SYNTHESIS OF SOFTWARE PROGRAMS FOR EMBEDDED CONTROL APPLICATIONS

by

Massimiliano Chiodo, Paolo Giusto, Attila Jurecska,
Luciano Lavagno, Kei Suzuki, Ellen Sentovich,
Harry Hsieh, and Alberto Sangiovanni-Vincentelli

Memorandum No. UCB/ERL M94/87

1 November 1994

SYNTHESIS OF SOFTWARE PROGRAMS FOR EMBEDDED CONTROL APPLICATIONS

by

Massimiliano Chiodo, Paolo Giusto, Attila Jurecska,
Luciano Lavagno, Kei Suzuki, Ellen Sentovich,
Harry Hsieh, and Alberto Sangiovanni-Vincentelli

Memorandum No. UCB/ERL M94/87

1 November 1994

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

SYNTHESIS OF SOFTWARE PROGRAMS FOR EMBEDDED CONTROL APPLICATIONS

by

Massimiliano Chiodo, Paolo Giusto, Attila Jurecska,
Luciano Lavagno, Kei Suzuki, Ellen Sentovich,
Harry Hsieh, and Alberto Sangiovanni-Vincentelli

Memorandum No. UCB/ERL M94/87

1 November 1994

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

Synthesis of Software Programs for Embedded Control Applications

Abstract

Software components for embedded reactive real-time applications must satisfy tight code size and run-time constraints. Cooperating Finite State Machines provide a convenient intermediate format for embedded system co-synthesis, between high-level specification languages and software or hardware implementations. We propose a software generation methodology that takes advantage of the very restricted class of specifications and allows for tight control over the implementation cost. The methodology exploits several techniques from the domain of Boolean function optimization. We also describe how the simplified control/data-flow graph used as an intermediate representation can be used to accurately estimate the size and timing cost of the final executable code.

1 Introduction

In this paper we address the problem of synthesizing efficient software for embedded reactive real-time systems. Such systems are in general composed of software and hardware components, and the software has tight memory-size and execution-speed constraints.

In particular, we focus on *control-dominated* embedded systems, where the emphasis is on the decision process that leads from a set of input events to a set of output events (reaction). This class of systems covers a fairly broad range of applications, from microwave ovens and watches to telecommunication network management and control functions. Finite State Machines (FSMs) provide a convenient and common mechanism for specifying the intended behavior of such systems. FSMs are either explicitly used as input (both in graphic and in textual form [HLN+90], [SSR89]), or as intermediate format ([Hal93], [WTHM92]) of embedded system design tools. Although we use the Codesign Finite State Machine (CFSM) model defined in [CGH+93a, CGH+93b] for the sake of explanation, our results can be applied to any FSM-based specification.

The use of FSMs for embedded control specification offers several advantages over apparently more powerful formalisms (such as unrestricted programming languages). First of all, they are easily understood and widely used even as informal specifications. Secondly, there are abundant theoretical and practical results concerning their manipulation (minimization, encoding, formal verification of properties, ...).

Unfortunately "pure" FSMs don't provide a very convenient representation for systems that perform even a small amount of computation. It is customary to extend them with the capability to perform assignments of expressions to variables, and to use comparisons to determine transition conditions. This mechanism increases the expressive power at the expense of the synthesis and verification capabilities (e.g., there is no longer a "canonical" form for such extended FSMs, verification becomes much more difficult, etc.).

The purpose of this paper is to describe algorithms for an *optimizing compiler* from an FSM specification to object code on a micro-controller. This compiler is not to be compared against traditional compilers from a programming language like C or Pascal to object code, because we are solving a much simpler and more restricted problem. But exactly for this reason we can afford to perform optimizations that are either impossible or simply too expensive in the general case ([ASU88]). We can even tightly couple the optimization process with a fast and accurate timing and code size estimation procedure, to take into account constraints at a much finer granularity level than is possible with a truly general-purpose compiler.

Throughout this paper we make the following main assumptions:

1. The standard resource allocation and operation scheduling steps have already been performed on the high-level specification ([CW91]). A global approach, where resource allocation, scheduling and control implementation are all considered simultaneously is left to future research.

2. A Real Time Operating System is used to activate appropriately the tasks implementing the FSMs, ensuring the satisfaction of timing constraints that span more than one FSM transition. Our synthesis procedure, on the other hand, provides the Operating System with execution time estimates that can be used either off-line or on-line to schedule the FSMs. We are investigating means to allow an off-line scheduling procedure to propagate timing constraints

to the compiler so that scheduling becomes easier (or feasible at all).

3. A general-purpose (but machine-specific) compiler is used to transform the code that we produce into machine code. This allows us to concentrate on domain-specific transformations, while leaving general ones such as register allocation and instruction selection to the general-purpose compiler. Note that the C code that we produce is so simple and low-level that we can keep a very tight control over the resulting machine code, and the compiler cannot "undo" our optimizations.

We use, like most compilation strategies, a control/data-flow diagram (called *s-graph*, for software graph, in the following) as an intermediate data structure. The s-graph is simpler than general control/data-flow diagrams, because it needs only to represent a single function from a discrete domain (the set of input events and values) to a discrete domain (the set of output events and values). As such, it requires only conditional branch and assignment as primitives (augmented with arithmetic and relational expressions without side effects). This simple representation has a straightforward representation in C and can be translated with equal ease into object code by any available compiler. In this way we can obtain good cost and performance estimates at any intermediate stage of the optimization process, without the need to compile the code and analyze the results.

Our software synthesis procedure is composed of the following main steps:

1. Translation of a given CFSM into an s-graph.
2. S-graph optimization.
3. Translation of the s-graph into a target language.
4. Scheduling of the CFSMs.
5. Compilation into machine code to be run on the target processor.

Step 1 uses Binary Decision Diagrams (BDDs, [Bry86]) as an intermediate representation, to generate a *very fast* initial s-graph, potentially at the expense of code size. It is based on a new result, described in this paper, that states the equivalence between:

- a multioutput multivalued function f and
- an s-graph that is directly obtained from a BDD representing f.

Steps 2 and 4, which are currently being implemented, will use the software performance estimation package to minimize the code size and meet the given timing constraints.

The paper is organized as follows. Section 2 contains background information and the s-graph structure definition. Section 3 describes the software cost and performance estimation technique based on s-graphs. Section 4 describes the s-graph synthesis and optimization procedure. Section 5 shows some experimental results demonstrating the effectiveness of the approach.

2 Preliminaries

2.1 Previous Work

Previous approaches to automated code generation for reactive real-time systems have started either from synchronous programming languages (e.g., Esterel, [BCG91]), or from other high-level languages ([CWB94, GJM94]).

In the first case, the main problem is the identification of a *single* FSM equivalent to the Esterel specification, and its efficient implementation as a software program. The approach has the advantage of producing a very *fast* implementation (as all the internal communication between modules disappears when the single FSM is produced), at the expense of code size. Our approach allows a finer trade-off between size and speed, because the designer can choose the granularity of the generated CFSMs, even if they are produced from an Esterel specification ([Yee94]). Moreover, the currently distributed version of the Esterel compiler (V3) produces C code representing the FSM in tabular format, with a fast interpreter for it. Our software implementation strategy, on the other hand, produces an implementation of each transition function in machine code, thus relying on the efficiency of the micro-controller instruction set coding.

In the second case, the main emphasis is on the *scheduling* of operations derived from a concurrent high-level specification (e.g., hardware-C, [KM92]). The problem is that of choosing an order for potentially concurrent operations that satisfies the given timing constraints. In our case, we decompose the problem of satisfying timing constraints into two (possibly iterated) steps:

1. code generation for each CFSM,
2. scheduling of CFSM transitions to satisfy timing constraints.

The difference is that we can take advantage of the large body of research about scheduling for real-time systems (e.g., [LL73]) for the second step. On the other hand, some of the algorithms of [GJM94] can also be used to perform a preliminary optimization before our synthesis algorithm. This would allow an easier satisfaction of "short term" timing constraints (e.g., those dictated by a specific interface protocol implemented directly in software) which may be more difficult to satisfy with classical scheduling techniques (designed for "long term" response and input rate constraints).

2.2 Binary Decision Diagrams and Characteristic Functions

A *Binary-Decision Diagram* (BDD, [Ake78, Bry86]) is an efficient representation for storing and manipulating Boolean functions. A BDD is a directed acyclic graph with a root node for each output function and leaf nodes representing the value of each function for each input minterm. Each non-leaf node represents an input variable, and each of the two out-edges of the node represents the value of the variable (0 or 1) along that branch. The representation is made compact (*reduced*) by sharing common functional subgraphs. Given a function f and an *ordering* of the input variables, the Reduced Ordered BDD (simply called BDD in the following) is a canonical form for f.

While the size of the BDD may be exponential in the number of inputs for any ordering, in many practical cases a good ordering can be found that produces a polynomially-sized BDD. Functional operations on the BDD take at most n^2 space and time (n is the number of nodes in the BDD); equivalence checking between two BDDs requires only a graph isomorphism check. The canonicity property of BDDs, efficient BDD package implementation, and recent improvements in variable ordering strategies have made BDD-based algorithms efficient and effective for a variety of problems involving Boolean function manipulation.

Multioutput functions (or, equivalently, sets of functions defined on the same domain) can be represented by their *characteristic functions*. A single-output binary-valued function $\chi^f : (X \times Y) \rightarrow \{0, 1\}$, where $X = X_1 \times \ldots X_m$ and $Y = Y_1 \times \ldots Y_l$, represents the multioutput multivalued function $f : X \rightarrow Y$ if $\chi^f(x, y) \Leftrightarrow (y = f(x))$. The same notation can also be used to describe a *Boolean relation* ([BS89]) R, as $\chi^R(x, y) \Leftrightarrow (y \in R(x))$

The function resulting when some argument x_j of a function f is replaced by a constant b is called a *restriction* (or *cofactor*) and is denoted $f_{x_j=b}$. The projection of a function f onto a space orthogonal to x_j (or *Smoothing* of f by x_j) is denoted $S_{x_j}f$. That is, if $x_j \in \{0, 1\}$, then $S_{x_j}f = f_{x_j=1} \vee f_{x_j=0}$.

The *support* of an output variable y_i of a multioutput function is the set of inputs upon which y_i *essentially depends*. More precisely, an input variable x_j belongs to the support of y_i if $S_{x_j}y_i(x_1, \ldots x_n) \neq y_i(x_1, \ldots x_n)$.

2.3 Codesign Finite State Machines

According to [CGH$^+$93a, CGH$^+$93b], a CFSM is a reactive Finite State Machine with a set of input and output events. Events are entities that may occur at determinate instants of time and may or may not carry a value (cfr. the notion of *signal* in Esterel). For the purpose of this discussion, each event is associated with a binary-valued variable which is true in the time interval between its emission and its detection, and with an optional discrete-valued variable carrying its value. The value is defined only when the associated binary-valued variable is true, and remains unchanged until a new event of the same type occurs. In the following, variables denoting the presence of an event are identified by the event name preceded by a "*", while variables denoting its value are identified just by the event name. An input event to a CFSM may be *detected* at most once at any time after its emission. The logical reaction is determined by the *transition function* of the CFSM. The transition function maps the set of input events and values onto the set of output events and values: $f : X_1 \times \ldots X_m \rightarrow Y_1 \times \ldots Y_l$.

2.4 Software Graphs

In this section we define more precisely the control-flow graph that we use as internal representation of the CFSM transition function.

Definition 1 *An s-graph G is a* directed acyclic graph *(DAG) with one source and one sink. Its vertex set V contains four types of vertices:* BEGIN, END, TEST, *and* ASSIGN. *The source has type* BEGIN, *the sink has type* END. *All other vertices are of type* TEST *or* ASSIGN. *Each* TEST *vertex v has two children true(v) and false(v)*[1]. *Each* BEGIN *or* ASSIGN *vertex u has one child next(u) only. Any non-source vertex can have one or more parents. Each vertex is labeled with a function defined on a set of discrete finite-valued variables* $z_1, \ldots z_j, \ldots z_n$*:*

[1] The implementation described in Section 5 allows more than two children. The extension of the definitions and theorems to the more general case is trivial.

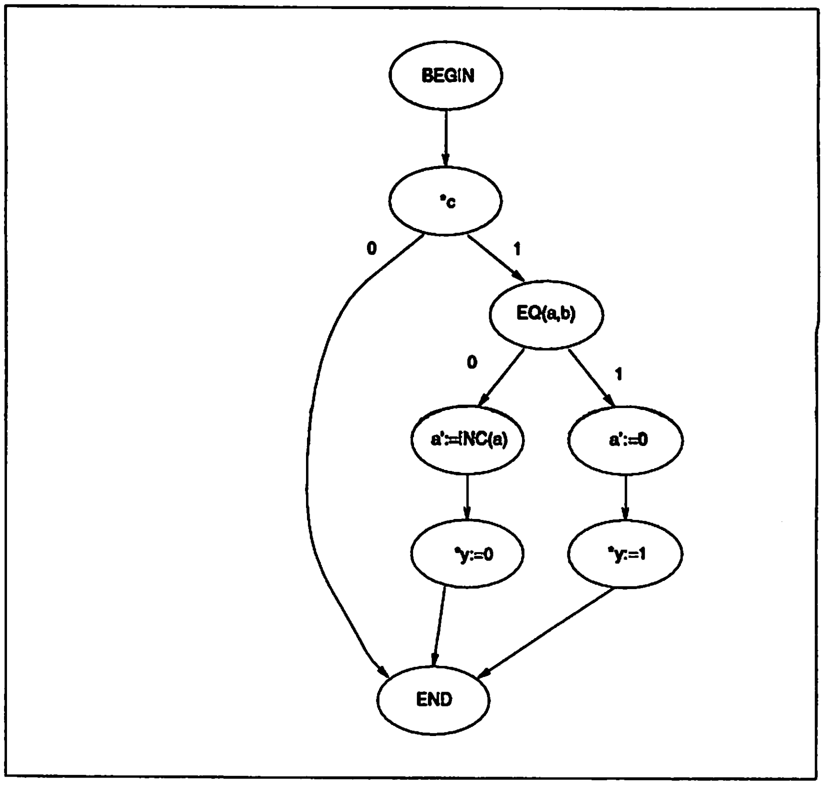

Figure 1: A simple s-graph

- *An* ASSIGN *vertex v is associated with a function $a_v(z_1, \dots z_j, \dots z_n)$ and an output variable z_j (intuitively, it assigns the result of $a_v(z_1, \dots z_j, \dots z_n)$ to z_j).*
- *A* TEST *vertex v is associated with a predicate $p_v(z_1, \dots z_n)$.*

A simple s-graph is shown in Figure 1.

The s-graph model resembles *branching programs* ([Lee59], [Mei89]) and *Binary Decision Diagrams*. Both branching programs and BDDs are different from s-graphs because they allow:

- only single-variable predicates on TEST nodes and
- only assignments to a single output variable (as the last level of nodes).

We will see in Section 4, though, that there is a close connection between a BDD representation of a CFSM transition function and an s-graph computing it.

The definition of the multioutput function computed by an s-graph with BEGIN node v is given by the following algorithm:

```
procedure **evaluate** (v: vertex; $x_1, \dots x_m, y_1, \dots y_l$:variable)
begin
    $z_j \Leftarrow x_i$ for $1 \leq j \leq m$
    $z_j \Leftarrow \epsilon$ for $m + 1 \leq j \leq m + l$
    **eval** (next(v), $z_1, \dots z_{m+l}$)
```

```
end

procedure eval (v:vertex; $z_1, \ldots z_{m+l}$:variable)
begin
    if $v$ is a TEST
    then
        if $p_v(z_1, \ldots z_{m+l})$
        then eval (true(v), $z_1, \ldots z_{m+l}$)
        else eval (false(v), $z_1, \ldots z_{m+l}$)
    else if $v$ is an ASSIGN
    then
        $z_j \Leftarrow a_v(z_1, \ldots z_{m+l})$
        eval (next(v), $z_1, \ldots z_{m+l}$)
    else ($v$ is END)
        $y_j \Leftarrow z_{m+j}$ for $1 \leq j \leq l$
end
```

Definition 2 *An s-graph is* functional *if every (output) variable z_{m+j}, for $1 \leq j \leq l$:*

1. *is assigned by* **eval** *at least one defined value for each combination of values of the input variables, and*

2. *has a defined value whenever a predicate or a function depending on z_{m+j} is visited by* **eval**.

Definition 3 *Let G be a functional s-graph, with* BEGIN *vertex v. Let $z = (x_1, \ldots x_m, y_1, \ldots y_l)$ be a vector of variables appearing in the vertex labels of G ($x_i \in X_i \cup \{\epsilon\}$ and $y_j \in Y_j \cup \{\epsilon\}$, where ϵ is a distinguished "undefined" value). Then G denotes a function $f : X_1 \times \ldots X_m \rightarrow Y_1 \times \ldots Y_l$ computed by procedure* **evaluate** *$(v, x_1, \ldots x_m, y_1, \ldots y_l)$.*

It is easy to show that the definition is consistent (i.e., that f is indeed a completely specified function), and that a *non-functional* s-graph computes:

- either an incompletely specified function (if condition 1 in Definition 2 is violated),

- or a *relation* between the input and the output variables (if condition 2 in Definition 2 is violated).

This fact can be used in optimizing the s-graph, as briefly explained in Section 4.

If we apply procedure **evaluate** to the s-graph of Figure 1, we see that it defines a function which is triggered by event $*c$ and increments a until it equals b (assuming that a and a' are externally connected in feedback, e.g. by the Operating System). Then it emits $*y$ and resets a to 0.

The next two sections describe how the timing and size of the code generated from an s-graph can be accurately estimated, and how an s-graph implementing a given transition function can be built.

3 Software Cost and Performance Estimation

Hardware/software partitioning and software synthesis for real-time embedded systems require accurate and quick estimates of code size and of minimum and maximum execution time.

There are at least two aspects of the problem that we must consider here:

1. the structure of the code, e.g., loops and false paths[2].
2. the system on which the program will run, including the CPU (instruction set, interrupts, etc.), the hardware architecture (cache, etc.), the operating system, and the compiler.

The s-graph structure is very similar (as shown more precisely in Section 4.2) to the final code structure, and hence helps in solving the problem of cost and performance estimation:

- each vertex in an s-graph is in one-to-one correspondence with a statement of the synthesized C code, and
- the form of each statement is determined by the type of the corresponding vertex.

This means that the resulting C code is poorly structured from a user's point of view, but is simple enough so that the effects of the target system on the execution time and code size of each vertex type can be easily determined, as described below.

The timing analysis is drastically simplified, because the s-graph implements the FSM transition function: looping is dealt with at the Operating System level. Moreover, false paths can be determined with a good degree of accuracy from the structure of the CFSM network, e.g. by computing event incompatibility relations.

Cost estimation can hence be done with a simple traversal of the s-graph. Costs are assigned to every vertex, representing its estimated execution cycle requirements and code size (including most effects of the target system).

3.1 Cost Estimation on the S-graph

Our estimation method consists of two major parts. First, we determine the cost parameters for the target system. Secondly, we apply those parameters to the s-graph, to compute the minimum and maximum number of execution cycles and the code size.

Each vertex is assigned two specific cost parameters (one for timing and one for size), depending on the type of the vertex and the type of the input and/or output variables of the vertex. Edges may also have an associated cost, as the *then* and *else* branches of an *if* statement generally take different times to execute. Currently, we use seventeen cost parameters for calculating execution cycles, fifteen for code size, and four for characterizing the system (e.g., the size of a pointer).

The parameters are determined for each target system (CPU, memory/bus architecture, compiler) with a set of sample benchmark programs. These programs are written in C, and consist of about 20 functions, each with 10 to 40 statements. Each *if* or assignment statement which is

[2] A path in the control flow graph is false if it can never be executed, e.g., due to conflicting Boolean conditions.

contained in these functions has the same style as one of the statements generated from a TEST or ASSIGN vertex. The value of each parameter is determined by examining the execution cycles and the code size of each function. A profiler or an assembly level code analysis tool, if available, can be used for this examination (we are currently using an internally developed cycle calculator for the Motorola 68HC11 micro-controller, as well as the *pixie* tool for MIPS CPUs).

The calculation of software performance is based on graph traversing algorithms. The minimum execution cycle requirement is calculated by finding a minimum cost path (based on Dijkstra's shortest path algorithm) from the BEGIN to the END vertex of the s-graph. Similarly, the maximum execution cycles are calculated by finding a maximum cost path (based on the PERT longest path algorithm). The code size is calculated simply by summing up all the code size parameters for all the vertices of the s-graph. The computational complexity of those graph traversing algorithms is $O(ElogN), O(E)$ and $O(E)$ respectively (where E is the number of edges, and N is the number of vertices in the s-graph).

4 S-graph Implementation and Optimization

We have shown in the previous sections how an s-graph computes a function, and how its timing and code size characteristics can be estimated. In this section we describe the s-graph synthesis and optimization algorithms more in detail. The currently implemented procedure is as follows:

1. Initial s-graph implementation from the transition function of an FSM.
2. S-graph optimization.
3. Translation of the s-graph into C code.

The next sections are devoted to an explanation of each step.

4.1 Initial S-graph Implementation

The first step of the software synthesis procedure derives an s-graph implementing the transition function of a given CFSM. The method is based on the Shannon decomposition and is given assuming that all variables are binary $x_i \in \{0, 1\}$ (the next section describes how this can be extended to multivalued variables). It takes as input an *arbitrary* ordering $z_1, \ldots z_{m+l}$ on the input and output variables, a variable index i (initialized as 0 at the root of the recursive call), and the characteristic function χ^f of the transition function. As we will see below, the choice of the ordering influences the form of the final s-graph (specifically the relative mix of TEST and ASSIGN nodes).

```
procedure build($z_1, \ldots z_{m+l}$:variable; $i$:index; $F$:function)
begin
    if $i = 0$
    then
        create a BEGIN vertex $v$
        next($v$) ⇐build($z_1, \ldots z_{m+l}$, 1, $F$)
    else if $F = 1$
    then
```

```
            create or retrieve the END vertex $v$
        else if $z_i$ is an input
        then
            create a TEST vertex $v$ with $p_v = (z_i)$
            true($v$) $\Leftarrow$build($z_1, \ldots z_{m+l}$, $i+1$, $F_{z_i=1}$)
            false($v$) $\Leftarrow$build($z_1, \ldots z_{m+l}$, $i+1$, $F_{z_i=0}$)
        else if $z_i$ is an output
        then
            create an ASSIGN vertex $v$ labeled with $z_i$ and $a_v = S_{z_{i+1} \ldots z_{m+l}} F_{z_i=1}$
            next($v$) $\Leftarrow$build($z_1, \ldots z_{m+l}$, $i+1$, $S_{z_i} F$))
        return $v$
end
```

We can now show the correctness of this algorithm.

Theorem 1 *Let $\chi^f(x_1, \ldots x_m, y_1, \ldots y_l)$ be the characteristic function of multioutput function f, such that $y_k = f_k(x_1, \ldots x_m)$, and let $z_1, \ldots z_{l+m}$ be an arbitrary total ordering of its variables. Then the s-graph G returned by procedure* **build***$(x_1, \ldots, x_m, y_1, \ldots, y_l;\ 0;\ \chi^f)$ computes f.*

Proof Consider an assignment of values $(c_1, \ldots c_m)$ to the input variables $x_1, \ldots x_m$ of f, and an arbitrary output $y_k = f_k(x_1, \ldots x_m)$.

Traverse the s-graph up to an ASSIGN node v labeled with y_k (there exists exactly one such node on any path from BEGIN to END due to the way the s-graph is built). Suppose, without loss of generality, that TEST nodes labeled with the first n input variables, and ASSIGN nodes labeled with the first $k-1$ output variables have been traversed. When node v is created, $F = S_{y_1, \ldots y_{k-1}} \chi^f_{x_1=c_1, \ldots, x_n=c_n}$. Hence $a_v = S_{y_1, \ldots y_{k-1}, y_{k+1}, \ldots y_m} \chi^f_{x_1=c_1, \ldots, x_n=c_n, y_k=1}$ (a function which depends on the unvisited inputs $x_{n+1}, \ldots x_m$ only). The value of a_v is 1 if and only if $f_k(c_1, \ldots c_m) = 1$ (i.e., the value assigned to y_k is correct), because (1) by definition the characteristic function is $\chi^f = \prod_{k=1}^{l} (y_k \equiv f_k(x_1, \ldots, x_m))$; (2) the smooth function distributes over functions that are independent of the smoothing variable: $S_x(f \cdot g) = f \cdot (S_x g)$ if f is independent of x. (1) and (2) imply that $a_v = S_{y_1, \ldots y_{k-1}, y_{k+1}, \ldots y_m} \chi^f_{x_1=c_1, \ldots, x_n=c_n, y_k=1} = f_k(x_1 = c_1, \ldots x_n = c_n, x_{+1}, \ldots x_m)$. ■

Note that the mapping from the transition function to the s-graph is not unique since it is based on the ordering of the variables. Section 4.1.2 discusses the influence of this choice over the timing and size characteristics of the generated code.

Moreover, the input to this algorithm need not always be a *function*, but could also be a *relation* (e.g., when non-determinism is used to describe design freedom, or *don't cares*). In that case, with a modification to the ASSIGN function a_v as indicated below, there may be cases in which the value assigned to z_i is not a constant, but is a function of $z_i, \ldots z_{m+l}$. The simplest case, when $S_{z_{i+1} \ldots z_{m+l}} \chi^f = z_i$, corresponds to a classical "don't care", because z_i can be assigned any value (including the cheapest option of no assignment) and still be compatible with the characteristic function.

This flexibility can be exploited to minimize the size of the s-graph, because the ASSIGN label a_v could in fact be any function which:

- is 1 whenever

$(S_{z_{i+1}\dots z_{m+l}}\chi^f_{z_i=1}) \wedge \overline{(S_{z_{i+1}\dots z_{m+l}}\chi^f_{z_i=0})}$
is 1 and

- is 0 whenever
$(S_{z_{i+1}\dots z_{m+l}}\chi^f_{z_i=1}) \vee \overline{(S_{z_{i+1}\dots z_{m+l}}\chi^f_{z_i=0})}$
is 0.

4.1.1 Arithmetic Function Handling

A transition function involving arithmetic operations cannot be handled efficiently by a direct application of procedure **build**. First of all, representing the function χ^f (e.g., as a BDD) would be very inefficient. Moreover, the cofactoring operations were described assuming binary variables. In practice, we need to handle the more general case of *extended* FSMs, with arithmetic and relational operators. So we must consider a *data flow* graph associated with both the BDD representation of χ^f and with the s-graph, which performs arithmetic computations. Formally, this can be described by representing the global CFSM transition function as a composition of a *reactive block* and a set of *side functions*. The reactive block specifies a function from a set of input variables (some of which can be outputs of some side functions) to a single *selection* variable. This variable selects which function of the primary inputs of the CFSM is assigned to each output. In practice, side functions will be implemented as arithmetic and logical expressions in the target language.

The selector can be viewed as a matrix of pointers to the functions $h_{(k,j)}$, where k is a value of the selection and j is the index of an output variable. Each $h_{(k,j)}(z_1, \dots z_{m+l})$ is a specific function that is used to compute the value of variable y_j for selection k.

The algorithm to build an s-graph from a transition function represented in composite form is derived from procedure **build** as follows. First, procedure **build** is applied to the reactive block R only, with just one multivalued output variable, the selection s. In the resulting s-graph each path has only one ASSIGN vertex labelled with $s \Leftarrow k$. Each ASSIGN vertex of this initial s-graph is replaced by a chain of l ASSIGN vertices each associated with y_j and $h_{(k,j)}$.

Figure 2 shows a data-flow graph of the CFSM of Figure 1. The control-flow graph is contained within the reactive block R.

The output function selection matrix $h_{(k,j)}$, defining the value of y_j given selection k, is shown below.

	a'	$*y$
$s = 1$	INC(z)	0
$s = 2$	z	1

4.1.2 Algorithm Implementation and Practical Issues

As we said above, the initial s-graph can be formed in a variety of ways, depending on the variable order used in procedure **build**. The following three major classes of orderings can be used:

1. Ordering each output *after* its support yields an s-graph where all the decision computation is done by TESTs. ASSIGN nodes are labeled only with the $h_{(k,j)}$ side functions.

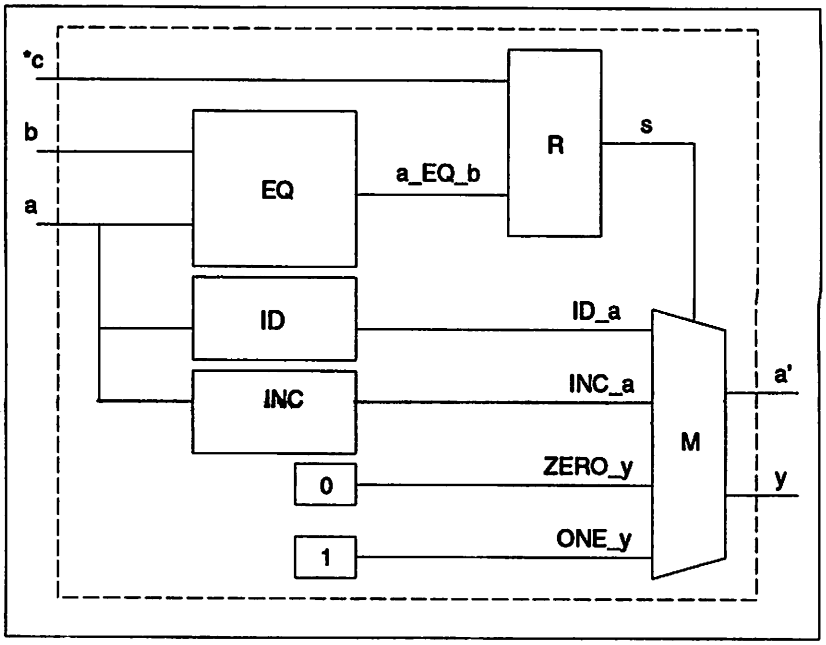

Figure 2: Block diagram of a mixed control- and data-flow graph

2. Ordering each output *before* its support yields an s-graph without TEST nodes. Each ASSIGN node is labeled with the logical *and* of the enabling condition for the multiplexer and the $h_{(k,j)}$ side function.

3. All other orderings yield an s-graph with some intermediate mix between TEST and ASSIGN nodes.

Our current implementation allows only the first two possibilities, while the range of solutions that can be obtained with the third class of orderings is available through an optimization step described in the next section.

In both cases, the s-graph is built from a BDD representing the characteristic function of the reactive component of the transition function. In case 1 a traversal of the BDD from the root corresponds exactly to the traversal performed by procedure **build**, because the smoothing of the output variables yields one of the two children of the current BDD node. The size of the s-graph is exactly the same as the size of the BDD.

For this reason, it is very important to have a small BDD representing the transition function. Dynamic variable reordering is run to locally optimize the BDD size. The reordering is done with the "sift" algorithm [Rud93], which sifts one variable at a time up and down in the ordering and freezes it in the position where the BDD size is minimized (with the constraint that no output can sift before any input in its support).

The s-graph obtained in this way has the following important properties:

1. each input variable is tested only once per path; this provides the minimum depth s-graph (which translates to minimum execution time), and

2. the ordering of the variable tests is heuristically optimal for code size, in the sense that no single variable can be moved in the ordering while decreasing the size of the BDD, and hence of the s-graph.

Case 2 is currently implemented by directly building a Boolean circuit implementing the reactive component of the transition function. The Boolean circuit is optimized using, e.g., the logic synthesis algorithms described in [SSL+92]. The s-graph can now be constructed as a string of ASSIGN vertices, one for each output variable y_j for each value k of the *selection* variable s. The function associated with each vertex v is
$a_v = y_j \vee ((s = k) \wedge h_{(k,j)})$[3].

The s-graph obtained in this way has the following important properties:

1. it contains no TEST vertices. Hence all its executions take exactly the same time, if we ignore the effects of the memory hierarchy and of different execution times for the same arithmetic operation with different input data. This property is very important for highly critical real-time systems where absolute exactness in execution time prediction is a key for safe operation.

2. the expression computing the selection variable is heuristically optimal for code size (and execution speed, due to the above mentioned property).

4.2 S-graph to C Translation

The s-graph obtained according to one of the procedures described in the previous section can now be translated into C code, to be compiled on the target machine. There are a number of optimization operations that can be performed on the s-graph before the final code generation. Some of them, such as code motion, common subexpression factoring, etc., are common to general-purpose compilers and will not be described in detail here. Other optimizations are specific to the FSM domain.

For example, we can collapse multiple adjacent TEST nodes with a unique entry point into a single TEST node with a multivalued function and multiple children. This complex TEST node can be implemented using constructs such as *switch* in C or *case* in Pascal. The choice of whether such collapsing should be used depends essentially on the *relative efficiency* of implementation of conditional jumps versus multiway jumps in the chosen micro-controller. For example, micro-controllers where the program counter is a general-purpose register allow efficient multiway jumps by using *jump tables*. The cost and performance estimation procedure can be used to quickly evaluate the alternatives and drive the optimization.

The final translation of the s-graph into C (or any other high-level language) is fairly straightforward, due to the direct correspondence between s-graph node types and basic C primitives. The fact that the code is so unstructured may hinder its readability[4], but allows greater efficiency.

Basically, a TEST node is translated to an *if* and two *gotos*, while an ASSIGN is translated to an assignment. Appropriate declarations of local and global variables, as well as Operating System

[3] Where logical *and* and *or* are performed bit-wise on multivalued variables, and predicate $s = k$ is extended to the appropriate number of bits.

[4] Note that in our envisioned codesign methodology the designer should never be exposed to it, just like the user of a general-purpose compiler should never have to look at the assembly code.

function	estimated			measured			perc. difference		
	timing		size	timing		size	timing		size
	min	max		min	max		min	max	
FRC	147	447	649	130	485	687	13.08	-7.84	-5.53
TIMER	212	775	1341	180	901	1433	17.78	-13.98	-6.42
ODOMETER	134	339	446	120	352	465	11.67	-3.69	-4.09
SPEEDOMETER	134	369	462	120	382	465	11.67	-3.40	-0.65
BELT	108	311	508	100	312	517	8.00	-0.32	-1.74
CROSSDISP	251	16883	37196	210	23160	39284	25.50	-27.10	-5.53

Table 1: Results of the cost/performance optimization procedure

support statements (e.g., handling of input and output event buffers) are also inserted into the output code.

After the synthesis of the code is complete, we can evaluate the effectiveness of the methodology, as shown in the next section.

5 Experimental Results

In this section we report the results of the cost/performance estimation procedure and of the s-graph synthesis procedure, applied to a car dashboard control system. In all cases, the numbers are given for a Motorola 68HC11 micro-controller. They are obtained using our estimation package, as well as by actual measurements done on the output of the *introl* C compiler for the 68HC11. The timing columns are given in terms of clock cycles for a *single transition* of each FSM, and the code size columns are given in terms of bytes.

Table 1 describes the output of the cost estimation procedure, and compares it against an exact measurement of the code size and timing, performed by analyzing the compiled object code. Function *CROSSDISP* has a significant discrepancy because it performs many arithmetic computations, which are poorly handled by the current version of the estimator (all operations are assumed to have the same size and timing as an *add*).

Table 2 shows the effect of the different orderings in procedure **build** on the software size and timing. In both cases, the computed function is exactly the same. The only difference is the order of the variables, which affects the number of TEST nodes. The first case uses a naive ordering, in which all outputs are ordered after all inputs, the second case forces each output to appear after its support (case 1 in Section 4.1.2). The timing in both cases is exactly the same, because the number of TEST and ASSIGN nodes on a path on the s-graph is the same. The only difference is the size, due to the sharing among subgraphs, which can be performed better in the second case.

We have also tried to compile the same code using the MIPS compiler, which has much better optimization capabilities than the *introl* compiler, and the results are similar. This demonstrates that the optimization space that we can explore in our restricted case is significantly larger than in the case of a general-purpose compiler.

function	in before out code size		support code size	
	estimated	exact	estimated	exact
FRC	723	783	649	687
TIMER	14752	16971	1341	1433
ODOMETER	408	410	446	465
SPEEDOMETER	457	465	462	465
BELT	854	876	508	517
CROSSDISP	38612	40768	37196	39284

Table 2: Effect of different TEST variable orderings

6 Conclusions and Future Work

In this paper we have presented a new methodology for the synthesis of software for embedded real-time control-dominated systems. The methodology exploits the use of a Finite State Machine specification, and unlike classical compilation algorithms starts from a description of the *function* to be computed, rather than from one operational implementation of it. This allows the use of powerful optimization algorithms based on Boolean function manipulation methods.

The internal representation that we use is also the basis of a quick but fairly precise cost and performance estimation procedure. The procedure is based on assigning cost parameters to the control/data-flow graph, and can be easily customized for different CPUs and runtime environments.

In the future we plan to exploit the cost estimation procedure to perform global optimizations aimed at satisfying timing and size constraints, with a much finer tuning than is currently possible. We are also exploring the coupling between scheduling algorithms and code synthesis, to allow the scheduling procedure to transmit user-defined constraints to the compilation steps.

References

[Ake78] S. B. Akers. Binary Decision Diagrams. In *IEEE Transactions on Computers*, volume C-27, pages 509–516, June 1978.

[ASU88] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques and Tools.* Addison-Wesley, 1988.

[BCG91] G. Berry, P. Couronné, and G. Gonthier. The synchronous approach to reactive and real-time systems. *IEEE Proceedings*, 79, September 1991.

[Bry86] R. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, August 1986.

[BS89] R. K. Brayton and F. Somenzi. Boolean Relations and the Incomplete Specification of Logic Networks. In *Proceedings of VLSI 89*, August 1989.

[CGH+93a] Massimiliano Chiodo, Paolo Giusto, Harry Hsieh, Attila Jurecska, Luciano Lavagno, and Alberto Sangiovanni-Vincentelli. A formal specification model for hardware/software codesign. Technical Report UCB/ERL M93/48, U.C. Berkeley, June 1993.

[CGH+93b] Massimiliano Chiodo, Paolo Giusto, Harry Hsieh, Attila Jurecska, Luciano Lavagno, and Alberto Sangiovanni-Vincentelli. A formal specification model for hardware/software codesign. In *Proceedings of the International Workshop on Hardware-Software Codesign*, 1993.

[CW91] R. Camposano and W. Wolf, editors. *High-level VLSI synthesis.* Kluwer Academic Publishers, 1991.

[CWB94] Pai Chou, E. Walkup, and G. Borriello. Scheduling for reactive real-time systems. *IEEE Micro*, August 1994.

[GJM94] R. K. Gupta, C. N. Coelho Jr., and G. De Micheli. Program implementation schemes for hardware-software systems. *IEEE Computer*, pages 48–55, January 1994.

[Hal93] N. Halbwachs. *Synchronous Programming of Reactive Systems.* Kluwer Academic Publishers, 1993.

[HLN+90] D. Har'el, H. Lachover, A. Naamad, A. Pnueli, et al. STATEMATE: a working environment for the development of complex reactive systems. *IEEE Transactions on Software Engineering*, 16(4), April 1990.

[KM92] D. Ku and G. De Micheli. *High level synthesis of ASICs under timing and synchronization constraints.* Kluwer Academic Publishers, 1992.

[Lee59] C. Y. Lee. Representation of switching functions by binary decision programs. *Bell System Technical Journal*, 38:985–999, 1959.

[LL73] C. Liu and J.W Layland. Scheduling algorithms for multiprogramming in a hard real-time environment. *Journal of the ACM*, 20(1):44–61, January 1973.

[Mei89] C. Meinel. *Modified branching programs and their computational power.* Springer-Verlag, 1989.

[Rud93] R. Rudell. Dynamic variable ordering for ordered binary decision diagrams. In *Proceedings International Conference on Computer-Aided Design*, November 1993.

[SSL+92] E. M. Sentovich, K. J. Singh, L. Lavagno, C. Moon, R. Murgai, A. Saldanha, H. Savoj, P. R. Stephan, R. K. Brayton, and A. Sangiovanni-Vincentelli. SIS: A system for sequential circuit synthesis. Technical Report UCB/ERL M92/41, U.C. Berkeley, May 1992.

[SSR89] R. Saracco, J.R.W. Smith, and R. Reed. *Telecommunications systems engineering using SDL.* North-Holland - Elsevier, 1989.

[WTHM92] W. Wolf, A. Takach, C.-Y. Huang, and R. Manno. The Princeton University behavioral synthesis system. In *Proceedings of the Design Automation Conference*, June 1992.

[Yee94] S.Y. Yee. An esterel to SHIFT compiler for a hardware/software codesign environment. Master's thesis, U.C. Berkeley, 1994.