DON'T CARE COMPUTATION IN MINIMIZING EXTENDED FINITE STATE MACHINES WITH PRESBURGER ARITHMETIC

by

Yunjian Jiang and Robert Brayton

Memorandum No. UCB/ERL M01/35

7 December 2001

DON'T CARE COMPUTATION IN MINIMIZING EXTENDED FINITE STATE MACHINES WITH PRESBURGER ARITHMETIC

by

Yunjian Jiang and Robert Brayton

Memorandum No. UCB/ERL M01/35

7 December 2001

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

Don't Care Computation in Minimizing Extended Finite State Machines with Presburger Arithmetic

Yunjian Jiang Robert Brayton
Department of Electrical Engineering and Computer Sciences
University of California, Berkeley
{wjiang,brayton}@eecs.berkeley.edu

Abstract

This paper addresses the problem of generating logic don't cares in minimizing an Extended Finite State Machines (EFSMs). 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. A structural representation of such EFSMs and an optimization scheme using multi-valued logic is proposed in this paper. It consists of new methods to transfer don't cares through the datapath and to generate logic don't cares from the data path. Potential applications are discussed and preliminary results validate the scheme on some reasonable examples.

1 Introduction

Extended Finite State Machines (EFSMs) have been studied for system level design modeling and synthesis [3, 4], as a way to raise abstraction levels and capture both hardware and software issues. An EFSM is a system with a finite state controller interacting with an unbounded integer datapath [8]. Each transition of the controller is guarded by a predicate over the integer variables, and associated with an action function which updates the new values of the integer variables. In all applications, smaller EFSMs are beneficial since they correspond to less complex systems to verify, more compact code to be generated, and easier interpretation of timing specifications. However, minimizing an EFSM has been little studied.

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 [9], 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 [16]. In case the datapath 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, 20].

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 datapath constructs, such as predicates, multiplexers and data expressions. The main contribution of this paper is in computing logic don't cares from the Presburger expressions and transferring don't cares through the datapath.

Other than the applications of EFSMs in formal verification, our approach can be used in symbolic verification of timing diagrams as discussed by Amon *et al* [1], and for Esterel compilation. In [13], we discussed using multi-valued logic combined with datapath constructs as an intermediate representation for optimization and code generation from Esterel. If the data expressions used in Esterel are limited to Presburger arithmetic, the approach presented here can be used to generate more efficient implementation code.

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 datapath. 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.

2 Methodology and Related Work

To avoid the state space explosion, we use a structural circuit representation, called control-data networks, for EFSM minimization. A control-data network has control nodes and data nodes interconnected with wires, or variables. There are two types of variables: multi-valued variables with finite ranges and data variables with unbounded ranges. There are four types of nodes: control, multiplexer, data and predicate nodes. These are categorized according to their input and output variables types, as in Table 1. 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 net-

work. There are also latches for both control and data variables to model sequential behaviors.

node types	*operation*	*input*	*output*
control	logical	MV	MV
expression	arithmetic	data	data
multiplexer	assignment	MV/data	data
predicate	predicate	data	MV

Table 1: Node types in a control data network

Having explicit multiplexers enables further logic minimization and datapath simplification. We define a multiplexer as $f = f(y_c, y_0, \cdots, y_{n-1})$, where y_c is a MV-variable with n values, $y_i, i \in [0, n-1]$ are data inputs, and the output f is assigned to y_i if $y_c = i$. Logic optimization heuristics like node collapsing and elimination are generalized for multiplexers.

2.1 Multi-Valued Logic Networks

The part of the network that consists of control nodes are represented as a multi-valued logic network. Each control node in the network is a multi-valued function. In general, a variable x_i is multi-valued and takes on values from the set $P_i = \{0, 1, ..., |P_i| - 1\}$. A **literal** of a MV-variable x is associated with a subset of values for that variable. A **product term** or **cube** is a conjunction of literals and evaluates to 1 if each of the literals evaluates to 1. A **sum-of-products** (SOP) is the disjunction of a set of product terms and it evaluates to 1 if any of the products evaluates to 1.

A set of optimization methods for such multi-valued logic networks are implemented in MVSIS [11]. *Algebraic methods* [10] include methods for finding common sub-expressions, semi-algebraic division, decomposing an MV-network, factoring an expression, and algebraic resubstitution. *Node simplification* [12] uses a generalization of compatible observability don't cares (CODC) to minimize the logic of a node in the network. This performs Boolean resubstitution as well. *Elimination* merges a node into its fanouts. *Resubstitution* tries to substitute existing nodes into larger nodes in order to save cubes or literals. Also methods specifically tuned for multi-valued logic, like pair decode and encoding [11], are used.

In this paper, new optimization schemes that incorporate datapath information are introduced. We first extend the CODC computation to consider different types of data nodes. This is similar to the black box approach [14], 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 can't occur are used as don't cares to minimize the logic.

2.2 Related Work

The Polis project [3] uses EFSMs for intermediate representation for synthesis and optimization. A high-level design language, like Esterel [6], is interpreted into a circuit EFSM representation, which is subsequently optimized and mapped into hardware and/or software [4], depending on system constraints. Binary Decision Diagrams (BDD) are used to represent and optimize the control logic, while the datapath is stored separately in a look-up table. The BDD optimizations are tuned for low level hardware implementation, and the datapath information can not be utilized for optimization. Subsequent research [2] included data expressions, but only as black boxes.

Presburger arithmetic is adopted for its decidability, but the best known procedure for deciding a Presburger expression is triple exponential in the length of the formula [15]. There two basic approaches for manipulating and checking the satisfiability of Presburger formulas: automata-based and polyhedra-based. A good comparison of these is presented in [20].

Amon *et al* [1] proposed a method to simplify a Presburger expression in an application for symbolic timing verification. This is the work most related to this paper. Given a set of quantifier-free Presburger inequalities, the approach uses a heuristic to collect predicate combinations that can't occur. They are then presented as don't cares for a logic minimizer [17]. 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 [16] for satisfiability. It is computationally expensive to invoke such tools for each candidate.

3 Transferring Don't Cares through the Datapath

Compatible Observability Don't Cares (CODCs) are defined as the set of minterms, for an intermediate node in a logic network, that make the logic value of this node non-observable at the primary outputs. This has been used as a powerful mechanism in minimizing a multi-level logic network, as implemented in SIS [17].

Traditional methods for CODC computation have been generalized for multi-valued logic networks [12]. Here we further generalize it to incorporate datapath information. In the discussion that follows, the CODC set is defined in the logic domain composed of all intermediate MV-variables in the network.

For control nodes, where inputs and outputs are all MV variables, the same multi-valued CODC computation [12] applies. For a multiplexer $f = f(y_c, y_0, \cdots, y_{n-1})$, let $CODC_f$ be

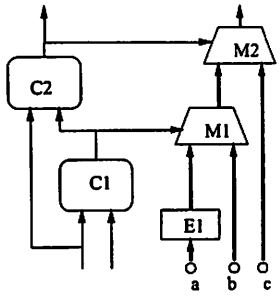

Figure 1: Multiplexer Example

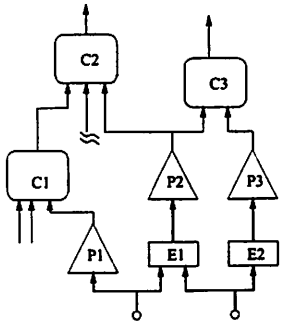

Figure 2: Predicate Example

the CODC set computed for the output f. It is straight forward that:

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

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

$$CODC_{y_i} = CODC_f.$$

The method above does not look into the computation inside a data expression. This is very similar to the "black box" approach [14], except that the use of multiplexers produces additional don't cares.

A multiplexer can also be simplified using its CODC set. Let X be the Boolean space of all intermediate MV-variables, and $N_c(X)$ be the functional mapping from X to the controlling variable y_c.

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

S is the set of "care" values for y_c. Let M be the set of all values that y_c can take. Then the data inputs associated with values in $\{M \backslash S\}$ are not observable at primary outputs and hence can be removed.

Figure 1 shows an example of the minimization with multiplexers. Bold wires indicate data variables. Multiplexers `M1` and `M2` are controlled by MV-nodes `C1` and `C2` respectively. The CODC set computed for node `M1` includes the CODC set passed from node `M2`, plus the set of minterms that make MV-node `C2` select the value from input `c`. This CODC set is passed to MV-node `C1` and used for minimizing `C1`.

4 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 can't occur. This can be computed by unifying $\{p_1, \ldots, p_n\}$ into inequality expressions and solving a linear algebraic equation. This is sent to the fanouts of $\{p_1, \ldots, p_n\}$ as external don't cares, as illustrated in Figure 2.

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 $\overline{p_1} \overline{p_2} p_3$.

4.1 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:

$$Ax \supset C$$

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:*

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

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. □

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.

4.2 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 3 shows the pseudo code for this procedure.

```
BnB(i, sign_pattern)
    if (i>n) compute_dontcare(sign_pattern);

    Constr(θ) = Constr(θ) ∪ {F_i(θ) = 0};
    sign_pattern[i] = 0;
    if (satisfied(Constr(θ)) BnB(i+1, sign_pattern);
    else back_track();

    Constr(θ) = Constr(θ) ∪ {F_i(θ) > 0};
    sign_pattern[i] = 1;
    if (satisfied(Constr(θ)) BnB(i+1, sign_pattern);
    else back_track();

    Constr(θ) = Constr(θ) ∪ {F_i(θ) < 0};
    sign_pattern[i] = -1;
    if (satisfied(Constr(θ)) BnB(i+1, sign_pattern);
    else back_track();
End
```

Figure 3: 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. 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 $\lambda C = 0$, 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 $\overline{p_1} \overline{p_2} p_3$ are obtained as the final result.

4.3 Monte Carlo

Using random simulation, we generate a large number of normalized vectors in the null space $null(A')$. These are used to compute the vector θ, which are then tested on the constraint $\lambda' C > 0$. We record the set of distinct sign patterns without invoking any linear programming computation.

For each resulting minterm, we expand its logic space by removing a subset of literals; a larger don't care cube is obtained if it still satisfies the constraints.

The branch and bound method generates the complete set of don't cares, but may require extensive computation on large examples. The Monte Carlo method randomly computes a subset of don't cares, but can be extremely fast on large examples.

4.4 Special Case

For the case where A is unimodular, we propose an efficient method using multi-valued logic. We present it with an example found in [1].

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

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

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:

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

Here, a three-valued logic is used. Cube $c^1 g^1 j^2$ corresponds to a set of minterms: $a\{0,1,2\} b\{0,1,2\} c\{1\} \cdots k\{0,1,2\}$. 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 subset of 4 non-zero elements; and $C_6^3 = 15$ cubes for each subset of 6 non-zero elements, and so on. The total number of cubes to be produced in this process is

$$\sum_{k=1}^{n/2} \binom{2k}{n} \cdot \binom{k}{2k}$$

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

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

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 in 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.

5 Experimental Results

The multi-valued logic optimizations and extended datapath don't cares are implemented in MVSIS [11]. The don't care computation from Presburger expressions are prototyped in the Matlab system. We report our results first in the application of minimizing Presburger expressions, and then in EFSM minimization.

5.1 Presburger Example

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 4.4. 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.

5.2 EFSM Example

In this experiment, we use Esterel as a high-level specification language to obtain our EFSMs. The Esterel compiler is used to parse the input Esterel program and produce an intermediate circuit representation called DC. We translate the DC format into our intermediate control-data network representation in BLIF-MV. As a back-end experiment, we also generate implementation C code after the optimization, which is described in [13].

We use an Esterel example that has reasonable size and a decent amount of interaction between control and data, which is the emphasis of our techniques. This is an Esterel program that drives a Lego Mindstorms Acrobot [5], which has a front bumper and a rear wheel. The Acrobot performs a dancing pattern, holding the dance for a while and backing up when a shock occurs on the bumper.

Table 2: Lego Mindstorms Acrobot Example

	origin	MVSIS	MVSIS-D
nodes	307	180	180
MUXs	73	39	39
PREDs	13	13	13
EXPRs	41	41	41
LATCHes	38	38	38
cubes	261	139	130
lits	529	320	303
code-size	-	6695	6359

Table 3: Other examples

	MVSIS					
Examples	lits	size	E-auto	E-sort	E-opt	comb
eng-ctr	103	2393	2081	2809	2145	1895
instr-ctr	399	6695	60019	10057	2965	3259
mem-ctr	621	15380	51443	23256	5615	4924

In Table 2, column `origin` shows the statistics translated from the DC format; column `MVSIS` shows the results after normal logic optimizations scripts from [11]; column `MVSIS-D` shows the results that combine don't cares computed from the datapath. `nodes` shows the total number of control and data nodes of all types. `code-size` is the sizes of the binary objects compiled with gcc -O3, from the C code generated from the network.

As shown, the pure datapath (predicates and expression) remains the same, but the number multiplexers are cut in half. With don't cares from the datapath, the logic representation (cubes and literals) are reduced further, which is also shown in the final compiled code size.

5.3 More Examples

We have experimented with other Esterel examples, such as controllers in microprocessor designs and automobiles designs, etc. The minimization results are encouraging as compared with the Esterel compiler. Some of the examples are shown in Table 3. Due to lack of interaction between data and control, the data-path don't cares did not contribute significantly in these minimizations.

The three examples consists of an electronic engine fuel controller, an instruction decoder and a direct memory access controller. The size of the examples ranges from 100 to 500 lines of Esterel source code. The two `MVSIS` columns show the number of MV literals and the size of compiled branching program generated from MDD representations. The three columns in the middle show the binary size of the code generated by the Esterel compiler.

`E-auto` shows the code generated from an automata repre-

sentation, which for large examples tend to blow up; `E-sort` is the code from a binary circuit representation; `E-opt` is also circuit code but optimized by an extension of SIS called `Basicopt`, which consists of binary combinational area optimizations, state encoding and latch removal. As shown, the optimized circuit code is much smaller in size, and smaller than the code generated by MVSIS. This is because sequential redundancy is introduced by the Esterel DC compiler, which can be minimized away by powerful latch removal algorithms [18, 19]. This cannot be achieved by MVSIS for its primitive sequential optimization capabilities. If we treat the optimized circuits after `Basicopt` as input, optimize further in MVSIS and generate code, the results are shown in column `comb`. In general, this produces smaller code, due to the benefits of multi-valued variables and the MDD-based branching program.

We did not have time to compare execution speed. With similar code size, MDD-based branching code executes faster on average cases than circuit equation-based code generated by Esterel, because not every line of code is executed.

6 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 where this paradigm do appear and the techniques described in this paper can bring significant benefits.

Future research includes devising heuristics to explore the solution space of potential don't cares for large Presburger examples, and applying this to code generation in MVSIS. Sequential minimizations like latch removal are very important for circuits generated from high-level languages, which we should incorporate as well. Also we would like to incorporate formal verification algorithms to validate the minimization results.

Acknowledgement

The authors would like to acknowledge Max Chiodo from Cadence Berkeley Labs, for providing an intermediate format and its parser from Esterel DC; also Hongjing Zou for prototyping the don't care generation method from Presburger. We would like to thank Ellen Sentovich and Michael Kishinevsky for providing the Esterel examples. 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.

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 and M. Chiodo. Software synthesis for complex reactive embedded systems. In *Proc. of the Intl. Conf. on Computer Design*, Oct. 1999.

[3] 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.

[4] 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.

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

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

[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] 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.

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

[10] M. Gao and R. K. Brayton. Semi-algebraic methods for multi-valued logic. In *Proc. of the Intl. Workshop on Logic Synthesis*, May. 2000.

[11] 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.

[12] 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.

[13] Y. Jiang and R. K. Brayton. Logic optimization and code generation for embedded control applications. In *Proc. of the Intl. Symposium on Hardware/Software Co-Design*, Apr. 2001.

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

[15] D. Oppen. A $2^{2^{2^{pn}}}$ upper bound on the complexity of Presburger arithmetic. *the Journal of Computer and System Sciences*, 16(3):323–32, July 1978.

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

[17] 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.

[18] E. M. Sentovich, H. Toma, and G. Berry. Latch optimization in circuits generated from high-level descriptions. In *Proc. of the Intl. Conf. on Computer-Aided Design*, pages 428–35, Nov. 1996.

[19] E. M. Sentovich, H. Toma, and G. Berry. Efficient latch optimization using exclusive sets. In *Proc. of the Design Automation Conf.*, pages 8–11, June 1997.

[20] 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.