Partitioning and Optimizing Controllers Synthesized from Hierarchical High-Level Descriptions

Andrew Seawright and Wolfgang Meyer
Synopsys, Inc. 700 E. Middlefield Road Mountain View, CA 94043

Abstract - *This paper describes methods for partitioning and optimizing controllers described by hierarchical high-level descriptions. The methods utilize the structure of the high-level description, provide flexible exploration of the trade-off between combinational logic and registers to reduce implementation cost, and allow the designer to control the synthesis process. Results are presented using industrial examples.*

1. Introduction

To tackle the exponential growth in the complexity of digital circuits, designers are moving to higher levels of abstraction in the design process. In control dominated applications, several abstractions are popular for managing the complexity of the design of sequential control logic. Examples include: hierarchical state machine description styles such as State Charts [Har87] and the many variants, the reactive programming language ESTEREL [Ber92], and other compositional description styles such as used in the PBS system [Sea94] and used in the Protocol Compiler design environment (DALI) [Sea96] [Mey97].

In these abstractions, the high-level controller description is typically described in a hierarchical fashion as depicted by the tree in Figure 1. The nodes of the tree represent the compositional operators of the control abstraction. For example, a particular node might represent the sequencing of the sub-behaviors or the concurrent execution of the sub-behaviors, etc.

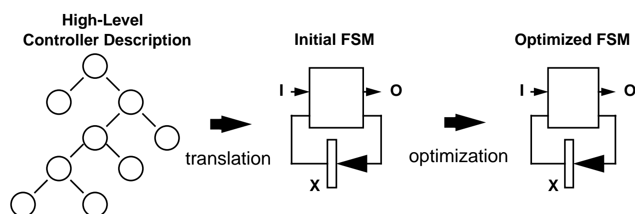

Figure 1. FSM synthesized from a high-level description

Synthesis from these abstractions involves the translation of the high-level controller description into an FSM. Typically, the synthesis is performed as an initial translation step followed by optimization steps. This is the case in the ESTEREL compilers and in Protocol Compiler. The translation ensures correct implementation of the high-level semantics into an FSM, and the subsequent optimizations aim to reduce the cost of the implementation while preserving the observable sequential behavior.

Separation of the translation and optimization phases can however lead to the loss of information about the high-level description that is useful for optimization.

This work presents methods for partitioning and optimizing controllers synthesized from high-level descriptions which 1) utilize the structure of the high-level description, 2) provide flexible exploration of the trade-off between sequential and combinational logic to reduce implementation cost, 3) and enable the designer to control the synthesis process.

This paper is organized as follows: Section 2 summarizes related work. Section 3 describes the overall compilation and partitioning process. Automatic partitioning is discussed in Section 4. Section 5 presents experimental results and conclusions are drawn in Section 6.

2. Relation to Prior Work

Most of the work involving the optimization of controllers operates on the circuit (or state graph) after translation from the high level description, and without any additional guidance from the structure of the high level description. The classical FSM optimization techniques -- state minimization, state assignment, state encoding -- and sequential circuit level optimizations such as retiming fall into this category. In the work presented here, the structure of the high-level description is central in guiding the partitioning and optimization of the controller.

Touati and Berry [Tou93] studied the optimization of ESTEREL generated controller circuits using specialized techniques for removing redundant registers using reachability don't cares. Specific optimizations such as replacing registers by combinational functions of other registers, detecting and exploiting net equivalences, and performing local retimings at the circuit level are described. Continuing in this area, Sentovich, Toma, and Berry [Sen96] [Sen97] presented additional methods for optimizing controller circuits generated from high-level descriptions such as ESTEREL. Single register, multiple register, and related re-encoding optimizations are described to improve the register / combinational logic ratio to reduce the cost of implementation or formal verification of the designs. These methods are simply suitable for the characteristics of the circuits generated by the high-level description translation process. The high-level representation is only used to calculate an over approximate reachable state set and is not central in the optimization process.

Crews and Brewer [Cre96] presented various techniques for optimizing protocol controllers generated from hierarchical regular expression descriptions. The input regular expression DAG (directed acyclic graph) is minimized before circuit translation resulting in a reduction of registers after circuit generation. Other techniques for removing registers during circuit translation are described. These techniques do in fact utilize the structure of the input description. However, since the information about the global reachability and observability is unknown, the optimizations are generally local in scope.

The above techniques do not consider partitioning the controller, based on the high level description, into multiple sub-machines and do not allow designer control of the optimization process. However, these techniques can and are combined with the approach presented here.

35th Design Automation Conference ®
Copyright ©1998 ACM
1-58113-049-x-98/0006/$3.50
DAC98 - 06/98 San Francisco, CA USA

3. Controller Compilation

Optimizing the controller by extracting a single state transition graph for the entire controller is generally only feasible for small controllers (less than a few hundred states). Optimizing the controller using techniques such as redundant register removal and local re-timing tends to result in optimizations which are local in scope (each transformation only re-encodes a small part of the logic). To avoid these limitations and allow architectural exploration of large controllers, it is advantageous to allow the optimization and re-encoding of "chunks" or *partitions* of the controller logic as illustrated in Figure 2.

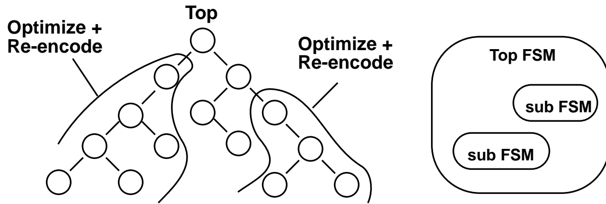

Figure 2. Optimizing and re-encoding portions of the controller related to portions of the high level description

To select the portions of the circuit to optimize, two methods are used: automatic and manual. In automatic partitioning, reachability information is used *in conjunction with* the high level description to automatically identify portions of the circuit to optimize. Manual partitioning allows the designer to select portions of the circuit to optimize by placing attributes on the nodes of the high level description.

In the compilation process, the relationship between the high-level description and the initial translated FSM is maintained. This allows the partitioning and optimization to utilize the high-level structure. Data structures are created to identify the logic and registers translated for each node of the high level description (Figure 3).

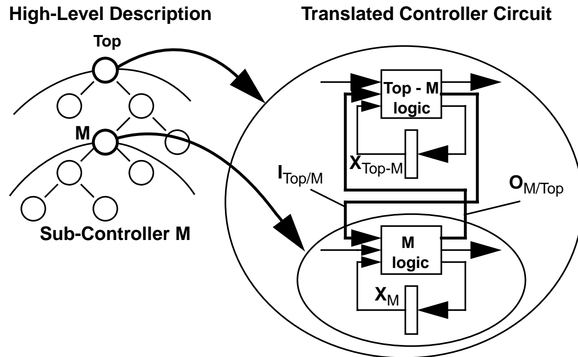

Figure 3. Relationship between the high-level description and the translated controller logic

Each partition sub-machine is optimized and re-encoded utilizing sequential don't care information. This optimization is performed by extracting and re-encoding the state transition graph of the sub-machine in the context of the overall controller. This is performed using a *transition pruning relation* to prune impossible state transitions of the sub-machine in the context of the overall controller.

Advantages of this transition pruning relation approach are that 1) the transition pruning relation can be constructed from either the implicitly computed reachability information *or* from certain designer specified properties of the nodes of the high level description (if the exact reachable states can't be computed) and 2) the avoidance of creating an explicit Cartesian product state transition graph of the sub-machine and the rest of the controller.

3.1 Overall Compilation Process

The overall compile process is illustrated in Figure 4. After the initial translation of the high-level description (`top_node`) into an initial circuit (`top`), a reachability analysis is performed to calculate the set of reachable states R(X) of the entire circuit using an implicit BDD-based approach [Cou89] [Tou90]. If the reachable states computation is too expensive, this phase is skipped. In this case manually specified partitions and designer specified property attributes are required.

```
 1.PartitionedCompile(description top_node,
                      circuit top) {
 2.    R(X) = AnalyzeController(top)
 3.    ChoosePartitions(top_node, top, R(X))
 4.    foreach partition p in top {
 5.      c = ExtractSubCircuit(top, p)
 6.      g = CreateStateGraph(c, R(X))
 7.      g'= MinimizeStateGraph(g)
 8.      EncodeStateGraph(g')
 9.      c'= StateGraphToCircuit(g')
 10.     MergeBackSubCircuit(top, p, c')
 11.   }
 12.}
```

Figure 4. Overall Compilation Process

Controller partitions are selected by the routine `ChoosePartitions()`. Automatic partition selection is described in detail later in Section 4. For manual partitioning, `ChoosePartitions()` picks the nodes designated by the designer.

Each controller partition is optimized by extracting the logic for the designated node as a sub-circuit. This sub-circuit is then converted into a state transition graph which is minimized and re-encoded. During this conversion, sequential don't care information is used to prune the state transition graph as described in the next section. The minimized and re-encoded state transition graph is converted back into a circuit and merged with the rest of the controller.

3.2 Sub-Controller State Graph Pruning

In order to effectively optimize a sub-controller, the communication between the sub-circuit and the rest of the controller must be carefully considered in the state graph generation process. In other words, the sequential don't care information should be utilized. Consider the diagram in Figure 3 which shows the logic for a sub-controller **M** in relation to the rest of the controller **Top**. The sub-circuit extracted for **M** comprises all the combinational logic (**M logic**) and registers ($\mathbf{X_M}$) for the sub-tree **M** in the high-level description. A portion of the inputs to **M** are internal wires from other parts of the controller and are not primary inputs. These inputs are designated $\mathbf{I_{Top/M}}$. In a similar way, a portion of the outputs of **M** are internal wires to the rest of the controller and are not primary outputs. These outputs are designated $\mathbf{O_{M/Top}}$.

Optimization of **M** must consider the sequential behavior of the inputs $\mathbf{I_{Top/M}}$ as these inputs are internally generated control signals and their signal values are typically strongly correlated with the sequential behavior of **M** itself. The sequential don't care information about $\mathbf{I_{Top/M}}$ should be considered in the optimization of **M**. If the don't care information is not considered, then the extracted state graph for **M** can be exponentially larger than necessary.

The state graph for a sub-controller is generated in a breadth first search process. Starting with the initial state of the sub-circuit, the next states and next state transitions are enumerated by forward image calculations of the present states through the next state logic [Cou89] [Tou90]. Each new state transition traversed is tested with respect to a *transition pruning relation* to determine if the state transition of the sub-circuit is consistent with the overall controller. States and transitions traversed consistent with the overall controller are allocated as the nodes and edges of the state transition graph. States and transitions inconsistent with the global controller are *pruned*. The pruning process is how the sequential don't cares of $\mathbf{I_{Top/M}}$ are incorporated in the optimization in an implicit way.

State graph pruning is illustrated by the example in Figure 4. The sub-controller **M** has one input **start** from the rest of the controller which signals when the sub-controller should start, and one output **done** to the rest of the controller which signals when **M** is done ($\mathbf{I_{Top/M}}$ = {**start**}, $\mathbf{O_{M/Top}}$ = {**done**}).

Assume **M** waits in its initial state until **start** is asserted and that **start** can only be true when **M** is in the initial state and can't be true again until after **done** has been asserted and M returns back to the initial state. The execution of **M** can't overlap with itself, it is *not pipelined*. A sub-machine that *can* be activated while it is in the process of execution is referred to as being *pipelined*. When the state graph for **M** is generated, the state $\mathbf{X_{M2}}$ is unreachable and is pruned because the transition $\mathbf{t_3}$ is taken from $\mathbf{X_{M1}}$ when **start** is true, which can't occur.

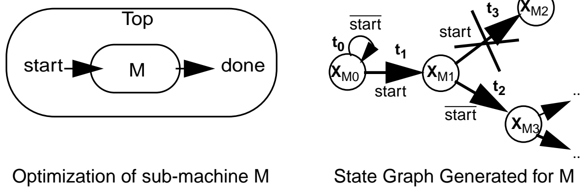

Figure 4. State Graph Pruning Example

Under the same assumptions about the behavior of the **start** signal, suppose the circuit extracted for sub-machine **M** is the simple shift register delay line illustrated in Figure 5. Without pruning, the number of states of the generated state graph would 2^n, where n is the number of registers in $\mathbf{X_M}$ (the size of the delay line). The state graph generation algorithm will assume that the shift register can fill with any binary value without using the sequential behavior of the **start** signal to prune the graph. However, the number of states in the pruned state graph is just n, as only one register in $\mathbf{X_M}$ can be set at a time based on the sequential behavior of the input **start**.

The above assumption about the behavior of the **start** signal is just a simple example to illustrate the pruning process. Under different assumptions about the sequential nature of **start**, the pruning can be more complicated.

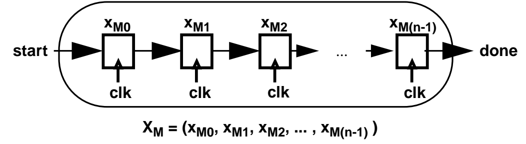

Figure 5. An Example Sub-Machine **M**

The transition pruning relation is a Boolean function $\mathbf{T : B^{n+m}} \rightarrow \mathbf{B}$. It returns true if and only if the current state and input condition specified leads to a valid next state. In this mapping, **n** is the number of the sub-machine state variables in $\mathbf{X_M}$ and **m** is the number of inputs $\mathbf{I_M}$ to the sub-controller, where $\mathbf{I_M} = \mathbf{I} \cup \mathbf{I_{Top/M}}$. **B** is the set {0, 1}.

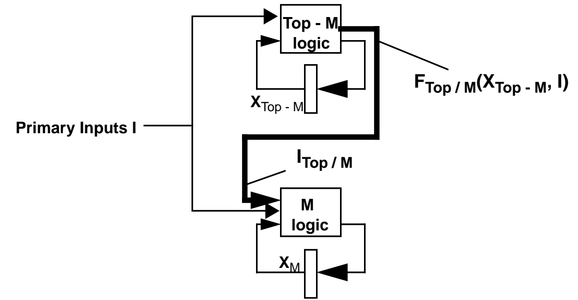

Figure 6. Functions $\mathbf{F_{Top/M}(X_{Top/M}, I)}$ drive sub-controller inputs $\mathbf{I_{Top/M}}$

The transition pruning relation $\mathbf{T(X, I_M)}$ of the overall controller is computed by setting the sub-machine internal inputs $\mathbf{I_{Top/M}}$ equivalent to the functions driving the inputs $\mathbf{F_{Top/M}(X_{Top\text{-}M}, I)}$ (Figure 6) and *anding* this result with the reachable states of the entire circuit R(X):

$$T(X, I_M) = (I_{Top/M} \Leftrightarrow F_{Top/M}(X_{Top\text{-}M}, I))\, R(X)$$

The transition pruning relation $T_M(X_M, I_M)$ for the *sub-machine* **M** is created by additionally quantifying out the state variables outside the sub-machine **M**:

$$T_M(X_M, I_M) = \exists(x_i\text{'s} \notin X_M)\, T(X, I_M)$$

For the example of Figure 4, the pruning relation $T_M(X_M, I_M)$ is computed as follows:

$$T_M(X_M, I_M) = \exists\big(x_i\text{'s} \notin X_M)\, (start \Leftrightarrow f_{start}(X_{Top\text{-}M}, I))\, R(X)$$

Table 1 describes how the transition pruning relation $T_M(X, I_M)$ is used to determine if the transitions of the example in Figure 4 are valid or invalid.

Table 1. Transitions of Pruning Example

Transition	Description	Reason
t_0	Valid	$T_M(X_M, I_M) X_{M0} \overline{start} \neq 0$
t_1	Valid	$T_M(X_M, I_M) X_{M1} start \neq 0$
t_2	Valid	$T_M(X_M, I_M) X_{M0} \overline{start} \neq 0$
t_3	Invalid	$T_M(X_M, I_M) X_{M1} start = 0$

If the exact reachable states information is not known, the transition pruning relations can be created by using 1) an over approximate R(X) calculated in a method similar to the approach in [Sen97] (beyond the scope of this paper), or by 2) using certain properties of the design set by the designer on the high level description. For an example of the latter case that is used in Protocol Compiler, consider the example of Figure 4. Suppose the designer -- based on knowledge of the design -- attaches a special property attribute on the node **M** of the high level description of the controller specifying that the sub-controller **M** is not pipelined. This property is then converted into a transition pruning relation $T_M(X_M, I_M)$ where **start** must be false in all states other than $\mathbf{X_{M0}}$. A suitable transition pruning relation can be directly constructed as follows:

$$T_M(X_M, I_M) = X_{M0} + \overline{X}_{M0}\overline{start}$$

3.3 Sub-Controller Optimization

After the state transition graphs are extracted for each sub-controller, the sub-controllers are optimized by minimizing and re-encoding their state graphs. During the state minimization process, the sub-controller outputs $\mathbf{O_{M/Top}}$ are treated as primary outputs so the state merging is done correctly as these outputs are observable by the other parts of the controller. After state minimization, state encoding methods are applied. For manual partitioning, the designer can select a particular style of encoding for each partition. For automatic partitioning, a minimum width binary code state encoding algorithm is used.

4. Automatic Partitioning

Section 3 described the overall compilation flow and how individual sub-controllers are extracted and optimized. This section describes an effective heuristic algorithm for automatically identifying the sub-controllers to extract and optimize.

The automatic partitioning approach uses the structure of the high-level description and information about the reachability of the controller.

The idea is to partition the controller, based on the high-level description, into the fewest sub-controllers each having the largest number of states below some *maximum* state threshold. This method is based on the observation that the re-encoding of state graphs is effective for state graphs below a certain size. Above this threshold, the combinational logic complexity for encoding and decoding the state generally outweighs the savings in register cost. The threshold should be set such that state graphs of this size or less -- when optimized and re-encoded -- will be implemented in efficiently.

The pseudocode for the automatic partitioning algorithm `AutoPart()` is shown in Figure 7. The first step of the algorithm `CollectData()`, collects data about the number of states contributed by each node of the high-level specification. This routine recursively traverses each node of the high-level description and calculates the number of states $\texttt{num_states}_n$ of the sub-controller `Mn` corresponding to the node `n` of the specification. The number of states for a node is computed by quantifying out the state variables of the parent node's reachable state set $R_p(X)$ passed down which are not state variables of node `n` (line 9).

The number of states for a node `n` is computed from the node's reachable state set by computing the number of minterms in $R_{Mn}(X_{Mn})$ (line 10). The algorithm recurses for each child node and passes down the node's reachable state set $R_{Mn}(X_{Mn})$ (line 12).

```
1.AutoPart(description top_node,
          circuit top, R(X))
2. {
3.   CollectData(top_node, R(X))
4.   PickPartitions(top_node)
5. }
6.
7. CollectData(node n, Rp(X))
8. {
9.   RMn(XMn) = ∃ (xi's ∉ XMn) Rp(X)
10.  num_statesn = num_minterms(RMn(XMn))
11.  foreach child node c of node n {
12.    CollectData(c, RMn(XMn))
13.  }
14.}
15.
16.PickPartitions(node n)
17.{
18.  if (num_states > MaxStates) {
19.    foreach child node c of node n {
20.    PickPartitions(c)
21.  } else {
22.    MarkPartition(c);
23.    return;
24.  }
25.}
```

Figure 7. Maximum State Threshold Algorithm

Concurrent behaviors which are independent or loosely interacting are generally selected as independent partitions as their concurrent execution leads to the Cartesian product of the state space of the individual concurrent behaviors. Thus, the partitioning algorithm tends to break the Cartesian products and each concurrent behavior is extracted and optimized separately.

5. Results

These techniques for partitioning and optimizing sub-controllers have been incorporated into Protocol Compiler [Sea96] [Mey97]. Controller partitioning was applied in the synthesis of two ATM controllers and 11 modules of a 70,000 gate commercial telecommunications ASIC (Tables 2-4).

Architecture exploration of the first ATM controller is shown in Table 1. The design is a receiver controller in an ATM switch. The controller extracts cell header information and performs routing lookup functions. The hierarchical high-level specification for this design comprised about 70 compositional constructs. Without changing the high-level controller description, seven different architectures were created using a variety of synthesis techniques for comparison. The architectures varied in the number of control state registers, from 135 registers in the initial translated circuit (first row), to 9 registers in the design implemented by optimizing it as a single large state transition graph (last row). In Protocol Compiler, the initial translated circuit is characterized by a large number of registers and with many small next state functions. This encoding style is referred to as a “distributed” encoding and is usually a fast implementation.

Table 2. ATM1 Architecture Exploration

architecture	time sec.	total state vars	# parts	final states	area	delay ns
initial translation (distributed)	2.2	135	--	751	1874	12.2
distributed + replace	14.3	120	--	751	1834	11.94
distributed + remove	82.1	117	--	751	2684	59.82
auto part @ 50	21.9	68	3	(31,49,1)	1964	15.28
manual part	29.4	21	4	(6,21,96,1)	1577	18.22
auto part @ 200	35.9	15	2	(31,96)	1719	18.40
single state graph	60	9	--	495	3795	30.97

The middle rows in the table compare architectures optimized by applying different implicit redundant register removal techniques and architectures optimized by different partitioning approaches. The implicit redundant register removal techniques (rows 2 and 3) are similar to the techniques described in [Tou93] and [Sen96] for removing redundant registers. The "distributed + replace" method removes redundant registers in the initial distributed encoding which can be replaced by wires to other registers. The "distributed + remove" method removes registers which can be replaced by combinational functions of other registers. The partitioned controller architectures were created by applying manual and automatic partitioning with different maximum state thresholds (50 and 200 states).

In the table, the times are for Protocol Compiler and are measured in CPU seconds on a Sun SparcStation 20 machine. The number of sub-controller partitions is shown (# part) along with the final number of states and transitions for each partition after state minimization for the partitioning examples. The final columns in the table show the gate level area and delay for the synthesized architectures after logic synthesis. The designs were mapped to the LSI 10k library.

The best design in terms of area was achieved by manually partitioning the design into four sub-controllers (row 5). In terms of delay, the best architecture is the "distributed + replace" design, where 15 registers were replaced by wires. It is important to point out that the single state graph design is the worst in area and the second worst in terms of delay. Optimizing this design as a single state graph minimized to 495 states and re-encoding resulted in very complex next state logic functions. The "distributed + remove" design architecture suffered, because the logic functions introduced to remove the additional 3 registers significantly impacted the area and critical path.

Results for architecture exploration of the second ATM controller design (ATM2) is shown in Table 3. ATM2 is a larger version of ATM1 which also performs operations and maintenance (OAM) and available bit-rate (ABR) control functions. This design is larger, its high-level description comprises over 270 compositional constructs. The number of states of the overall controller is 11,542, compiling the controller as a single state graph is not appropriate. In addition, manual partition of the design is tedious as the description is large. However, the auto partitioning approach can easily partition this controller and produced a far smaller design than the applicable non-partitioned approaches.

Table 4 shows the results of using Protocol Compiler to design 11 modules of a commercial telecommunications ASIC. The table compares the initial translated controller designs, optimized architectures selected manually by the designers, and optimized architectures created by applying automatic partitioning.

Table 3. ATM2 Architecture Exploration

architecture	time sec.	total state vars	# parts	final states	area	delay ns
initial translation (distributed)	24	414	--	11,542	3939	49.92
distributed + replace	647	402	--	11,542	3838	48.11
auto part @ 100	1213	55	9	(31,96,1,46, 46,46,46,50,49)	2813	22.76
auto part @ 200	1402	34	6	(31,96,1, 103,50,49)	3292	27.73

Partitioning was critical in achieving high quality of results in these designs. Initially, the automatic partitioning algorithm was not available to the designers at the time the ASIC implementations were synthesized. Thus, for each module, the designers studied a variety of architectures to achieve the best quality of results. The designers compared the final architectures selected for the ASIC implementation to similar or equivalent RTL descriptions to check the overall quality of results and they were satisfied that the Protocol Compiler implementations were comparable to what could be achieved by the conventional RTL methods.

For seven of the 11 modules, partitioning was superior to the other methods. In three other cases (d4, d7, and d10), optimizing the designs as a single state graph achieved the best results. The module (d11) was optimized by using the "distributed + replace" method. In the manual partitioning process, a full reachability analysis was not performed. Instead, the designers placed design property attributes on the high-level description to indicate how the sub-controllers interacted. From this information, transition pruning relations were created and used in the sub-controller optimizations.

After the automatic partitioning algorithm was implemented in Protocol Compiler, it was applied to all of these modules without changing the high-level controller description. The maximum state threshold was set to 200 states and a full reachablility analysis was performed during the automatic partitioning for use in optimizing the sub-controllers.

The automatic partitioning algorithm, in almost all cases, selected the same or very similar partitions to the designer's manual partitions. For the cases where the designers did not use manual partitioning, the automatic partitioning results were close. The difference in the CPU times between manual partitioning and automatic partitioning are due to the full reachability analysis performed only during the automatic partitioning.

6. Conclusions

This paper presented methods for partitioning and optimizing hierarchically described controllers. These methods exploited the hierarchical high-level description. Partitioning of the controller can be performed either manually or automatically. The results showed that the designer can explore a wide variety of architectures and demonstrated an effective algorithm for automatically partitioning these controllers.

References

[Ber92] G. Berry and G. Gonthier, "The ESTEREL synchronous programming language: design, semantics, implementation", in Science of Computer Programming, vol. 19 (no. 2), pp. 87-152, November 1992.

[Cou89] O. Coudert, C. Berthet, and J. C. Madre. “Verification of Synchronous Sequential Machines Based on Symbolic Execution”, in *Automatic Verification Methods for Finite State Systems, International Workshop,* Grenoble France, vol. 407, Lecture Notes in Computer Science, Springer-Verlag, June 1989.

[Cre96] A. Crews and F. Brewer. “Controller Optimization for Protocol Intensive Applications”, proceedings of Euro-DAC 96, Geneva, Switzerland, pp. 140-145, September 1996.

[Har87] D. Harel, “Statecharts: A Visual Approach to Complex Systems”, in Science of Computer Programming, vol. 8 (no.3), pp. 231-275, August 1987.

[Mey97] W. Meyer, A. Seawright, and F. Tada, “Design and Synthesis of Array Structured Telecommunication Processing Applications”, proceedings of the 34th Design Automation Conference, pp. 486-491, June 1997.

[Sea94] A. Seawright and F. Brewer. “Clairyoyant: A System for Production-Based Specifications”, in the proceedings of IEEE Transactions on VLSI Systems, vol. 2., (no. 2), pp. 172-185, June 1994.

[Sea96] A. Seawright, U. Holtmann, W. Meyer, B. Pangrle, R. Verbrugghe, and J. Buck. “A System for Compiling and Debugging Structured Data Processing Controllers”, proceedings of Euro-DAC 96, Geneva, Switzerland, pp. 86-91. September 1996.

[Sen96] E. Sentovich, H. Toma, G. Berry, “Latch Optimization in Circuits Generated from High-level Descriptions”, in the proceedings of the International Conference on Computer-Aided Design ICCAD96, pp. 428-435, November 1996.

[Sen97] E. Sentovich, H. Toma, G. Berry, “Efficient Latch Optimization Using Exclusive Sets”, proceedings of 34th Design Automation Conference, pp. 8-11, June 1997.

[Tou90] H. Touati et al. “Implicit State Enumeration of Finite State Machines using BDD’s” in the proceedings of International Conference on Computer-Aided Design ICCAD90, pp. 130-133, November 1990.

[Tou93] H. Touati and G. Berry, “Optimized Controller Synthesis Using Esterel”, proceedings of the International Workshop on Logic Synthesis IWLS93, Lake Tahoe, 1993.

Table 4. Telecommunications ASIC Results

design	initial translation				manually selected architecture				auto partition @ 200 (full reachablility)			
	time sec.	total state vars	states	area	time sec.	total state vars	final states	area	time sec.	total state vars	final states / trans	area
d1	3	62	414	1,722 (668+1054)	8	15	(18,11,29)	1,572 (833+739)	22	15	(18,11,29)	1,549 (824+725)
d2	31	453	7,430	4,560 (851+3709)	395	13	(59,59)	1,156 (523+633)	3,157	13	(59,59)	1,175 (542+633)
d3	16	344	3,482	4,343 (665+3678)	113	13	(59,59)	1,836 (633+1203)	1,884	13	(59,59)	1,812 (611+1201)
d4	8	217	125	3,186 (877+2309)	99	7	min_encoded 114	1,353 (514+839)	305	8	(115)	1,416 (570+846)
d5	8	240	11,316,504	2,832 (312+2520)	39	28	(8,58,58,58,58)	1,524 (484+1040)	233	28	(8,58,58,58,58)	1,539 (496+1043)
d6	2	44	4,906	1,694 (609+1085)	6	15	(7,17,6,5)	1,585 (703+882)	15	15	(7,17,6,5)	1,582 (700+882)
d7	11	261	124	2,321 (282+2039)	187	6	min_encoded 54	601 (319+282)	694	7	(54)	620 (331+289)
d8	40	567	13,456	6,549 (1422+5127)	491	13	(59,61)	2,032 (756+1276)	3,437	13	(59,61)	2,033 (763+1270)
d9	5	153	3,970	2,260 (609+1651)	22	13	(62,62)	1,159 (568+591)	111	13	(62,62)	1,150 (559+591)
d10	8	218	125	3,656 (1070+2586)	111	7	min_encoded 117	1,966 (857+1109)	292	8	(118)	1,933 (817+1116)
d11	3	63	52,225	1,330 (478+852)	244	62	distributed +replace 52,225	1,206 (443+763)	107	9	(2,2,17)	1,125 (661+464)
totals	135			34,453	1,715			15,990	10,257			15,934