Software Synthesis from Synchronous Specifications Using Logic Simulation Techniques

Yunjian Jiang
wjiang@eecs.berkeley.edu

Robert K. Brayton
brayton@eecs.berkeley.edu

Department of Electrical Engineering and Computer Science
University of California, Berkeley CA 94720

ABSTRACT

This paper addresses the problem of automatic generation of implementation software from high-level functional specifications in the context of embedded system on chip designs. Software design complexity for embedded systems has increased so much that a high-level functional programming paradigm need to be adopted for formal verifiability, maintainability and short time-to-market. We propose a framework for efficiently generating implementation software from a synchronous state machine specification for embedded control systems. The framework is generic enough to allow hardware/software partition for a given architecture platform. It is demonstrated that the logic optimization and simulation techniques can be combined to produce fast execution code for such embedded systems. Specifically, we propose a framework for software synthesis from multi-valued logic, including fast evaluation of logic functions, and scheduling techniques for node execution. Experiments are performed to show the initial results of our algorithms in this framework.

1. INTRODUCTION

We adopt a platform-based design methodology, with two major features: (a) using standardized or application specific processing platforms to execute intelligent software, and (b) raising the abstraction level of software designs with implementation code automatically generated from functional models.

By using standardized processors and moving intellectual properties into software design, the methodology reduces hardware manufacturing cost, increases flexibility of the design, hence extends product life cycle and shortens time-to-market. However, software design in such systems becomes extremely complex. High-level functional programming paradigms tackle this problem by abstracting away implementation details with code generated automatically from functional computation models.

We focus on control intensive embedded systems, for instance, automobile engine, airplane, and network protocol controls. The computation model used is a network of extended finite state machines (EFSMs). An EFSM is a system with a finite state controller interacting with an unbounded integer data-path [9]. Each transition of the controller is guarded by a predicate over the integer variables, and is associated with an action function which updates the values of the integer variables. This model has been studied for system design and verification. High-level synchronous languages like Esterel [7] can be used to program in such models.

A static synchronous model like EFSMs cannot model or verify dynamic aspects of the embedded software, like dynamic task and memory allocation, which may be important in some applications. We argue that in safety critical control applications, like avionics and automobile electronics, such synchronous models have to be used for formal verifiability and executable specifications. There are research efforts to combine the synchronous model with dynamic aspects to provide powerful system level modeling, which is not the focus of this paper.

Our design flow starts from high-level synchronous specification, textual or graphical, like Esterel, Argos, Lustre, or Matlab StateFlow. EFSMs are derived from these and optimized using multi-valued (MV) logic synthesis techniques. The internal representation, a network of functional nodes, supports abstract data variables (which may be implemented by integer or floating point data path) and black-box function calls. The computation involved with the abstract data variables are not interpreted by the tools, but the information flow is utilized for synthesis in the logic domain.

In the mapping phase, the functional model can be mapped to different architecture platforms. For a configurable system-on-chip platform, each node can be estimated for performance and implemented either in hardware or software, using techniques described in [4]. For pure software implementation, this paper presents a framework for generating efficient code using logic simulation techniques. This includes fast logic function evaluation and scheduling for node execution.

In the remainder of this paper, Section 2 reviews related work in this area; Section 3 describes the computation model we use for optimization and synthesis; Section 4 details our techniques on code generation and a scheduling scheme; Section 5 shows experimental results followed by conclusions.

2. RELATED WORK

The Polis project [2] uses EFSMs for intermediate representation and synthesis. A high-level design language, like Esterel [7], is compiled into a set of EFSMs, which are subsequently optimized and mapped into hardware and/or software [3], depending on system constraints. The communication model among these EFSMs is a Globally Asynchronous Locally Synchronous (GALS) mechanism. Each EFSM is represented by a single state transition table for the control path and a lookup table for the data-path. Binary Decision Diagrams (BDDs) are used to optimize the state transitions and synthesize the code. This representation cannot scale to very

DAC 2002, June 10-14, 2002, New Orleans, Louisiana,USA.

Table 1: Node types in a control data network

node types	operation	input	output	example
control	logical	MV	MV	$a\{0\}b\{1,2\}+a\{1\}$
data	arithmetic	data	data	x + y
multiplexer	assignment	MV/data	data	$c\{0\}$x$+c\{1\}$y
predicate	predicate	data	MV	x > y

large designs. In addition, data-path information is not utilized for optimization.

The Esterel compiler from the Esterel team [11] has two ways of generating code from different representations of a FSM. The automata method [6] enumerates all possible states and generates the action code for each state. This can produce very fast code, but for large examples, the code size blows up with the state space. The circuit method [5] combines a multi-level Boolean circuit with a data-path table, and generates code based on the logic equations. The code size is scalable, but with compromises in execution speed. The communication model between processes is synchronous composition through a set of completely unordered synchronous events.

Edwards proposed compiling Esterel directly into sequential C code [10], by static scheduling and context switching. This approach applies only to statically schedulable applications. Other Esterel compilation techniques that have been proposed include modular compilation [14], and reaction acceleration [24]. By compiling Esterel directly into C code, these approaches do not easily support partitioning between hardware and software.

The area of code generation for control applications overlaps largely with discrete function simulation. Most closely related publications are in cycle simulation using decision diagrams [19, 1]. The scheduling techniques to be presented here also have some resemblance with event-driven and back-tracing simulation [23, 18]. However, these simulation techniques are tuned for implementation in high-performance computer servers, rather than cost sensitive and resource constrained embedded systems.

The literature is also rich in software synthesis for other application domains and computation models, for instance, code generation from synchronous data flow, process networks, and Petri Nets [22]. These are not studied or compared in our paper.

3. COMPUTATION MODEL

We use a network of EFSMs as the computation model. The techniques presented are for synthesis and code generation for a single EFSM, but they are easily extensible to interacting EFSMs with synchronous or asynchronous communication.

We adopt a functional representation that uses multi-valued variables and Boolean algebra. We believe this applies naturally to embedded software synthesis with advantages over binary logic.

- It explores bit level parallelism and thus saves execution instructions for updating variables and RAM usages.
- A larger optimization space is explored for finding an optimal implementation without being limited to a particular binary encoding.

We use MVSIS [13, 8] for optimization and synthesis. For EFSM minimization, it operates on a control-data network representation. These have control nodes and data nodes interconnected with two types of variables: multi-valued variables with finite ranges and abstract data variables with unbounded ranges. There are four types of nodes: *control*, *multiplexer*, *data* and *predicate*. These are classified according to their input and output variables types, as shown in Table 1. In the examples in the table, a, b, c are multi-valued control variables, and x, y are data variables.

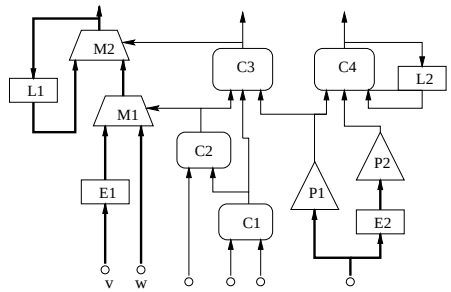

Figure 1: Control-data network

A multiplexer is defined as $f = f(y_c, y_0, \ldots, y_{n-1})$, where y_c is a MV-variable with n values, y_i, ($i \in [0, n-1]$) are data inputs. The output f is assigned to y_i if $y_c = i$. The data computation contained in predicate nodes and expression nodes are currently modeled as uninterpreted strings. The expressions must be arithmetics definable by the semantics of the C language.

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. There are also latches for both control and data variables. Figure 1 shows an example of a control-data network with two latches, where bold wires indicate data variables.

Each control node is a multi-valued function represented in Multi-valued Decision Diagrams (MDDs) and sum-of-products. 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 and evaluates to 1 if the variable takes on any of the values in the set. A **product term** or **cube** is a conjunction of literals and evaluates to 1 if each of the literals evaluates to1. A **sum-of-products** (SOP) is the disjunction of a set of product terms and evaluates to 1 if any of the products evaluates to 1. For detailed definitions refer to [13].

A set of technology independent optimization methods have been implemented in MVSIS [8]. These include algorithms extended from binary logic: algebraic decomposition [12], don't care-based simplification [15], elimination, resubstitution; and also algorithms specifically tuned for multi-valued logic: node pairing and encoding [13].

4. SOFTWARE SYNTHESIS

The software synthesis problem is, given such a multi-level control-data network, generate an efficient software implementation in C (or target dependent machine language), such that appropriate constraints are satisfied. These may include resource constraints, like RAM and ROM usages, and timing constraints. This is a mapping problem from a parallel execution model, originally targeted for hardware, to a sequential execution model, where all tasks share the same processing unit. The goal is to generate fast sequential simulation code with constraints on the code size. This may be achieved by reducing the time spent on each node, and reducing the total number of nodes that has to be evaluated. We therefore solve two sub-problems: code generation for individual logic functions, and scheduling of node execution, detailed in the remainder of this section.

4.1 Code Generation

Given multi-valued logic function $f(x_1, \ldots, x_n) : P_1 \times P_2 \times \ldots \times P_n \rightarrow Q$, where P_i is the domain of input variable x_i and Q is the domain of the output variable, find software implementation such

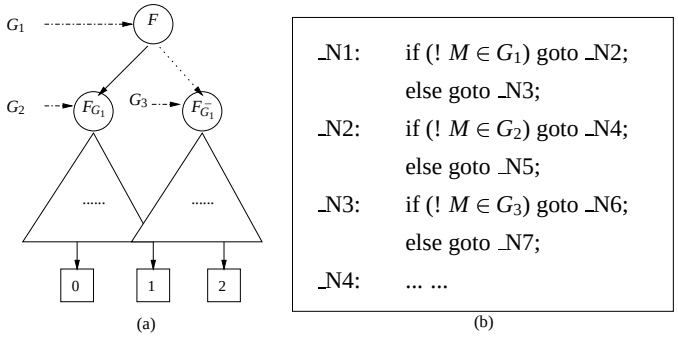

Figure 2: Generalized Cofactoring Diagram (a) and its generic software implementation (b)

that the code size and execution speed is optimized.

Different code generation schemes exist depending on various logic function representations: sum-of-products (SOP), multi-valued decision diagrams (MDDs), and lookup tables (LUT). We use a generic model called Generalized Cofactoring Diagram (GCD) to model and unify these different methods.

Similar to MDDs, GCD is a diagram composed of functional nodes and terminal nodes; different from MDDs, there is a set of general cofactoring functions $\{G_1, G_2, \ldots, G_n\}$ (rather than variables), one for each node in the structure, to be cofactored by the intermediate function nodes. As illustrated in Figure 2(a), the top node function F has two out-going edges, representing the cofactors F_{G_1} and $F_{\bar{G_1}}$. Generalized cofactoring for functions is defined as $F_G = F \cap G$. Note that in general, G_i can be multi-valued functions. In that case each value k of G_i is a binary function G_i^k. F would have as many output edges as the number of values for G, each representing the cofactoring of F on one of G_i's value functions: $F_{G_i^k}$.

The software derived from such a GCD takes an input minterm M and determines the output value based on the branching structure, as shown in Figure 2(b). Each branching point represents a sub-space of the input logic domain, and the cofactoring function G_i partitions this further into smaller domains, until the output value can be decided in the sub-space. The input minterm follows the GCD structure and makes branching decisions based on the result of $(M \in G_i)$. In the figure, label `_N1` refers to the code derived from the root node F; label `_N2` refers to the code derived from node F_{G_1}. Label `_N4` and beyond are not shown in the figure.

The code generation problem is to find the set of cofactoring functions $\{G_1, G_2, \ldots, G_n\}$ and build the cofactoring structure, such that the evaluation of the logic function in software is optimized. Long computation time can be tolerated for deriving the structure that gives the best performance, since the generated code is to be loaded in the final products and executed numerous times.

Note that unlike MDDs, GCDs need not be ordered so they are more like free MDDs. A similar optimization scenario based on free MDDs was proposed in [17]. The optimality is measured in code size, which is proportional to the total number of branching nodes, and speed, which is related to (a) the average depth of the diagram, and (b) the cost of the branching test $(M \in G_i)$.

Different code generation methods based on specific functional representations are variations of the GCD:

- **MDD**: Cofactoring functions G's are single variable functions. At each GCD node, the input minterm is tested for the value of a particular input variable. The evaluation speed is bounded by the number of inputs, but the size may be exponential depending on the logic function.

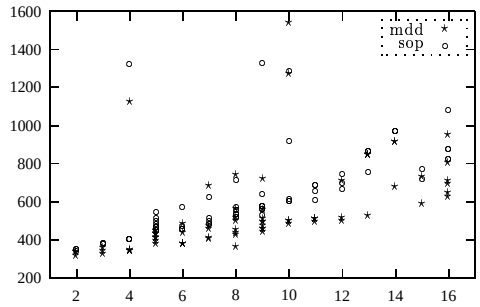

Figure 3: Comparison of code size w.r.t. the number of inputs

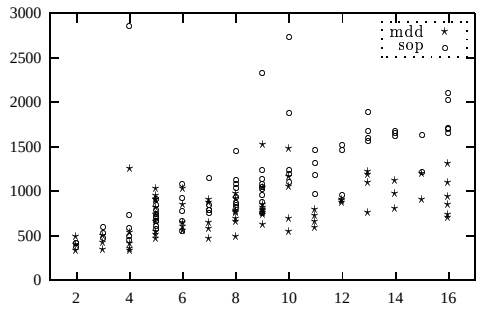

Figure 4: Comparison of run-time w.r.t. the number of inputs

- **SOP**: Cofactoring functions are single cubes. At each GCD node, the input minterm is checked to see if it is contained in the cube associated with that GCD node. The containment test is performed by a bit-wise *AND* operation if the cubes and minterms are represented in bit vectors [16], or by evaluating the value of the cube through Boolean operations. Both the size and the evaluation speed may be exponential depending on the logic function.

- **LUT**: There is only one GCD node that performs a memory lookup. There are as many entries in the memory as the number of input minterms. The output value is retrieved by one memory operations. The code size is exponential, but can be managed by decomposing into cascading LUTs; the evaluation speed is potentially limited by the memory latency of the system architecture. Experiments show that the memory bottleneck of common processor architectures prevents this method from being competitive compared with the others.

Figure 3 and 4 shows code size and run-time comparison of MDD and SOP code for some 150 logic functions extracted from a common benchmark suite, with input variables ranging from 2 to 16. The codes are generated and compiled with `gcc -O2` on an Intel PIII platform running Linux. The code size is the number of bytes in the object code; the run time results are the average of 10 million simulations on pseudo random input vectors. We make the following observations: (a) Both methods demonstrate a linear run-time degradation as the number of inputs grows. The SOP code degrades faster than the MDD code. (b) The SOP code is in general slower then the MDD code, except for only very small logic functions (less than 3 inputs) and occasionally large logic functions with small cube counts. (c) There are cases when the SOP code is faster or smaller then the MDD code. This indicates that a hybrid approach (such as GCDs) which derives a specialized cofactoring

structure for each logic function would produce consistently superior results.

The real runtime performance depends on the target processor architectures. Our result gives an approximate comparison of the code generation methods. New GCD structures based on optimal cube cofactoring is to be explored in future work. In the final experiments, we use MDD code for large logic functions and SOP code for small functions.

4.2 Scheduling

Given a network of nodes, as described in Section 3 (see Figure 1), find a scheduling scheme for the firing of the nodes, such that the overall execution time is minimized. A straight-forward approach is to statically schedule the nodes with a topological sort, similar to oblivious logic simulation. Every node in the network is always evaluated, even though sometimes they do not need to be.

Unlike hardware execution, where timing is characterized by the critical path delay, software execution time is associated with the total number of nodes that have to be evaluated for a particular input vector. A data expression node may not be evaluated if it is de-selected by its fanout multiplexer. (For example node `M1` and `E1` in Figure 1.) A control node or a predicate node may not be evaluated if its value is not observable at the primary outputs. In the remainder of this section, we develop theories for identifying these cases and develop techniques to make use of them in software scheduling.

4.2.1 Triggerability

To simplify the 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.

DEFINITION 1. *A node is called a trigger node (or triggerable) if there exists at least one input vector, which can determine the output values without this node.*

The triggerability of a node can be tested by the existence its Maximal Observability Don't Cares (MODC). The $MODC(x_i, y_k)$ for an edge from node x_i to node y_k is the set of minterms in the domain of all intermediate variables, such that the values of x_i cannot be observed at the output of node y_k. The MODC of a node x_i is the intersection of the MODCs for all its fanout edges [15]. The MODC of node y_k is also inherited by all of its fanin nodes. Expressed in equation:

$$MODC_{x_i} = \cap_{y_k \in fanout(x_i)} (MODC(x_i, y_k) \cup MODC_{y_k})$$

THEOREM 1. *A node is a trigger node if and only if its MODC set is not empty.*

If the MODC set of a node is empty, it is always observable at the outputs for all possible input combinations. This means they have to be evaluated for all cases in order to obtain the output value. We call these nodes *static* nodes.

For control nodes with multi-valued logic, the MODC theory and algorithms have been developed in [15]. For a multiplexer $f = f(y_c, y_0, \ldots, y_{n-1})$, where y_c is the control variable and y_i are data input variables, let $MODC_f$ be the MODC set computed for the output f. Then the MODC set for its fanin edges are:

$$\begin{aligned} MODC(y_i, f) &= (y_c \neq i) \cup MODC_f \\ MODC(y_c, f) &= MODC_f \end{aligned}$$

Intuitively, the control variable has to be evaluated in order to do the multiplexer assignment, therefore its MODC directly inherits from node f; a data variable is blocked from the output of f, if it is de-selected by the control variable.

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

$$MODC(y_i, f) = MODC_f$$

In other words, an expression node is a trigger node if and only if all of its fanouts are trigger nodes. We use the following theorem to simplify the analysis of the data-path triggerability:

DEFINITION 2. *A trigger tree is a largest set of predicate and expression nodes, among which only one node fans out to the external world. This node is called the root node of this trigger tree.*

COROLLARY 1. *All nodes in a trigger tree share the same triggerability.*

A trigger tree is treated as a single node in our analysis and code generation. In the example in Figure 1, node `P2` and `E2` are combined as a trigger tree.

4.2.2 Triggerized Scheduling

After the triggerability of all nodes in the network are computed, we develop a quasi-static execution schedule based on this information. We first outline some features of the scheduling scheme and then give a general algorithm.

The generated code has two portions: (a) a static portion consisting of static nodes generated in a topological order from inputs to outputs; (b) a dynamic portion consisting of trigger nodes. A trigger node is generated as a "cheap" subroutine call (meaning the only overhead is storing the return address). They are executed *on-demand* as requested by their fanout nodes.

Because calling a trigger node has overhead, we use a simple heuristic to detect very small trigger nodes (less than three inputs), and generate the code on the spot, rather than using the subroutine call.

```
Algorithm [Network Code Generation]:
input: control-date-network
Partition network into trees and compute triggerability;
generate_header();
List1 = topological list of static trees;
Foreach tree i in List1
    code_gen_tree(i);
generate_tail();
List2 = topological list of trigger trees;
Foreach tree i in List2
    code_gen_tree(i);
End
code_gen_tree [Tree Code Generation]:
input: trigger tree
Foreach node k in tree in topological order
    If k is data node
        Generate expression; Continue;
    Build GCD for logic function k;
    Traverse each node g in GCD
        If g requires value from trigger node
          If cost(g) < threshold
            code_gen_tree(g);
          else
            Generate function call for g;
        Generate branching code for g;
End
```

Figure 5: Code generation with triggerized scheduling

Trigger nodes may be invoked more than once depending on its number of fanouts. A trigger node is called *level_1* triggerable if it has only one fanout node; otherwise it is called *level_2* triggerable. We create a one bit flag for each level_2 trigger node to indicate the status of its execution. Before firing this trigger node, the associated flag is tested and the subroutine is called only if it has never been evaluated before. This again creates additional overhead for level_2 trigger nodes, which is also considered in the screening heuristic. The fact that data nodes are combined into trigger trees increases the size of triggered subroutines, which amortizes the overhead cost.

Figure 5 gives the overall procedure of the code generation scheme. Function call `generate_header()` refers to variable declaration and state initialization; `generate_tail()` refers to the update of latch variables and returning evaluation results. Generating expressions for data nodes is straight-forward for predicates and data expressions, but also involves trigger node testing for multiplexers. It is relatively simple, and hence not detailed in the figure.

A control node is treated as a tree by itself. The GCD structure for a control node is built from either an MDD or a SOP representation, depending on their estimated cost according to the size of the logic function. We use an estimation heuristic based on the number of inputs and number of product terms. At each GCD node, the cofactoring function (single variable for MDD and single product for SOP), is tested for triggerability. If the variables required are coming from trigger nodes, then they would be fired, either through cheap function call or on-the-fly computation. And then the bit-flag for level_2 trigger nodes would be updated.

4.3 Event Driven Simulation

The notion of triggerability explores statically the observability of a node with respect to its transitive fanouts, which extends the idea of back-tracing simulation [23]. Event driven simulation technique [18] tries to achieve the same goal (reducing the number of nodes that need to be evaluated), by dynamic scheduling according to input activities.

As a comparison we apply event-driven techniques to the static nodes in the network. Here a more conservative definition of static node is used instead of the one defined in the previous section. At a given node, if there is no activity among the static fanin nodes, the trigger fanin nodes will not be evaluated even if there might be activity in them. Therefore, for static node k, if k is a control node, all its fanins are labeled as static; if k is a multiplexer, only its control input is labeled as static.

In the generated code, we create a bit vector, one bit for each static node, indicating the activity at its local inputs. The evaluation of a static node then includes additional overhead of checking activity at local inputs and propagating activity to all fanout nodes, which proves to be useful in speeding up the code. Below shows the pseudo code of evaluating node η with input minter M.

eval_node (η, M):
1 **if** (!activity(η)) return;
2 **if** (result_changed(eval_logic (M));
3 **foreach** fanout k of η
4 set_activity(k);
End

`eval_logic()` contains the actual code for evaluating the logic function (or data black-box). `activity()` checks the event bit of the corresponding node. These are implemented as fast inline function calls. As an illustration, the example in Figure 1 (without observing the logic in each node) produces the scheduling below, where `E1` and `M1` are trigger nodes:

Table 2: Results on execution speed on 10 million pseudo random inputs

Examples	strl	E-auto	E-sort	E-opt	MVSIS
fuel-ctr	103	420	10100	6360	5880
pulse-ctr	132	430	12400	9890	7770
instr-ctr1	215	270	28770	8200	5800
instr-ctr2	240	275	30860	8800	6290
dma-ctr	455	400	114000	11900	8510

```
update_latch(L1); update_latch(L2);
eval_node(C1); eval_node(C2); eval_node(P1); eval_node(C3);
eval_node(E2); eval_node(P2); eval_node(C4);
if (C3) { M2 = L1; }
else {
    if (C2) { eval_logic(E1); M1 = E1; M2 = M1; }
    M1 = w; M2 = M1;
}
```

5. EXPERIMENTS

In the experiment, we use Esterel as a high-level specification language, and the Esterel compiler to parse the input Esterel program and produce an intermediate circuit representation called DC. We translate DC into our intermediate control-data network representation in an extended BLIF-MV format. Implementation C code is generated after multi-valued logic optimization.

The examples consist of an electronic engine fuel controller, an instruction decoder (both acyclic and cyclic version), and a direct memory access controller. Table 2 shows the average execution speed of the compiled code on 10 million pseudo random input vectors. Column `strl` shows the lines of Esterel source code, indicating the size of these applications. `E-auto` shows the code generated by the Esterel compiler from an automaton representation, which for large examples tends 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 [20] called `Basicopt`, which consists of binary combinational area optimizations, state encoding and latch removal [21]. If we treat the optimized circuits after `Basicopt` as input, and then optimize further in MVSIS to generate code, the results are shown in column `MVSIS`. The results are obtained on a Intel PIII platform with a Linux 6.1 operating system and 128MB RAM. We use GNU tool "`gprof`" to obtain the average run time consumed by the core evaluation code. As shown, the code generated from MVSIS has faster evaluation speed over `E-opt`. Both the triggerability and event driven approach are experimented and the best results are reported here. Table 3 shows the object code size in bytes after compiled by "`gcc -c -O2`". In general, the code size generated by our approach is larger due to the overhead of back-tracing and event propagation.

Discussion about the experiments:

- These examples are originated from hardware systems (microprocessors and DMAs). However the functionality is representative for controller designs (80-90% control).
- The DC circuit derived from the Esterel compiler is pure binary, and includes both combinational and sequential redundancies. MVSIS is able to rebuild MV variables through node pairing and merging, but it is hard to reconstruct the original program structure. Also MVSIS has limited sequential optimization capability.
- Automata-based code is very fast, because there is a piece of specialized code tailored for each reachable state. However, the code size explodes with the state space. The structural (circuit) representation is very compact with the state

Table 3: Results on code size compiled by `gcc-O2`

Examples	E-auto	E-sort	E-opt	MVSIS
fuel-ctr	2239	2565	2141	2514
pulse-ctr	1691	3525	2845	3400
instr-ctr1	65111	9997	2981	4380
instr-ctr2	65111	10905	3349	4460
dma-ctr	220783	23237	5665	5776

space encoded into a limited number of latches, but this has much slower evaluation speed. We experimented with merging binary latches to create multi-valued ones. This does give some run-time improvements, but exploring the state encoding space for fast simulation is still an open issue.

6. CONCLUSIONS AND FUTURE WORK

The methodology of high-level programming based on a functional model is inevitable in future embedded software applications, especially for mission critical discrete control as in avionics. For these applications, we adopt a design flow centered around a particular model of computation, EFSMs. This model supports high-level specification with synchronous languages (with data types and function calls), and supports hardware/software partitioning and mapping onto a set of architecture platforms.

We developed a framework for generating efficient software implementation from synchronous specifications. A generic setting called Generalized Cofactor Diagram (GCD) was proposed for evaluating individual logic functions. We developed the theory of triggerability for a quasi-static scheduling scheme, and gave an algorithm to generate code. Event-driven simulation techniques are also applied in the code generation phases, and prove to be useful in deriving faster code. Preliminary experiments are performed to analyze our first attempt on some of the algorithms in the framework.

A few algorithms need to be fine-tuned for the software synthesis framework. Logic evaluation of individual functions can be improved by searching for an optimal generalized cube cofactoring structure. Applications that are sequentially very deep (large number of latches) can benefit from exploring the state re-encoding space, or generalized cofactoring on the latch variables.

In the top-level design flow, we have experimented on mapping EFSMs to a reconfigurable architecture platform that has a specialized data-path augmented with reconfigurable FPGA instructions. The mapping onto other standardized system-on-chip architectures, especially the automatic synthesis of the communication scheme between hardware and software, seems to be a promising research direction.

Acknowledgement

The authors would like to acknowledge Max Chiodo for providing an intermediate format and its parser from Esterel DC. We are grateful for the support of the SRC under contract 683.004 and the California Micro program and our industrial sponsors, Fujitsu, Cadence, and Synplicity.

7. REFERENCES

[1] P. Ashar and S. Malik. Fast functional simulation using branching program. In *Proc. of the Intl. Conf. on Computer-Aided Design*, pages 408–412, Nov. 1995.

[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] M. Baleani, F. Gennari, Y. Jiang, Y. Patel, R. K. Brayton, and A. Sangiovanni-Vincentelli. Hw/sw partitioning and code generation of embedded control applications on a reconfigurable architecture platform. In *Proc. of the Intl. Symposium on Hardware/Software Co-Design*, May. 2002.

[5] G. Berry. Esterel on hardware. *Philosophical Transactions of the Royal Society of London. Series A*, 1992.

[6] G. Berry. *The constructive semantics of pure Esterel*. Book in preparation, 1996.

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

[8] R. K. Brayton and et al. MVSIS. `http://www-cad.eecs.berkeley.edu/mvsis`.

[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. Edwards. Compiling esterel into sequential code. In *Proc. of the Design Automation Conf.*, June 2000.

[11] The ESTEREL language. [On-line] http://www.esterel.org.

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

[13] 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.

[14] O. Hainque, L. Pautet, Y. L. Biannic, and E. Nassor. Cronos: a separate compilation toolset for modular esterel applications. *Formal Methods*, 1999.

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

[17] C. Kim, L. Lavagno, and A. Sangiovanni-Vincentelli. Free MDD-based software optimization techniques for embedded systems. In *Proc. of the Conf. on Design Automation & Test in Europe*, Mar. 2000.

[18] P. M. Maurer. Event driven simulation without loops or conditionals. In *Proc. of the Intl. Conf. on Computer-Aided Design*, Nov. 2001.

[19] P. McGeer, K. McMillan, A. Saldanha, A. Sangiovanni-Vincentelli, and P. Scaglia. Fast discrete function evaluation using decision diagrams. In *Proc. of the Intl. Conf. on Computer-Aided Design*, pages 402–407, Nov. 1995.

[20] 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.

[21] 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.

[22] M. Sgrio, L. Lavagno, Y. Watanabe, and A. L. Sangiovanni-Vincentelli. Synthesis of embedded software using free-choice petri nets. In *Proc. of the Design Automation Conf.*, June 1999.

[23] R. Ubar, J. Raik, and A. Morawiec. Back-Tracing and Event-Driven Techniques in High-Level Simulation with Decision Diagrams. In *Proc. of the Intl. Symposium on Circuits and Systems*, pages 208–211, May 2000.

[24] D. Weil, V. Bertin, E. Closse, M. Poize, P. Venier, and J. Pulou. Efficient compilation of Esterel for real-time embedded systems. In *Proc. of the Intl. Conf. on Compilers, Architecture and Synthesis for Embedded Systems*, Nov. 2000.