, Muller acceptance conditions,etc. In particular, consider an automata with Streett

fairness conditions, ie let M be a machine, and $\mathcal{C} = \{(U_1, V_1), \ldots, (U_n, V_n)\}$ be a class of pairs of subsets of the state space. An output sequence $o \in O^\omega$ is generated by the input sequence $i \in I^\omega$ if there is a run $\vec{s} = (s_0, s_1, s_2, \ldots)$ through the state space such that

- s_0 is an initial state of M
- $\forall k\, T(s_k, i_k, s_{k+1}), \Theta(s_k, o_k)$
- $\forall j\, inf(\vec{s}) \cap U_j \neq \phi \rightarrow inf(\vec{s}) \cap V_j \neq \phi$

These conditions define a set of fair paths through the machine. We can extend the definition of bisimulation equivalence to such machines as given above:

Definition 11 (Bisim-aa, Streett fairness) *Let M be a machine with fair paths defined by Streett fairness. Define states s_1 and s_2 to be bisimulation-aa equivalent ($\mathcal{E}^{bis-aa}(s_1, s_2)$) if*

- $\forall o \in O_{Obs}\, (\Theta(s_1, o) \leftrightarrow \Theta(s_2, o))$ *and*
- $\forall i, z\, [T(s_1, i, z) \rightarrow \exists z'\, (T(s_2, i, z') \wedge \mathcal{E}^{bis}(z, z'))]$ *and*
- $\forall i, z'\, [T(s_2, i, z') \rightarrow \exists z\, (T(s_1, i, z) \wedge \mathcal{E}^{bis}(z, z'))]$ *and*
- $\forall j\, [s_1 \in U_j \leftrightarrow s_2 \in U_j \wedge s_1 \in V_j \leftrightarrow s_2 \in V_j]$

Definition 12 (Bisim-lc, Streett fairness) *Let M be a machine. Define states s_1 and s_2 to be bisimulation-lc equivalent ($\mathcal{E}^{bis-lc}(s_1, s_2)$) if*

- $\forall o \in O_{Obs}\, (\Theta(s_1, o) \leftrightarrow \Theta(s_2, o))$ *and*
- $\forall i, z\, [T(s_1, i, z) \rightarrow \exists z'\, (T(s_2, i, z') \wedge \mathcal{E}^{bis}(z, z'))]$ *and*
- $\forall i, z'\, [T(s_2, i, z') \rightarrow \exists z\, (T(s_1, i, z) \wedge \mathcal{E}^{bis}(z, z'))]$ *and*
- *For all cycles C_l through s_1 there are states $s_a, s_b \in C_l$, and $s_a \in U_i$, $s_b \in V_i$, $\leftrightarrow$ for all cycles C_l' through s_2 there are states s_a', s_b' in C_l' such that $s_a' \in U_i$, and $s_b' \in V_i$.*

These too can be computed in polynomial time, using fixed point operations.

The above equivalences have been defined assuming no knowledge about the environment. In general, partial knowledge is available, e.g. it may be known that the set of inputs I_{Cont} that the environment can produce is a subset of the full input space $I_1 \times I_2 \times \ldots \times I_n$. This often arises in closed systems when the environment itself is a collection of machines. The reached state set

in the environment is typically much smaller than the full state space, so large numbers of output patterns will not be generated. These correspond to inputs that will never be seen. In this case during reachability and equivalent state calculations, the inputs that are used to find the reached state set and the equivalent state pair set are taken only from the set I_{Cont}. Similarly, if it is known that certain outputs are equivalent to the environment, then more states can be taken as equivalent. In general, one can further characterize the environment in terms of output sequences that it can generate, and input sequences to it that are equivalent. These will result in equivalences that are coarser, and reached states that are more accurate. Below is the calculation for the **bisim-aa** relation, where I^{Poss} is the set of possible inputs, and $\mathcal{E}^{equiv}$ is the equivalence relation on the output space.

Definition 13 (Bisim-aa, Buchi fairness) *Let M be a machine. Define states s_1 and s_2 to be bisimulation-aa equivalent ($\mathcal{E}^{bis-aa}(s_1, s_2)$) if*

- $\forall o_1 (\Theta(s_1, o_1) \leftrightarrow \exists o_2 \, \Theta(s_2, o_2) \wedge \mathcal{E}^{equiv}(o_1, o_2)) \wedge$
  $\forall o_2 (\Theta(s_2, o_2) \leftrightarrow \exists o_1 \, \Theta(s_1, o_1) \wedge \mathcal{E}^{equiv}(o_1, o_2))$ *and*
- $\forall i \in I^{Cont}, z [T(s_1, i, z) \rightarrow \exists z' (T(s_2, i, z') \wedge \mathcal{E}^{bis}(z, z'))]$ *and*
- $\forall i \in I^{Cont}, z' [T(s_2, i, z') \rightarrow \exists z (T(s_1, i, z) \wedge \mathcal{E}^{bis}(z, z'))]$ *and*
- $F(s_1) \leftrightarrow F(s_2)$

3.1.2 Equivalence with respect to CTL formula

In this section we develop the heuristic procedure outlined in fig 2 for the specific problem of CTL model checking a given formula ϕ over a specified state s in a system of interacting machines $M_1, \ldots, M_n$. Extending the syntax and semantics of CTL for interacting machines is straightforward and described in [13]. Given a machine M, a **closing environment** is any machine M' such that the product machine $M \otimes M'$ forms a closed system. Given a closed system $M_1 \otimes M_2 \otimes \ldots M_n$ the corresponding **Kripke structure** $K^{M_1 \otimes M_2 \otimes \ldots M_n}$ is simply the state transition graph, where the product states are marked by the union of the outputs of the component states. We define an equivalence relation $\mathcal{E}^{\phi}(s_1, s_2)$ on the state space of the component machines and prove that it results in a minimum state machine. Calculating the relation is shown of be P-space hard, and we suggest heuristics for computing a finer relation.

Definition 14 $\mathcal{E}^{\phi}(s_1, s_2) = 1$ *if for every closing environment E and state e in E, $(s_1, e), M \otimes E \models \phi$ iff $(s_2, e), M \otimes E \models \phi$.*

Notation: $EQ^{\phi}(s_1)$ is the set of states forming the equivalence class of s_1

This is a natural definition; it says two states in a component machine are equivalent modulo the formula ϕ iff there are no environments which define structures in which the formula comes out true at one state but not the other. We now show that the quotient machine derived from this equivalence is minimum.

Definition 15 *Let M be a component machine, and ϕ a CTL formula. We say a machine M' defined on the same input-output space as M is ϕ* **compatible** *with M if for every state s in S^M we can associate a state s' in $S^{M'}$ such that for every closing environment E and state $e \in S^E$ $K^{M \otimes E}, (s, e) \models \phi \Leftrightarrow K^{M' \otimes E}, (s', e) \models \phi$*

Lemma 3.4 *The quotient machine is the smallest machine that is ϕ compatible to M.*

Proof:
Analagous to the proof for observability equivalence. ∎

Theorem 3.1 *Given a component machine M, two states s_1 and s_2, and a formula ϕ, it is P-space hard to determine if $\mathcal{E}^{\phi}(s_1, s_2)$*

Proof:[*Sketch:*]
Consider the case when ϕ does not refer to any propositions from the machine M. Then the states s_1 and s_2 are equivalent modulo ϕ if and only if they are behaviorally equivalent, ie if and only if $\mathcal{E}^{\beta}(s_1, s_2)$ as defined in section 3.1.1, which by lemma 3.2 is P-space hard to compute. ∎

As discussed in section 3.1.1, bisimulation equivalence is computable in polynomial time, where as complete equivalences are P-space hard to compute. This motivates the following definition, where states are bisimulation equivalent up to the point where the truth of the given formula is known.

Definition 16 *Given a component machine M_i, a formula ϕ, and two states s, t in S^{M_i} define $\mathcal{E}^{\phi}_{bis}(s, t)$ as follows:*

$$\begin{aligned}
\mathcal{E}^{\phi}_{bis}(s,t) \Leftrightarrow\ & (PASS(s) \wedge PASS(t)) \\
& \vee (FAIL(s) \wedge FAIL(t)) \\
& \vee [\Theta(s,o) = \Theta(t,o) \\
& \wedge \forall i, z\, (T^{M_i}(s,i,z) \rightarrow \exists z'\, T^{M_i}(t,i,z') \wedge \mathcal{E}^{\phi}_{bis}(z,z')) \\
& \wedge \forall i, z'\, (T^{M_i}(t,i,z') \rightarrow \exists z\, T^{M_i}(s,i,z) \wedge \mathcal{E}^{\phi}_{bis}(z',z))]
\end{aligned}$$

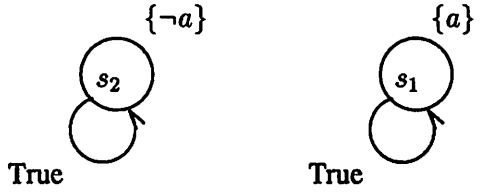

Figure 3: 2 state FSM used to derive formula satifiability

where PASS(x) iff x satisfies ϕ in all environments, and FAIL(x) iff x fails ϕ in all environments.

Lemma 3.5 *Given states s,t $\mathcal{E}^{\phi}_{bis}(s,t) \Rightarrow \mathcal{E}^{\phi}(s,t)$*

Proof:
States that are bisimilar satisfy exactly the same set of CTL formulae[14]. Observe that s and t are bisimilar up to the point where the formula fails or passes. Thus they can not be differentiated by any CTL formula until they reach states at which ϕ must pass or must fail, and hence ϕ will not differentiate them.

■

Remark: Given PASS(x) and FAIL(x) one can devise a fast fixed point calculation for computing $\mathcal{E}^{\phi}_{bis}(s,t)$ using the above theorem.
Calculating $\mathcal{E}^{\phi}_{bis}(s,t)$ turns out to be P-space hard. We show this by a reduction from CTL-satisfiabilty, which is known to be P-space complete.

Theorem 3.2 *Given a component machine M, two states s_1 and s_2, and a formula ϕ, it is P-space hard to determine if $\mathcal{E}^{\phi}_{bis}(s_1,s_2)$*

Proof: Let ϕ be a given CTL formula. Let a be an atomic proposition not appearing in ϕ. Define ϕ^* to be $a \wedge \phi$. Let M be a single two-state machine as shown in figure 3. Then for no closing environment E and state e can $(s_2,e), M \otimes E \models a \wedge \phi$. We now show that ϕ is satisfiable $\Leftrightarrow \neg\mathcal{E}^{\phi}_{bis}(s_1,s_2)$.
($\Rightarrow$) Suppose ϕ is satisfiable. Let K be the Kripke structure, and s_K the state in K such that $s_K, K \models \phi$. Then treating K as a closing environment for M, we see $(s_1,s_K), M \otimes E \models a \wedge \phi$. Thus K serves to differentiate s_1 and s_2, and so $\neg\mathcal{E}^{\phi}(s_1,s_2)$.
($\Leftarrow$) Suppose $\neg\mathcal{E}^{\phi}(s_1,s_2)$. Since $\forall E, e \in S^E, ((s_2,e), M \otimes E) \not\models a \wedge \phi$, it follows that $\exists E, e \in S^E, ((s_1,e), M \otimes E) \models a \wedge \phi$. But then ϕ is satisfiable: the Kripke structure derived from $M \otimes E$ and the state (s_1,e) satisfy ϕ.

Thus given a CTL formula, we can reduce the problem of deciding satifiability of the formula to checking state equivalence. Furthermore this reduction can be performed in polynomial time and so the equivalence calculation is P-space hard. ■

Computing $\mathcal{E}^{\phi}_{bis}(s_1, s_2)$ is P-space hard because formula satisfiability is P-space hard. As most formulae are short, one may still be able to compute the relation relatively efficiently. In appendix A, we give a set of fixed point calculations for $\mathcal{E}^{\phi}_{bis}$ which would be efficient enough when ϕ is simple.

3.2 Partitioning and Sequencing

As pointed out in the introduction, large sequential designs sometimes arise as the product of interacting FSMs. Identifying these components in large sequential netlists is a significant problem. A natural way of identifying the components is to find a balanced partition of the latches and gates so as to minimize the communication across the partition.

The design may be already specified as a collection of communicating processes. In this case a natural decomposition may be the initial processes. In general, it may be preferable to collapse the collection, and then partition the flattened machine. For example, if the property were specified as an automaton, a partition which has smaller clusters "near the task" and coarser ones further away would yield smaller representations.

Given a decomposition of the design, there are various sequences in which the minimization can proceed. One possibility is a balanced decomposition, where a tree of processes is formed, and the minimization proceeds from the leaves to the root. Another approach is to incrementally form the product by multiplying in one machine at a time and minimizing the incremental product at each stage. This is essentially the method we followed. We start with the component machine specifying the property, and multiply in the other machines one at a time in the order of their distance from the task in the communication graph.

Partitioning and sequencing strategies are described in detail in Appendix B.

3.3 BDD Techniques

Given a machine M, let R and $\mathcal{E}$ be the reached state and equivalent state pair sets. If the representation for M were explicit, the quotient machine on the reached state set would be the minimum representation of a machine equivalent to M. However we are interested in BDD based representations. The BDD for a subset of $S \subset \{0,1\}^n$ bears no correlation to the cardinality of S. Therefore, in the spirit of [8] we derive maximal don't care sets which we use to simplify the BDD representing the STG, using some new BDD minimization techniques[17].

Caveat: Minimizing the BDD for component machines doesn't necessarily mean that the product of these BDDs will be smaller than the product of the original BDDs. Also, smaller BDDs for the transition relation do not imply that verification will proceed faster. However experimental evidence indicates this is true. The equivalent state pairs and reached state set provide all the flexibility in

choosing representations for the system, so with the development of even better understanding of BDD minimization for reached state analysis, these calculations will continue to be significant.

3.3.1 Bounds on the STG

Let

- $T(x,i,y)$ be the transition relation of a given FSM
- $R(x)$ be the set of reachable states
- $\mathcal{E}(x,y)$ be an equivalence relation on the state space
- $C(x)$ be a set of representative states derived from $\mathcal{E}$

Consider all transitions that may be added to the FSM based on equivalence between two states. If there exists a transition between two states x and z, then we may add the corresponding transition (on the same set of inputs) between all other states equivalent to x and equivalent to z. By the definition of equivalence, adding in these transitions will in no way change the behavior of the machine. The set of transitions which may be thrown in based on equivalence relations between states is given by $\tilde{T}(x,i,y)$:

$$\tilde{T}(x,i,y) = \exists w(T(w,i,y)\cdot\mathcal{E}(x,w)) + \exists z(T(x,i,z)\cdot\mathcal{E}(y,z))$$

Lemma 3.6 *The following transition relation is minimum:*

$$\begin{aligned} T_{min} &= \tilde{T}(x,i,y)\cdot C(x)\cdot C(y) \\ &\quad \wedge \exists w,z\,[R(w)\wedge R(z)\wedge\mathcal{E}(x,w)\wedge\mathcal{E}(y,z)] \end{aligned}$$

The following transition relations are maximal

$$\begin{aligned} T_{max\text{-}1} &= \tilde{T}(x,i,y)\wedge R(x)\wedge R(y) + \bar{R}(x) \\ T_{max\text{-}2} &= \tilde{T}(x,i,y) + \bar{R}(x)\wedge\neg(\exists y\mathcal{E}(x,y)\wedge R(y)) \end{aligned}$$

In general, $T_{\text{max-1}}$ and $T_{\text{max-2}}$ are only maximal. In some cases it may be possible to add more transitions if some transitions are deleted.This requires making some unreachable states reachable,

and the space of compatible representations becomes highly discontinuous. In the absence of unreachable states, $T_{\text{max-1}}$ equals $T_{\text{max-2}}$, and they are in fact both maximum.

The above expressions give maximal and minimal bounds on the transition structure that preserve behavioral equivalence. Given the range of BDDs possible for the transition relation, our objective at this stage is to compute a small BDD within that range. There is no known efficient method to solve this problem. Several heuristic methods exist, among the most common are the constrain and restrict heuristics[18]. We use a version of this heuristic[17]. Given a function and a care set, it returns a BDD between $f \cdot c$ and $f + c$, which is chosen to have a small size. Thus given BDDs of the minimal and maximal set, we use the following expression to compute an intermediate BDD of minimal size.

$$\begin{aligned} f &= f_{min} \\ c &= f_{min} + \neg f_{max} \end{aligned}$$

It is important to keep in mind that the choice of C, the set of representative elements, is not unique. We have not analyzed this problem in detail.

An important observation to keep in mind is the fact that adding (and subtracting) in the transitions can actually change a deterministic finite state machine into a non-deterministic one. This may pose problems since some problems in sequential synthesis are not yet solvable for non-deterministic machines. However there are a large class of problems which are solvable for non-deterministic machines and for those problems, our results are useful.

4 Experiments and Results

The techniques described in this paper help build the product machine for a system of interacting component machines. We form the product machine iteratively, and minimize the product at each step of the iterative composition. We start with the component machine specifying the property (the **task automaton**), and multiply in the other machines in order of their distances from the task. Equivalence minimization and don't care information for unreachable states is used for this minimization as described above.

The objective is to achieve a smaller representation for the composed product system. This representation is smaller both in the implicit form (size of the BDDs of the final and intermediate product machines) and in the explicit form (number of representative states in the final and intermediate product machines). These smaller representations should allow us to do formal verification

on larger systems than before.

The experiments described here are based on computing the **bisim-aa** equivalence described in Section 3.1.1. We arrange the components of the interacting machine to form an incremental decomposition (as described in Appendix B.) Since our experiments pertain to the language containment model, we start with the component which represents the property. Then we sequence the other components in the order in which they are connected to the property as described in the appendix.

We ran our experiments on a series of benchmarks described in Table 1. For each example we compute statistics for both the explicit and the implicit representations.

For explicit representations, we report the total number of states for the product machine, the number of reached states in the product machine, and the maximum number of reached states seen at any stage of forming the incremental product. We also report the number of non-equivalent states in the final product and the maximum number of distinct states seen at any stage of the algorithm when the component specifying the property is included in the system. Note that in several cases (*dynachek3, p6.safe, p10.safe*), when we are verifying a safety property there is only one distinct state at the end of the calculation. This is happens because the property holds, and so the component machine specifying the property always remains in the safe state. We also ran experiments to compute the number of distinct states when the property is not used, i.e. all outputs were observable. In most cases, we were unable to compute the equivalence relation due to spaceouts, and the when we did the number of equivalence classes is much larger than when the property is used.

Looking at the number of non-equivalent states it is obvious that the true state complexity of all these examples is much smaller than the apparent state complexity, i.e. the total number of states. This is dramatically illustrated by *2mdlc.0* which has 2.31×10^{19} states, of which only 12 are distinct. The enormous number of states in this design arises from the fact that it contains a 40 bit shift register. The property being verified turns out to be independent of the state of the register, and thus all the states in the register are equivalent when viewed from the task.

For implicit methods we report the size of the BDD for the transition relation of the original product machine, and the largest BDD seen while forming this product. We then used the reached state sets that are incrementally generated to minimize the BDDs for the product at each stage; this yields substantial reductions in many cases. Finally we report the reductions obtained when both equivalent state pairs and reached states are used minimize the BDD for the transition relation at each iteration.

To minimize the BDDs using don't care information, we use techniques in [17] (these are based on methods described in [18]). These techniques are currently not very robust (for example, sometimes

using a strictly larger don't care set for the same representation set, the resulting BDD is larger than the one obtained by using the smaller don't care set). Improved BDD minimization techniques will enhance our results.

In Table 3 we show the improvement in BDD sizes of the product machines using our equivalence computations. Some improvements are modest; however, even these modest improvements in BDD sizes affect the reachability computations greatly. Table 2 shows the number of representative reachable states for the examples. In some cases we were unable to perform the reached state and equivalent state calculations for the product machine because of spaceouts, but were able to do so when the product was formed incrementally (for example, for *sched5, 2mdlc.0*). This demonstrates the advantage of forming the product incrementally, and minimizing at each step. We observed that even small improvements in BDD sizes for the transition relations can cause some reachability computations to finish that would otherwise require infeasibly large amounts of memory. For safety properties when the task is specified as a machine that is part of the hierarchy, verification can be performed by reachability analysis. So for these properties our equivalence and reached state minimizations help us verify properties which could not verify otherwise. For liveness properties also, reachability is an important step in the verification algorithm and large improvements in doing the reachability computations certainly would improve the ability to verify larger systems. Forming the product incrementally uses less memory but may take more time since in some cases the original product machine may be tractable, so there is a space-time tradeoff.

Example	Description
p6.safe	Dining philosopher protocol, 6 diners, checking mutual exclusion
p6.liv	Dining philosopher protocol, 6 diners, hungry $\rightarrow$ always get to eat
p10.safe	Dining philosopher protocol, 10 diners, checking mutual exclusion
p10.liv	Dining philosopher protocol, 10 diners, hungry $\rightarrow$ always get to eat
dynachek3	Railway controller, ensure trains never collide
sched5.safe	Milner's job shop scheduler, 5 jobbers, jobs always proceed in correct sequence
sched5.liv	Milner's job shop scheduler, 5 jobbers, jobber always eventually gets hammer
2mdlc.0	Part of a data-link controller from an industrial design, safety property
2mdlc.1	Part of a data-link controller from an industrial design, safety property

Table 1: Verification Examples

Example	$\mid \mathcal{S} \mid$	$\mathcal{R}^{prod}$	$\mathcal{R}^{max}$	$\mathcal{E}^{prod}_{prop}$	$\mathcal{E}^{max}_{prop}$	$\mathcal{E}^{prod}_{no-prop}$	$\mathcal{E}^{max}_{no-prop}$
p6.safe	8,192	78	164	1	84	78	164
p6.liv	8,192	131	268	28	144	131	268
p10.safe	2,097,152	890	9,136	1	1,152	*spaceout*	*spaceout*
p10.liv	2,097,152	1,691	14,928	37	1,920	*spaceout*	*spaceout*
dynachek3	65,536	22,488	31,744	1	33	824	1576
sched5.safe	4,194,304	33,408	63,400	1,398	11,004	*spaceout*	*spaceout*
sched5.liv	4,194,304	33,720	33,720	1,492	6,084	*spaceout*	*spaceout*
2mdlc.0	2.31×10^{19}	1.12×10^{18}	1.12×10^{18}	12	24	*spaceout*	*spaceout*
2mdlc.1	2.31×10^{19}	1.12×10^{18}	1.12×10^{18}	3	6	*spaceout*	*spaceout*

Table 2: Results on state reduction

$\mid \mathcal{S} \mid$: Number of states in product machine

$\mathcal{R}^{prod}$: Number of reached states in product machine

$\mathcal{R}^{max}$: Maximum number of reached states seen at any stage of forming incremental product

$\mathcal{E}^{prod}_{prop}$: Number of non-equivalent reached states in the product machine

$\mathcal{E}^{max}_{prop}$: Maximum number of non-equivalent reached states seen at any stage of forming incremental product

$\mathcal{E}^{prod}_{no-prop}$: Number of non-equivalent reached states, not using property

$\mathcal{E}^{max}_{no-prop}$: Maximum number of non-equivalent reached states, not using property, seen at any stage of forming incremental product

Example	T_{prod}	T_{max}	$T^{prod}_{reached}$	$T^{max}_{reached}$	$T^{prod}_{reached,equiv}$	$T^{max}_{reached,equiv}$
p6.safe	404	1950	116	1240	15	909
p6.liv	427	1853	247	1240	178	909
p10.safe	836	5659	252	3563	23	1679
p10.liv	887	5562	583	3563	342	1679
dynachek3	204	552	204	552	18	582
sched5.safe	657	2544	499	2508	499	2508
sched5.liv	496	1752	425	1378	424	1398
2mdlc.0	6,621	17,336	3,951	13,268	462	1,888
2mdlc.1	6,621	17,336	3,951	13,268	44	91

Table 3: Results on BDD reduction

T_{prod} : Size of unreduced transition relation

T_{max} : Maximum size of any BDD in forming the product; no minimization

$T^{prod}_{reached}$: Size of transition relation reduced with respect to the reached state set only

$T^{max}_{reached}$: Maximum size of any BDD in forming the product; minimization using only unreached states as don't cares

$T^{prod}_{reached,equiv}$: Size of transition relation reduced with respect to the reached and equivalent state sets

$T^{max}_{reached,equiv}$: Maximum size of any BDD in forming the product; minimization using both reached state and equivalent state set

5 Conclusion and Future Work

An efficient way of expressing and computing property verification through language containment is to use Streett automata to express the fairness constraints and the property specifications [19]. We intend to implement the equivalence relation computations for Streett automata and evaluate their effectiveness. We also hope to investigate the effectiveness of the more general equivalence definitions (**bisim-lc** in Section 3.1). A large body of literature exists on synthesizing interacting finite state machines [20, 21]; these results should provide a mechanism for approximating the environment and thus allow more minimization.

This paper has also described how state equivalences can be computed if the properties are expressed as CTL formulas, generalizing the results of [8]. We would like to experiment with such properties. Note that, for the more difficult problem of Fair-CTL model checking both equivalence computations (on fair cycles described in Section 3.1.1, and in CTL formulas described in Section 3.1.2) can be used in conjunction with each other. Our experiments in the previous section indicate the effectiveness of the equivalence computations for fairness.

It is widely believed that verification procedures that require explicit enumeration of the state space are not feasible because of the state explosion problem; this is the primary reason for the advent of BDD based techniques. The observation that the true state complexity of most large systems is much smaller than the size of the state space coupled with the fact that the state graph can be built incrementally motivates the need for a closer look at this premise. We plan to investigate this point in detail.

The explosion of states is often due to the presence of a data path and in many cases the

property being verified has only a partial dependancy on the actual values of the registers. Thus, state equivalence can be used to automatically simplify the datapath. We are working on techniques for abstraction that utilize user specified equivalences to minimize the system representation.

References

[1] Z. Manna and A. Pneuli, "Verification of Concurrent Programs: The Temporal Framework," in *The Correctness Problem in Computer Science* (R. S. Boyer and J. S. Moore, eds.), Int. Lecture Series in Computer Science, pp. 215–273, London: Academic Press, 1981.

[2] E. M. Clarke, E. A. Emerson, and A. P. Sistla, "Automatic Verification of Finite-State Concurrent Systems Using Temporal Logic Specifications," *ACM Trans. Programming Languages and Systems*, vol. 8, no. 2, pp. 244–263, 1986.

[3] M. Y. Vardi and P. L. Wolper, "An Automata-Theoretic Approach to Program Verification," in *Proc. of the IEEE Symposium on Logic in Computer Science*, pp. 332–334, 1986.

[4] R. P. Kurshan, "Reducibility in Analysis of Coordination," in *Discrete Event Systems: Models and Applications*, vol. 103 of *LNCIS*, pp. 19–39, Springer-Verlag, 1987.

[5] R. Hojati, T. R. Shiple, R. K. Brayton, and R. P. Kurshan, "A Unified Environment for Language Containment and Fair CTL Model Checking," in *Proc. of the Design Automation Conf.*, (Dallas, Texas), June 1993.

[6] G. York. Cadence Design Systems. Personal communication, 1993.

[7] O. Grümberg and D. E. Long, "Model Checking and Modular Verification," in *Proc. of CONCUR '91: 2nd Inter. Conf. on Concurrency Theory* (J. C. M. Baeten and J. F. Groote, eds.), vol. 527 of *Lecture Notes in Computer Science*, Springer-Verlag, Aug. 1991.

[8] M. Chiodo, T. R. Shiple, and A. L. Sangiovanni-Vincentelli, "Automatic Compositional Minimization in CTL Model Checking," in *Proc. of the Intl. Conf. on Computer-Aided Design*, pp. 172–178, 1992.

[9] R. Bryant, "Graph-based Algorithms for Boolean Function Manipulation," *IEEE Trans. Computers*, vol. C-35, pp. 677–691, Aug. 1986.

[10] J. R. Burch, E. M. Clarke, K. L. McMillan, and D. L. Dill, "Symbolic Model Checking: 10^{20} States and Beyond," in *Proc. of the IEEE Symposium on Logic in Computer Science*, 1990.

[11] H. Touati, R. K. Brayton, and R. P. Kurshan, "Checking Language Containment using BDDs," in *Proc. of Intl. Workshop on Formal Methods in VLSI Design*, (Miami, FL), Jan. 1990.

[12] A. Rabinovich, "Checking Equivalences Between Concurrent Systems of Finite Agents," in *Proc. of the Intl. Colloquium on Automata, Languages and Programming (ICALP)* (W. Kuich, ed.), vol. 623 of *Lecture Notes in Computer Science*, pp. 696–707, Springer Verlag, July 1992.

[13] A. Aziz and R. K. Brayton, "Verifying Interacting Finite State Machines," Tech. Rep. UCB/ERL M93/52, Electronics Research Lab, Univ. of California, Berkeley, CA 94720, July 1993.

[14] M. C. Browne, E. M. Clarke, and O. Grümberg, "Characterizing Finite Kripke Structures in Propositional Temporal Logic," *Theoretical Computer Science*, vol. 59, pp. 115–131, 1988.

[15] R. Milner, *Communication and Concurrency*. New York: Prentice Hall, 1989.

[16] 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.

[17] T. R. Shiple, R. Hojati, A. L. Sangiovanni-Vincentelli, and R. K. Brayton, "Heuristic Minimization of BDDs Using Don't Cares," Tech. Rep. UCB/ERL M93/58, Electronics Research Lab, Univ. of California, Berkeley, CA 94720, July 1993.

[18] O. Coudert and J. C. Madre, "A Unified Framework for the Formal Verification of Sequential Circuits," in *Proc. of the Intl. Conf. on Computer-Aided Design*, pp. 126–129, Nov. 1990.

[19] R. Hojati, V. Singhal, and R. K. Brayton, "Edge-Streett/Edge-Rabin Automata Environment for Formal Verification Using Language Containment," Tech. Rep. SRC Pub C93284, Semiconductor Research Corp., 79 Alexander Dr., Building 4401, Suite 300, P. O. Box 12053, Research Triangle Park, NC 27709, 1993.

[20] J.-K. Rho, G. Hachtel, and F. Somenzi, "Don't Care Sequences and the Optimization of Interacting Finite State Machines," in *Proc. of the Intl. Conf. on Computer-Aided Design*, pp. 418–421, Nov. 1991.

[21] Y. Watanabe and R. K. Brayton, "The Maximum Set of Permissible Behaviors for FSM Networks," in *Proc. of the Intl. Conf. on Computer-Aided Design*, 1993.

A State Equivalence for CTL Formulae

Definition 17 *Equivalence of states modulo a formula* $\mathcal{E}^{\phi}_{bis}(x,y)$

1. $\phi = a$ *where* a *is an output of another component machine. Then* $\mathcal{E}^{\phi}_{bis}(x,y)$ *iff 1*
2. $\phi = a$ *where* a *is a symbol of the output alphabet of* M*. Then* $\mathcal{E}^{\phi}_{bis}(x,y)$ *iff* $\Theta(x,a) \leftrightarrow \Theta(y,a)$
3. $\phi = \neg\psi$ *Then* $\mathcal{E}^{\phi}_{bis}(x,y)$ *iff* $\mathcal{E}^{\psi}_{bis}(x,y)$
4. $\phi = \exists X\psi$ *Then* $\mathcal{E}^{\psi}_{bis}(x,y)$ *iff* $\forall i,z[T^M(x,i,z) \rightarrow \exists z' T^M(y,i,z') \wedge \mathcal{E}^{\psi}_{bis}(z,z')$ *and* $\forall i,z[T^M(y,i,z) \rightarrow \exists z' T^M(x,i,z') \wedge \mathcal{E}^{\psi}_{bis}(z,z')$
5. $\phi = \exists X\psi$ *Then* $\mathcal{E}^{\psi}_{bis}(x,y)$ *iff* $\forall i,z[T^M(x,i,z) \rightarrow \forall z' T^M(y,i,z') \rightarrow \mathcal{E}^{\psi}_{bis}(z,z')$ *and* $\forall i,z[T^M(y,i,z) \rightarrow \forall z' T^M(x,i,z') \rightarrow \mathcal{E}^{\psi}_{bis}(z,z')$
6. $\phi = \exists aUb$ *Then* $\mathcal{E}^{\psi}_{bis}(x,y)$ *may be computed as follows:*
   First we compute $NR(x) = \{x \mid \nexists$ *a path in* M *from* x *to a state with output* $b\}$

$$\begin{aligned} NR_0(x) &= \neg\Theta(x,b) \\ NR_{i+1}(x) &= NR_i \wedge \forall i,z[T(x,i,z) \rightarrow NR_i(z)] \\ NR(x) &= NR_\infty(x) \end{aligned}$$

Next compute $FAIL(x)$ *which is the set of states in* M *which are guaranteed to fail* ϕ

$$\begin{aligned} FAIL_0(x) &= \neg\Theta(x,a) \wedge \neg\Theta(x,b) \vee NR(x) \\ FAIL_{i+1}(x) &= \neg\Theta(x,b) \wedge \forall i,z[T(x,i,z] \rightarrow FAIL_i(x)] \vee FAIL_i(x) \\ FAIL(x) &= FAIL_\infty(x) \end{aligned}$$

Then compute $PASS(x)$ *which is the set of states in* M *which are guaranteed to pass* ϕ

$$PASS(x) = \Theta(x,b)$$

The set $EQ(x,y)$ *is the relation "Both* x *and* y *are guaranteed to fail, or both are guaranteed to pass", it is formally defined as*

$$EQ(x,y) = FAIL(x) \wedge FAIL(y) \vee PASS(x) \wedge PASS(y)$$

We can now compute the relation $NE(x,y)$ which is the negation of $\mathcal{E}^{\psi}_{bis}(x,y)$

$$\begin{aligned} NE_0(x,y) &= FAIL(x) \oplus FAIL(y) \vee PASS(x) \oplus PASS(y) \\ NE_{i+1}(x,y) &= NE_i(x,y) \vee [\exists o[\Theta(x,o) \oplus \Theta(y,0)] \wedge \neg EQ(x,y)] \\ &\quad \vee \exists i \forall z, z'[T(x,i,z) \wedge T(y,i,z') \rightarrow NE_i(x,y)] \end{aligned}$$

7. *$\phi = \forall a U b$ Then $\mathcal{E}^{\psi}_{bis}(x,y)$ may be computed as above.*

B Partitioning and Sequencing

B.1 Partitioning

As pointed out in the introduction, large sequential designs arise as the product of interacting FSMs. Identifying these components in large sequential netlists is a significant problem. A natural way of identifying the compone nts is by partitioning the latches and gates so as to minimize a cost function. Our premise is the following:

- Latches corresponding to a single component will have a dense communication structure
- Communication across components is sparse

Based on this, we propose the following cost function as a measure of the partition:

$$\text{cost}(\mathcal{P}) = \sum_{P_i \in \mathcal{P}} (\text{in}(P_i) + \text{out}(P_i))$$

where $\text{in}(P_i)$ is the number of distinct bits of input to P_i and $\text{out}(P_i)$ is the number of distinct bits of output from P_i. In order that the partition be non-trivial, we further put the constraint that in a k-way partition no component has more than cn/k latches, where n is the number of latches in the network, and $c \geq 1$ is an arbitrarily chosen constant. Typically k will be chosen so that each component is as large as possible, while still being tractable for manipulation, so as to derive the most don't care information at each component.

One can refine this cost function further. Some possible extensions are:

- Use cost functions that weigh the partition towards keeping the number of latches in a partition balanced e.g. subtract a term $\sum_{P_i \in \mathcal{P}} (l(P_i) - n/ \mid \mathcal{P} \mid)^2$ from the cost function. Here $l(P_i)$ is the number of latches in partition P_i.
- Introduce dependencies on k to evaluate partitions of different sizes e.g. weigh the cost toward having k as small as possible while ensuring component machines are tractable

However, the partitioning problem is already algorithmically intractable. For this reason we restrict our attention to cost($\mathcal{P}$) defined above which is simple to calculate.

However, the partitioning problem is already algorithmically intractable. For this reason we restrict our attention to cost($\mathcal{P}$) defined above which is simple to calculate.

The design may be already specified as a collection of communicating processes. In this case a natural decomposition may be the initial processes. In general, it may be preferable to collapse the collection, and then partition the flattened machine. For example, if the property were specified as an automaton, a partition which has smaller clusters "near the task" and coarser ones further away would yield smaller representations.

B.1.1 Sequencing

The following theorem relates BDD size for the transition relation of a collection of interacting machines to the communication structure of the system.

Theorem B.1 *Let M be a system of n interacting machines $M_1, M_2, \ldots, M_n$, and σ a permutation on $\{1, 2, \ldots, n\}$. Then the number of nodes in the BDD for $T = \prod_{i=1}^{n} T_I$ under the non-interleaved ordering (i.e. all variables associated with M_i are contiguous) derived from σ is bounded by*

$$U^{\sigma} = \sum_{i=1}^{n} (2^{2(|x_{\sigma(i)}|)} \cdot 2^{w_f^{\sigma}(i+1)} \cdot 2^{2^{w_r^{\sigma}(i+1)}}) \tag{1}$$

where $| x_{\sigma(i)} |$ is the number of bits in the encoding of the state of M_i, $w_f^{\sigma}(i)$ and $w_r^{\sigma}(i)$ are the number of bits communicated forward and backwards through the partition σ at level i.

Experimental evidence indicates that this bound correlates well with the size of the BDD.

Let σ^* be a permutation on the machines that minimizes the bound. Then based on σ^* we can perform traversal and minimization according to the following two schedules:

- Balanced Decomposition
  - If the machine has a single component, perform standard optimizations and return the result; else
  - Split M into two machines $P_1 = \{M_{\sigma(1)} \times \ldots \times M_{\sigma(\lfloor (n/2) \rfloor)}\}$ and $P_2 = \{M_{\sigma(\lceil (n/2) \rceil} \times \ldots \times M_{\sigma(n)}\}$
  - Recursively traverse and minimize the machines P_1 and P_2
  - Traverse and minimize the product $P_1 \times P_2$
- Incremental Decomposition
  - Form $P = M_{\sigma(1)} \times M_{\sigma(2)}$; traverse and minimize P to obtain P^*
  - Recursively complete the traversal/minimization on $\{P^*, M_{\sigma(3)}, \ldots, M_{\sigma(n)}\}$

At first glance, balanced decompositions may appear to be superior. However, in the context of verification using task automata, incremental decomposition has the advantage that the task can be taken as the initial machine, and then at each stage the minimization is with respect to the property. Since much of the design is redundant with respect to the property, the incremental product will be smaller.