Don't Cares in Logic Minimization of Extended Finite State Machines

Yunjian Jiang Robert K. Brayton
Department of Electrical Engineering and Computer Sciences
University of California, Berkeley, CA 94720-1772
e-mail: {wjiang, brayton}@eecs.berkeley.edu

Abstract— Extended Finite State Machines (EFSMs) have been proposed to model control oriented systems. A version of this, with the data portion modeled by Presburger arithmetic, has been used in formal verification and test pattern generation. This paper proposes a general logic minimization scheme using don't care derived from both control and data path. It consists of methods to transfer don't cares through the data path and to generate logic don't cares from the data path using quantifier-free Presburger inequalities. Potential applications are discussed and preliminary results validate the scheme on reasonable examples.

I. Introduction

Extended Finite State Machines (EFSMs) have been studied for system level design modeling and synthesis. An EFSM is a system with a finite state controller interacting with an unbounded integer data path. Each transition of the controller is guarded by a predicate over the Boolean and integer variables, and associated with an action function which updates the new values of the integer variables. The earliest work on EFSMs can be traced back to Glushkov [13], where he studied the theory of digital machines controlled by finite automata. More recently, EFSMs appear frequently, although in different forms and flavors, in various contexts of VLSI designs, e.g. verification and reachability analysis in [10], symbolic model checking in [8], testing in [9], hardware-software codesign in the POLIS project [2, 3] and software synthesis [17].

Often, the predicates and action functions of a design are definable in Presburger arithmetic, a decidable subset of the general Peano arithmetic in number theory [11], excluding multiplication. Presburger formulas consist of natural number constants, natural number variables, addition, equality, inequality and first order logical connectives. Although studied extensively, they have been applied only recently, due to the introduction of efficient tools to analyze and check for satisfiability [21]. In case the data path can be expressed as a Presburger formula, a reachability analysis can be performed on the machine. This approach has be proposed and used in the formal verification [7, 24].

EFSMs can be derived from high-level specifications, e.g. synchronous language Esterel by compilation into circuits, or control-data flow graphs by scheduling and resource binding. It has nice properties as pure finite state machines where certain properties can be decided by reachability analysis, and yet provides a higher abstraction level that incorporates both control and data path. It is becoming a popular model for design, synthesis, verification and testing.

In all applications, smaller EFSMs are desired since they correspond to less complex systems to verify, more compact code to be generated, and easier interpretation of timing specifications. We focus on the minimization of the control logic parts of EFSMs, with the data information used to assist the logic minimization as needed. In this, we use a structural representation with a multi-valued logic network combined with data path constructs, such as predicates, multiplexers and data expressions. The objective of the minimization may be different, depending on the final implementation. We minimize the logic representation in sum-of-products (SOP) and multi-valued decision diagrams (MDD), which are tightly related to the implementation cost whether in hardware or software. Earlier work in this topic can be found in [20]. The authors use symbolic manipulation of the data path to generate don't cares for state and logic minimization of the finite state controller, which assumes a different design model and context. The main contribution of our paper is in computing logic don't cares from the Presburger expressions and transferring don't cares through the data path.

In Section 2, we describe our framework of EFSM minimization and related research. Section 3 presents our method of transferring logic don't cares through the data path. Section 4 discusses the detail of computing logic don't cares from Presburger inequality expressions. We give some results in Section 5 and conclude in Section 6.

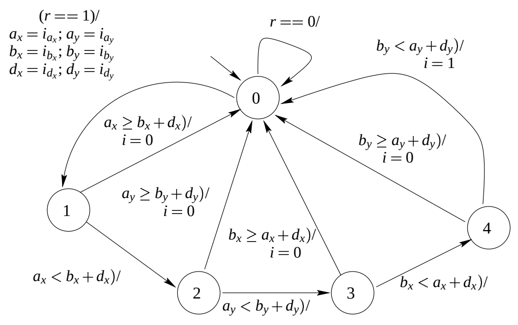
II. Methodology and Related Work

To avoid the state space explosion, we use a structural circuit representation, called control-data networks, for EFSM minimization. It has control nodes and data nodes interconnected with wires. (We assume each node produces a single output.) There is a directed edge from node i to node j, if the function at node j syntactically depends on the output variable at node i. The network has a set of primary inputs and a set of nodes designated as the outputs of the network. The discrete state space of the EFSM is encoded into multi-valued latches; the internal integer variables are represented by data latches.

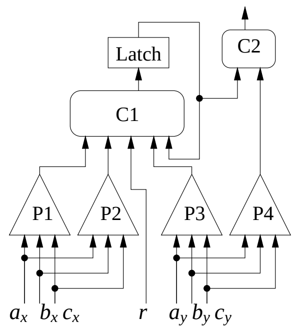
The set of control variables $\mathcal{C}$ have finite ranges and the set of data variables $\mathcal{D}$ have unbounded ranges:

$$\begin{aligned} C &: \forall c_i \in C,\, c_i \in P_i = \{0,1,\ldots,|P_i|-1\},\, |P_i| \in \mathcal{N} \\ D &: \forall d_i \in D,\, |d_i| = \infty \end{aligned}$$

A combinational node n_i in the network belong to the following types: (a) *Control:* $\forall v_i \in input(n_i)$, $v_i \in \mathcal{C}$ and



(a)



(b)

```
.model bound
.inputs ax ay bx by dx dy r
.outputs i
.n ax, ay, bx, by, dx, dy
.mv ns,s 5
.reset s  0
.latch ns s
.table p1 p2 p3 r s -> ns
.default 0
- - - 1 0 1
0 - - - 1 2
- 0 - - 2 3
- - 0 - 3 4
.table s p4 -> i
.default 0
 4 0 1
.data ax bx dx -> p1
"ax >= bx + dx"
.data ay by dy -> p2
"ay >= by + dy"
.data bx ax dx -> p3
"bx >= ax + dx"
.data by ay dy -> p4
"by >= ay + dy"
.end
```

(c)

Fig. 1. EFSM example (a) Explicit state representation; (b) Structural representation; (c) Specification in BLIF-MV.

$output(n_i) \in \mathcal{C}$. It contains a multi-valued logic function. (b) *Expression:* $\forall v_i \in input(n_i)$, $v_i \in \mathcal{D}$ or $v_i \in \mathcal{C}$ and $output(n_i) \in \mathcal{D}$. It contains arithmetic computation on data variables and possibly some control variables. (c) *Predicate:* $\forall v_i \in input(n_i)$, $v_i \in \mathcal{D}$ or $v_i \in \mathcal{C}$ and $output(n_i) \in \mathcal{C}$. It contains predicate logic on data variables and possibly some control variables. (d) *Multiplexer:* $f = f(y_c, y_0, \ldots, y_{n-1})$, where $y_c \in \mathcal{C}$, and $|y_c| = n$, and $y_i \in \mathcal{D}$, $i \in [0, n-1]$. The output f is assigned to y_i if $y_c = i$.

An example EFSM from [24] is shown in Figure 1. It reads the x, y coordinates of two points and a difference vector, and checks whether the two points are closer than the difference. Its structural representation and the corresponding specification in BLIF-MV (extension of [6] with data path) are also shown. The state space is one-hot encoded using a single latch variable s.

A suite of optimization methods for multi-valued logic minimization have been proposed in MVSIS [12]. In this paper, new optimization schemes that incorporate data path information are introduced. We first extend the CODC computation to consider different types of data nodes. This is similar to the black box approach [18], but deals with more cases. We then present methods to compute logic don't cares from Presburger expressions. For the case where a set of predicate nodes fans out to the MV logic, the combination of the predicates that do not occur are used as don't cares to minimize the logic.

To simplify description, in the sequel if not specified otherwise, we use `inputs` to denote primary inputs and the output of latch variables; we use `outputs` to denote primary outputs and the inputs of latch variables.

A. Related Work

The Polis project [2] uses EFSMs as intermediate representation for synthesis and optimization. A high-level design language, like Esterel [5], is interpreted into a circuit EFSM representation, which is subsequently optimized and mapped into hardware and/or software [3], depending on system constraints. Binary Decision Diagrams (BDD) are used to represent and optimize the state transition logic, which tends to blow up on large designs. The data path is stored separately in a look-up table and is not utilized for optimization.

EFSM is also used for high-level synthesis from flow graphs. The authors in [14] proposed an architecture for implementing EFSMs in hardware. The logic minimization is limited in the control domain and no consideration for the data-path.

Presburger arithmetic is adopted in system modeling for its decidability, but the best known procedure for deciding a Presburger expression is triple exponential in the length of the formula [19]. Two approaches exist for manipulating Presburger formulas: automata-based and polyhedra-based [24].

Amon *et al* [1] proposed a method to simplify Presburger expressions for symbolic timing verification. Given a set of quantifier-free Presburger inequalities, the authors use a heuristic to collect predicate combinations that can't occur. They are then presented as don't cares for a logic minimizer [23]. There are two basic limitations: (a) The heuristic examines the Presburger expressions and incrementally selects logic combinations with the number of literals gradually increasing. It is computationally impossible to enumerate all combinations. (b) Each potential combination is individually check by an Presburger tool, Omega [21], for satisfiability, which is computationally expensive.

III. Transferring Don't Cares through the Datapath

Flexibility for logic minimization includes Observability Don't Cares (ODCs), Satisfiability Don't Cares (SDC) and External Don't Cares (XDC). This has been used as a powerful mechanism in minimizing multi-level logic networks [22]. Traditional methods for compatible ODC (CODC) computation have been generalized for multi-valued logic networks [15]. Here we further generalize it to incorporate data path information.

For a general control data network, the ODC set is defined

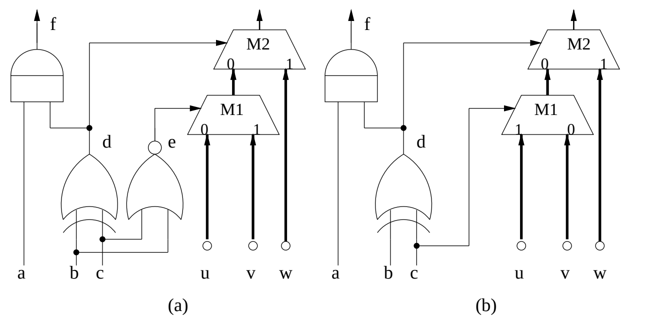


Fig. 2. Multiplexer Example: (a) before simplification, (b) after simplification

in the domain of all multi-valued variables in $\mathcal{C}$, i.e.

$$P = |v_0| \times |v_1| \times \ldots \times |v_{|C|-1}|, \forall v_i \in C$$

The ODC set for an edge E_{ij} in a network, is the set of minterms in $\mathcal{P}$, that can determine all output values without E_{ij}. The ODC set for a node n_i is the intersection of the ODCs for all its fanout edges.

The CODC computation consists of traversing the network from outputs to inputs, and for each node, collecting the CODC set computed from its fanout edges, and then distributing the CODC set to its fanin edges based on the functionality of the node.

For *control* nodes, where inputs and outputs are all MV variables, the same multi-valued CODC computation [15] applies. For a *multiplexer* $f = f(y_c, y_0, \ldots, y_{n-1})$, let $CODC_f$ be the CODC set computed for the output f. It is straight forward that:

$$\begin{aligned} CODC_{y_i} &= (y_c \neq i) \cup CODC_f \\ CODC_{y_c} &= CODC_f \end{aligned}$$

where a data input is not observable if de-selected by the controlling variable.

For a *predicate* or *data* node $f = f(y_1, \ldots, y_n)$, where all inputs are data variables:

$$CODC_{y_i} = CODC_f \ , i = 1 \ldots n$$

The method above does not look into the computation inside a data expression. This is similar to the "black box" approach [18], except that the use of multiplexers produces additional don't cares.

Control nodes are simplified using logic minimization packages like ESPRESSO-MV. Expression nodes and predicate nodes cannot benefit from the don't cares in the control domain $\mathcal{P}$ in our current model.

A multiplexer can be simplified using its CODC set. Let $N_c(X) : C \to P_c$ be the functional mapping from $\mathcal{C}$ to the domain of the controlling variable $y_c \in P_c$. Given its CODC set $CODC_f$, let

$$S = N_c(\overline{CODC_f})$$

where S is the set of "care" values for y_c through the image computation. The set of data inputs $\{y_i | i \notin S\}$ can be removed from the multiplexer f, because they can never be selected and hence are not observable at the outputs.

Figure 2 shows an example of the minimization with multiplexers (where bold wires indicate data variables). The CODC set computed for node `M1` includes the CODC set passed from node `M2`, plus the set of minterms that make node `d` select the value from input `w`. This CODC set is passed to node `e`, and minimizes `e` into an inverter gate. This cannot be achieved through conventional logic simplification.

IV. Don't Care Generation From Presburger Arithmetics

We use Presburger arithmetic to specify the computation of data variables. Here we consider only the subset of Presburger without quantifications. Suppose we have a set of Presburger predicates $\{p_1, \ldots, p_n\}$, which are driven, through some data computation, by a set of natural numbers $\{u_1, \ldots, u_m\}$. We can define Presburger don't cares in the domain of $\{p_1, \ldots, p_n\}$ as the set of combinations that do not occur. This can be computed by unifying $\{p_1, \ldots, p_n\}$ into inequality expressions and solving a linear algebraic equation (Section 4.1). This is sent to the fanouts of $\{p_1, \ldots, p_n\}$ as external don't cares, as illustrated in Figure 3.

Example 1 *Let predicates* $\{p_1, p_2, p_3\}$ *be:*

$$x < 2, \quad 2x + y > 9, \quad y > 5$$

Normalizing these into greater-than comparisons results:

$$-x > -2, \quad 2x + y > 9, \quad y > 5$$

Multiply the three inequalities with vector $\{2, 1, -1\}$. Multiplying with a negative constant is defined here as complementation of the inequality, i.e. changing $>$ to $\leq$. This results:

$$-2x > -4, \quad 2x + y > 9, \quad -y \geq -5$$

The sum of these inequalities becomes $0 > 0$, which is impossible. Since we treated -1 as complementation, the conclusion is that logic combination $p_1 p_2 \overline{p_3}$ can never occur, hence is a don't care for logic minimization. The goal of this computation is therefore generating all possible such vectors that result in an impossible inequality. This can be achieved by making the left hand side zero, which means solving a set of linear algebraic equations. The other don't care for this example is $\overline{p_1}\,\overline{p_2} p_3$.

A. Problem Formulation

Since equality formulas can be converted into inequalities, we only consider inequalities here. Given a set of predicate nodes, $\{p_1, p_2, \ldots, p_n\}$, expressed as inequalities of unbounded natural numbers, we normalize them into the following form:

$$Ax \supset C$$

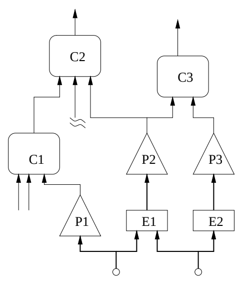


Fig. 3. Predicate Example

where A is the matrix of coefficients with n rows, x is the vector of input integer variables, C is the vector of constants to be compared against, and $\supset$ represents the vector of comparators consisting of only $>$ and $\geq$. Each row of A represent a predicate. We want to find a vector λ such that:

$$\lambda' A x = 0 \tag{1}$$

where λ' is the transpose of vector λ. There are two cubes associated with each λ: $C_p = \hat{p_1}\hat{p_2}\cdots\hat{p_n}$, where

$$\hat{p_i} = \begin{cases} p_i, & \text{if } \lambda_i > 0 \\ \overline{p_i}, & \text{if } \lambda_i < 0 \\ \text{nothing}, & \text{if } \lambda_i = 0 \end{cases}$$

and $C_n = \overline{\hat{p_1}\hat{p_2}}\cdots\overline{\hat{p_n}}$.

Definition 1 *A set of inequalities, $Ax \supset C$, is domain independent, iff there is at least one comparator that does not include equality.*

Theorem 1 *For each λ computed by equation (1), the don't care cube(s) associated with predicate $\{p_1, p_2, \ldots, p_n\}$ is DC_λ:*

$$DC_\lambda = \begin{cases} C_p, & \text{if } \lambda' C > 0 \\ C_n, & \text{if } \lambda' C < 0 \\ C_p, C_n, & \text{if } \lambda' C = 0 \text{ and domain independent} \end{cases} \tag{2}$$

Proof. (Sketch) The first case results in an inequality sum of the form $0 > N$, where N is a positive integer; the second case results in an inequality sum of the form $0 \leq -N$. The last case results in $0 \geq 0$, which means the sub-domain boundaries specified by the set of inequalities intersect at one Euclidean point. However the sub-domains have no intersection because at least one of the sub-domains does not include this point. Therefore there exists no Euclidean point that satisfies all inequalities. □

Theorem 2 *Equation (2) computes the complete don't care set for predicates $\{p_1, p_2, \ldots, p_n\}$.*

The set of λ vectors satisfying equation (1) is the set of solutions to the following:

$$A'\lambda = 0$$

Let the null space of A' have dimension k and basis vectors $B = [b_1, b_2, \ldots, b_k]$, where each b_i is a n dimensional vector. Then λ is a linear combination of vectors in B. Let $\lambda = B \cdot \theta$. Then:

$$\lambda' C = \theta' B' C$$

Therefore the problem becomes, given the null space base vectors B and constant vector C, find all possible distinct sign combinations for the λ's. For each such λ, if $\lambda' C > 0$, C_p is a don't care; if $\lambda' C < 0$, C_n is a don't care; if $\lambda' C = 0$, both C_p and C_n are don't cares if the set of inequalities are domain independent.

Since λ is a function of the θ's, the real problem is to find a proper set of θ's.

B. Branch and Bound

For the possible λ's, we only care to find one for each distinct sign combination. A sign can only be one of three: (-1, 0, 1). In this approach, we create n branching points, one for each λ_i. At each branching point, we branch on the three possible signs; for each branch, we use a linear programming solver to test the existence of θ's that satisfies the constraints; if it succeeds, we continue branching, otherwise backtrack. A don't care is found if we successfully branch to a leaf λ_n and obtain a complete sign pattern.

Given B and C, each λ is a function of θ, i.e. $\lambda_i = F_i(\theta)$. The set of constraints to be satisfied is initialized as $Constr(\theta) = \emptyset$. Figure 4 shows the pseudo code for this procedure.

```
$BnB(i, sign\_pattern)$
    if (i>n) compute_dontcare($sign\_pattern$);

    Constr($\theta$) = Constr($\theta$) $\cup$ $\{F_i(\theta) = 0\}$;
    $sign\_pattern$[i] = 0;
    if (satisfied(Constr($\theta$)) BnB(i+1, sign_pattern);
    else back_track();

    Constr($\theta$) = Constr($\theta$) $\cup$ $\{F_i(\theta) > 0\}$;
    $sign\_pattern$[i] = 1;
    if (satisfied(Constr($\theta$)) BnB(i+1, sign_pattern);
    else back_track();

    Constr($\theta$) = Constr($\theta$) $\cup$ $\{F_i(\theta) < 0\}$;
    $sign\_pattern$[i] = -1;
    if (satisfied(Constr($\theta$)) BnB(i+1, sign_pattern);
    else back_track();
$End$
```

Fig. 4. Branch and Bound pseudo code

Non-orthogonal branching on $\{-1, 0, 1\}$ produces better binding. The result of the branch and bound is a set of don't care cubes. A check is performed to test if the current sign pattern path is subsumed by existing don't care cubes. If it is, the branching is preempted. Branching on $\{-1,1\}$ would produce a set of pure minterms.

In case of $\lambda' C = 0$, we test the domain independence property by checking if there is at least one inequality. We create a flag vector v according the inequality structure: a -1 for $>$ and $<$; a 1 for $\geq$ and $\leq$, as shown below for Example 1.

$$\begin{bmatrix} > \\ > \\ > \end{bmatrix} \Rightarrow v = \begin{bmatrix} -1 \\ -1 \\ -1 \end{bmatrix}$$

This flag vector is array-multiplied by the λ vector. (The result is also a vector, where the i^{th} element is the multiplication of the i^{th} elements of both operand vectors.) If there is at least one -1 in the resulting vector, the inequality set is domain independent. This takes care of complementation for the negative entries in the λ vector.

In Example 1, we have, as input, matrix equation:

$$\begin{bmatrix} -1 & 0 \\ 2 & 1 \\ 0 & 1 \end{bmatrix} x \begin{matrix} > \\ > \\ > \end{matrix} \begin{bmatrix} -2 \\ 9 \\ 5 \end{bmatrix}$$

After computing the null space of A', we obtain the set of λ's as a linear combination of the null vectors B:

$$\lambda = B\theta = \begin{bmatrix} 2 \\ 1 \\ -1 \end{bmatrix} \theta_1$$

For branch and bound, we have only two choices for θ_1: $\theta_1 > 0$ and $\theta_1 < 0$. This results in two sign patterns for λ: $(2,1,-1)$ and $(-2,-1,1)$. For both λ's we have $\lambda C = -4+9-5 = 0$, which means domain independence needs to be checked. We check the array multiplication of v and λ, and apparently the result has at least one -1 in it. Therefore, the don't care cubes $p_1 p_2 \overline{p_3}$ and $\overline{p_1 p_2} p_3$ are obtained as the final result.

The branch and bound method generates the complete set of don't cares, but may require extensive computation on large examples since a linear constraint solver is called at each branching point. As a complementary method, we use Monte Carlo simulation to randomly compute a subset of don't cares for large examples, details in [16].

C. Special Case

In the timing specification applications as presented in [1], it is most likely that the matrix A is unimodular, i.e. the elements of A are from the set $\{-1,0,1\}$. For such cases, we propose an efficient method using multi-valued logic, which effectively avoids solving the linear programming problem and yet preserves the quality of the solution. We present it with an example found in [1].

Example 2 *Let input matrix equation be as follows:*

$$\begin{array}{r} \text{a:} \\ \text{b:} \\ \text{c:} \\ \text{d:} \\ \text{e:} \\ \text{f:} \\ \text{g:} \\ \text{h:} \\ \text{i:} \\ \text{j:} \\ \text{k:} \end{array} \begin{bmatrix} -1 & 0 & 0 & 0 & 1 & 0 \\ -1 & 0 & 0 & 0 & 0 & 0 \\ 1 & -1 & 0 & 0 & 0 & 0 \\ 0 & 0 & -1 & 0 & 0 & 0 \\ 0 & 0 & 1 & -1 & 0 & 0 \\ -1 & 0 & -1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 & 0 & -1 \\ 0 & 0 & 0 & -1 & 0 & 0 \\ -1 & 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 & -1 \\ 0 & 0 & 1 & 0 & 0 & 0 \end{bmatrix} x \begin{matrix} \leq \\ \leq \\ \leq \\ \leq \\ \leq \\ \leq \\ \leq \\ \leq \\ \leq \\ \leq \\ \leq \end{matrix} \begin{bmatrix} 160 \\ -30 \\ 0 \\ 0 \\ 0 \\ 10 \\ -30 \\ -150 \\ 150 \\ -180 \\ 150 \end{bmatrix}$$

where x is the vector of input data variables, and the letters on the left are the names of the predicate inequalities.

Since A is unimodular, the goal is to compute the set of integer λ vectors composed of elements from $\{-1,0,1\}$, which can reduce the matrix to constant zero. Let variables $\{a,\ldots,k\}$ represent the elements in this integer vector, each corresponding to one of the 11 rows. Let literal (a^0, a^1, a^2) represent the element a being $(-1,1,0)$ respectively, as our encoding.

For each column, we create a satisfiability Boolean formulae of variables $\{a,\ldots,k\}$, whose encoding corresponds to the λ vectors that reduce the column to 0. For instance, the second column produces the following equation:

$$\begin{aligned} f_2 \quad = \quad & c^2 g^2 j^2 + c^1 g^1 j^2 + c^0 g^0 j^2 + c^1 g^2 j^1 + \\ & c^0 g^2 j^0 + c^2 g^1 j^0 + c^2 g^0 j^1 \end{aligned}$$

Here, a three-valued logic is used. Cube $c^1 g^1 j^2$ corresponds to a set of minterms: $a^{\{0,1,2\}} b^{\{0,1,2\}} c^{\{1\}} \ldots k^{\{0,1,2\}}$. Each minterm corresponds to a λ vector, one of them being $\{0,0,1,0,0,0,1,0,0,0,0\}$.

If there are more than three non-zero elements in the column, we need to generate $C_2^1 = 2$ cubes for each pair of non-zero elements; $C_4^2 = 6$ cubes for each bipartitioning of 4 non-zero elements; and $C_6^3 = 15$ cubes for each bipartitioning of 6 non-zero elements, and so on. The total number of cubes to be produced in this process is

$$\sum_{k=1}^{n/2} \begin{pmatrix} 2k \\ n \end{pmatrix} \cdot \begin{pmatrix} k \\ 2k \end{pmatrix}$$

where n is the number non-zero elements in the column. For instance, the second column corresponds to equation:

$$\begin{aligned} f_3 \quad = \quad & d^1 e^1 f^2 k^2 + d^0 e^0 f^2 k^2 + d^1 e^2 f^0 k^2 + d^0 e^2 f^1 k^2 + \\ & d^1 e^2 f^2 k^1 + d^0 e^2 f^2 k^0 + d^2 e^1 f^1 k^2 + d^2 e^0 f^0 k^2 + \\ & d^2 e^1 f^2 k^0 + d^2 e^0 f^2 k^1 + d^2 e^2 f^1 k^1 + d^2 e^2 f^0 k^0 + \\ & d^1 e^1 f^1 k^1 + d^0 e^0 f^0 k^0 + d^0 e^1 f^1 k^0 + d^1 e^0 f^0 k^1 + \\ & d^1 k^0 e^1 f^0 + d^0 k^1 e^0 f^1 \end{aligned}$$

Producing the Boolean formulae can take exponential time. As a reasonable estimate, we consider up to 4 non-zero entries in each vector.

We generate the equation for each column; the intersection of these equations gives all the vectors in the null space that we consider: (Due to space limits, they are not listed individually.)

$$f_1 f_2 f_3 f_4 f_5 f_6$$

As a result, we obtain a three-valued function with 106 cubes. At this point, we only need to care about the non-zero elements in the vectors. Therefore, we switch from three-valued logic to binary logic, by removing literals with value 2, which represent the corresponding element not appearing in the vector.

Each cube in the representation corresponds to a unique sign λ sign pattern. Recall that each sign pattern has a corresponding *complement* sign pattern, as discussed for C_p and C_n. The last step is to check these sign patterns, along with their *complements*, on the constant vector C. For example, with dk, $\lambda C = 150$, which is feasible; with $d'k'$, $\lambda C = -150$, which implies $0 \leq -150$ and results in a don't care.

Out of the 106 cubes, along with their complements, 21 cubes pass the test. Making the cubes prime and irredundant, we have the set of don't cares as a final result:

$$af'k' + a'fk + a'i + d'fi' + d'k' + \\ eh'k' + e'hk + ghj' + g'h'j$$

Note that this is the same set of don't cares obtained in [1] by repetitively calling the Presburger tool Omega. Yet our method is simpler and deterministic, and we claim that this is the complete don't care set for this example.

V. Experimental Results

The multi-valued logic optimizations and extended data path don't cares are implemented in MVSIS [12]. The don't care computation from Presburger expressions are prototyped in the Matlab system.

For experiment on Presburger examples, we apply both the branch and bound and the Monte Carlo methods on Example 2. The branch and bound method produces about 600 don't care cubes after around 30 minutes. After logic minimization they are reduced to the same don't care set as presented in Section 4.3. The Monte Carlo method returns a few don't care cubes within a couple of minutes if we use 1000 random vectors in the null space.

In additional experiments, we derive EFSMs from Esterel programs for embedded control applications. They include automobile control, processor and memory control, and the controller for a Lego Mindstorms Acrobot [4]. The Esterel compiler is used to parse the input Esterel program and produce an intermediate data-flow representation called DC. We translate the DC format into our intermediate network representation in BLIF-MV (extension from [6]).

The statistics of these examples are shown in Table I under *original specification*. `PI` (`PO`) means the number of primary inputs (outputs) that are control variables (`c`) and data variables (`d`). Similarly the number of latches in control and data. `Pred` and `Expr` are the number of predicates and arithmetic expressions in the representation. These are not simplified in our current model and hence remain unchanged.

The results of our synthesis scheme are reported in the number of multiplexers (`Mux`), multi-valued cubes (`Cubes`) and multi-valued literals in the factored forms (`Lits`). Column `simplify` reports the result by just applying conventional logic minimization and ignoring the data path elements. Column `simplify-d` reports the results with simplification using don't cares proposed in this paper.

As shown, the logic representation is dramatically simplified as compared against the original specification. With additional don't cares, the number of multiplexers are also reduced, and in some cases the logic is simplified further.

Optimization results of the proposed techniques depends on the target application. In particular, the black-box don't care techniques can be benefited only when control and data portions are tightly interacted, sharing large amount of computation in the transitive fanin cones. The Presburger-base approach requires the predicate nodes share common support data variables, and therefore increasing the chances of arithmetic redundancies. These features are not frequently observed in the Esterel control applications that we have experimented with.

VI. Conclusion

A new approach of minimizing EFSMs using multi-valued logic and Presburger expressions is presented. We proposed methods to evaluate and utilize multi-valued don't cares in a general control data network environment; we proposed methods to compute logic don't cares from Presburger expressions, which do not invoke any computationally expensive arithmetic satisfiability checking. Preliminary results are encouraging in applications of Presburger expression simplification and EFSM minimization. We believe the overall approach is applicable to problems in synthesis and formal verification of embedded systems.

In our experiments with EFSM examples generated from Esterel programs, Presburger predicates do not appear in large amounts and with overlapping variable support sets. We need to study more EFSM applications (for example in high-level synthesis from flow graphs) where this paradigm do appear and the techniques described can bring significant benefits.

Future research includes devising heuristics to explore the solution space of potential don't cares for large Presburger examples. So far the don't care is limited in the control domain in $\mathcal{C}$. It is possible to extend this definition to the data variable domain as intervals of integer values, which can be used to simplify data expressions and predicates. This would require a package like *omega* or *Shasta* to do reasoning in an infinite space.

Acknowledgement

The authors would like to acknowledge Max Chiodo from Cadence Berkeley Labs, for providing some embedded control examples in an intermediate format; we thank Hongjing Zou for prototyping the don't care generation method for Presburger arithmetic. We are grateful for the support of the SRC under contract 683.004 and the California Micro program and industrial sponsors, Fujitsu, Cadence, Motorola and Synopsys.

TABLE I
SYNTHESIS RESULTS FOR EFSM EXAMPLES DERIVED FROM ESTEREL PROGRAMS

	original specification								simplify			simplify-d		
example	PI(c/d)	PO(c/d)	Latch(c/d)	Pred	Expr	Mux	Cubes	Lits	Mux	Cubes	Lits	Mux	Cubes	Lits
driver	43/42	27/22	2/69	41	58	118	251	422	118	84	185	72	103	170
engine_speed	4/2	4/0	7/14	15	24	35	290	373	35	70	110	17	70	134
fuel_ctr	7/2	2/1	10/4	7	4	14	188	263	14	68	99	8	62	103
fuel_pulse	10/8	4/2	10/8	4	10	14	184	242	14	41	59	10	38	63
instr	3/3	10/3	23/6	6	6	9	247	358	9	116	171	9	116	171
instr_cyc	3/3	10/3	24/6	6	6	9	285	414	9	125	202	9	127	204
legodance	3/0	7/7	11/27	13	42	66	313	407	66	102	128	38	121	200
mem_ctr	8/6	6/4	21/14	6	18	36	387	529	36	126	194	20	136	226
msd	9/7	9/8	13/8	10	11	19	310	422	19	107	143	8	88	146
supervisor	12/9	11/9	3/9	4	9	11	121	144	11	25	35	9	26	35
update_pulse	6/5	5/4	16/10	4	14	24	260	342	24	88	114	12	76	130
wheel_speed	4/3	4/0	4/14	13	23	33	209	252	33	21	42	18	27	49
average						1	1	1	1	0.31	0.34	0.65	0.31	0.37

REFERENCES

[1] T. Amon, G. Borriello, and J. Liu. Making complex timing relationships readable: Presburger formula simplification using don't cares. In *Proc. of the Design Automation Conf.*, June 1998.

[2] F. Balarin, M. Chiodo, P. Giusto, H. Hsieh, A. Jurecska, L. Lavagno, C. Passerone, A. Sangiovanni-Vincentelli, E. Sentovich, K. Suzuki, and B.Tabbara. *Hardware-Software Co-Design of Embedded Systems: The Polis Approach*. Kluwer Academic Press, 1997.

[3] F. Balarin, M. Chiodo, P. Giusto, H. Hsieh, A. Jurecska, L. Lavagno, A. L. Sangiovanni-Vincentelli, E. M. Sentovich, and K. Suzuki. Synthesis of software programs for embedded control applications. *IEEE Trans. Comput.-Aided Design Integrated Circuits*, 18(6):834–49, June 1999.

[4] G. Berry. A dancing lego mindstorms acrobot programmed in esterel. *Technical Report*, 2000.

[5] G. Berry and G. Gonthier. The Esterel synchronous programming language: Design, semantics, implementation. *Science of Computer Programming*, 1992.

[6] R. K. Brayton, M. Chiodo, R. Hojati, T. Kam, K. Kodandapani, R. P. Kurshan, S. Malik, A. L. Sangiovanni-Vincentelli, E. M. Sentovich, T. Shiple, K. J. Singh, and H.-Y. Wang. BLIF-MV: An Interchange Format for Design Verification and Synthesis. Technical Report UCB/ERL M91/97, Electronics Research Lab, Univ. of California, Berkeley, CA 94720, Nov. 1991.

[7] T. Bultan, R. Gerber, and W. Pugh. Symbolic model checking of infinite state programs using Presburger arithmetic. In *Proc. of the Computer-Aided Verification Conf.*, 1997.

[8] J. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang. Symbolic model checking: 10^{20} states and beyond. In *IEEE Symposium on Logic in Computer Science*, June 1990.

[9] K. T. Cheng and A. Krishnakumar. Automatic functional test generation using the extended finite state machine model. In *Proc. of the Design Automation Conf.*, June 1993.

[10] S. Devadas, K. Keutzer, and A. Krishnakumar. Design verification and reachability analysis using algebraic manipulation. In *Proc. of the Intl. Conf. on Computer Design*, Oct. 1991.

[11] H. B. Enderton. *A Mathematical Introduction to Logic*. Academic Press, 1972.

[12] M. Gao, J. Jiang, Y. Jiang, Y. Li, S. Singha, and R. K. Brayton. MVSIS. In *Proc. of the Intl. Workshop on Logic Synthesis*, May. 2001.

[13] V. Glushkov. Automata theory and structural design problems of digital machines. In *Cybernetics 1*, 1965.

[14] S.-Y. Huang. On speeding up extended finite state machines using catalyst circuitry. In *Proc. of the Asia and South Pacific Design Automation Conf.*, Feb. 2001.

[15] Y. Jiang and R. K. Brayton. Don't cares and multi-valued logic network minimization. In *Proc. of the Intl. Conf. on Computer-Aided Design*, Nov. 2000.

[16] Y. Jiang and R. K. Brayton. Don't care computation in minimizing extended finite state machines with presburger arithmetic. Technical Report UCB/ERL M01/35, Electronics Research Laboratory, University of California, Berkeley, Dec. 2001.

[17] Y. Jiang and R. K. Brayton. Software synthesis from synchronous specifications using logic simulation techniques. In *Proc. of the Design Automation Conf.*, June 2002.

[18] T. H. Liu, K. Sajid, A. Aziz, and V. Singhal. Optimizing designs containing black boxes. In *Proc. of the Design Automation Conf.*, June 1997.

[19] D. Oppen. A $2^{2^{2^{pn}}}$ upper bound on the complexity of Presburger arithmetic. *the Journal of Computer and System Sciences*, 1978.

[20] M. A. Perkowski and J. E. Brown. Automatic generation of don't cares for the controlling finite state machine from the corresponding behavioral description. In *Proc. of the Intl. Symposium on Circuits and Systems*, May 1990.

[21] W. Pugh and *et al*. The Omega project. `http://www.cs.umd.edu/projects/omega`.

[22] H. Savoj and R. K. Brayton. The Use of Observability and External Don't Cares for the Simplification of Multi-Level Networks. In *Proc. of the Design Automation Conf.*, pages 297–301, June 1990.

[23] E. M. Sentovich, K. J. Singh, L. Lavagno, C. Moon, R. Murgai, A. Saldanha, H. Savoj, P. R. Stephan, R. K. Brayton, and A. L. Sangiovanni-Vincentelli. SIS: A System for Sequential Circuit Synthesis. Technical Report UCB/ERL M92/41, Electronics Research Laboratory, Univ. of California, Berkeley, CA 94720, May 1992.

[24] T. R. Shiple, J. H. Kukula, and R. K. Ranjan. A comparison of presburger engines for EFSM reachability. In *Proc. of the Computer-Aided Verification Conf.*, 1998.