A System for Compiling and Debugging Structured Data Processing Controllers

Andrew Seawright, Ulrich Holtmann, Wolfgang Meyer, Barry Pangrle, Rob Verbrugghe, Joseph Buck
Synopsys, Inc., 700 E. Middlefield Road, Mountain View, CA 94043
{andy,ulrich,wolfgang,pangrle,verb,jbuck}@synopsys.com

Abstract

This paper describes a system for designing and implementing controllers for structured data processing. A graphical input style describes the format of the data to be processed along with the necessary control actions. Advantages over FSM approaches include: 1) ease of design changes, 2) ease of debugging, and 3) a shorter design cycle.

1. Introduction

The 10x increase in available gates that has been occurring every six years for the past two decades has forced designers to take a higher level approach to IC design. Some of the resulting increase in design complexity has been handled by libraries consisting of more powerful components. In part, because of the availability of these higher-level modules, the control portion of the design is frequently the bottleneck in the design cycle. While the control might occupy only 15% of the final chip area, it is not uncommon for the control portion of the design to take upwards of 75% of the design and debug effort time. The lack of structure and hierarchy, in the way control circuitry is designed today, makes incremental design changes caused by specification changes or design flaws extremely difficult and time consuming to implement.

This paper describes a system, called Dali, that shortens the overall design cycle by reducing the time and effort required to generate and debug the control circuitry. The user specifies the control by using a graphical symbolic format that closely matches the high level design specification. The system automatically synthesizes the controlling finite state machine. This frees designers from having to think of the control in terms of an FSM and allows them to concentrate on the design at a much higher level closer to the specification.

The system described in this paper provides an environment where simulation results are mapped back onto the graphical input specification. This allows the designers to find errors at the specification level quicker than using conventional means. Since the user doesn't specify an FSM, the redesign cycle at the FSM level is eliminated. The user modifies the graphical specification and the system automatically generates the new controller. Three key benefits to this new approach over traditional FSM approaches are: 1) design changes at the specification level are easier to handle, 2) debugging the design is easier since simulation results are back-annotated onto the high level graphical input specification, and 3) the design cycle time is decreased.

The paper is organized as follows. The next section discusses related work. Section 3 describes how controllers are specified in Dali. Compilation and debugging are described in sections 4 and 5. Results are presented in section 6.

2. Relation to Previous Work

Dali builds on previous work by Seawright and Brewer on logic synthesis from grammatical productions [Sea94a], [Sea94b]. Differences and improvements include use of a more graphical approach, smooth integration with HDL simulation and RTL synthesis, and others described in this paper.

There are other related approaches to high-level specification and design of controllers; the most prominent of these are the Esterel language [Ber92] and Statecharts [Har87].

Esterel and Statecharts are designed to cope with the complexities of designing reactive systems, where many external events, as well as internally generated events, may arrive at once, with events being more urgent than others. These languages have powerful mechanisms for coping with the complexity of dealing with many simultaneous events.

In systems that process structured data or protocols, the complexity is in the structure of the data and its evolution over time, and not necessarily in the complexity of the transactions that occur in an instant. The Dali approach supports a description style that explicitly uses the structure of the data.

This approach has many features in common with Esterel, including modularity, hierarchy, explicit parallelism, a synchronous model of communication, and well-defined, deterministic semantics. Because of these similarities, compilation and implementation techniques developed for producing controllers from Esterel input, e.g. [Tou93], are closely related to those used in Dali to generate

EURO-DAC '96 with EURO-VHDL '96
0-89791-848-7/96 $4.00

efficient implementations that avoid combinatorial explosion.

Dali does not pretend to be a completely general-purpose tool, but instead exploits the structure of a specific problem domain to yield powerful, high-level design solutions, and provides a means of interfacing with design structures implemented with other paradigms (e.g. logic specified at register-transfer level). In this philosophy (as opposed to the alternative of designing a completely general-purpose high-level design language), this project has been influenced by the Ptolemy project [Buc94].

3. Design Specification

The goal of Dali is to aid in the design of systems that process structured data streams or handle structured control protocols. In such systems, the data to be processed is typically organized in "frames", "cells", or "packets". Each of these structures has a hierarchy of sub-structures such as "headers", "markers", or "fields". Examples include telecommunications protocols (such as ATM and SDH/ Sonet) and systems that process compressed digital data streams such as in MPEG. Dali focuses on the design of the controllers and/or timing generators for these systems.

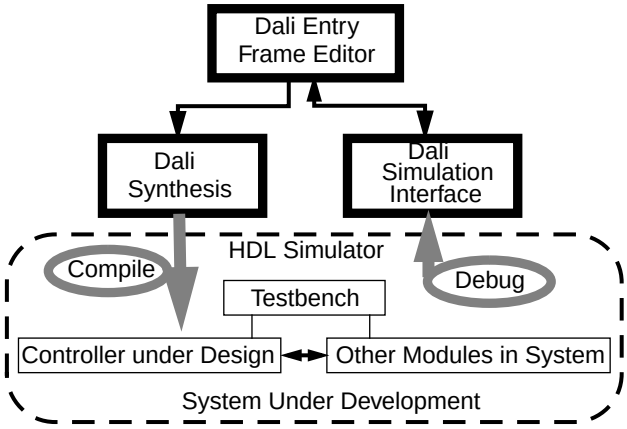

Figure 1. Dali

Dali is used to design controllers that are part of a larger system design (Figure 1). In Dali, a controller is specified graphically via a hierarchy of *frame* definitions and associated data operations called *actions*. The actions determine *what* to do and the frames describe *when* the actions are executed. Frames and actions will be described in greater detail below.

The graphical entry of the specification for the controller is performed using the Frame Editor. The controller specification is compiled by Dali and integrated with the other components of the system. The Dali simulation interface allows the system as a whole to be tested and debugged.

A simplified example is described here to demonstrate design specification in Dali. The example design is a receiver for the ATM protocol which processes a stream of incoming ATM cells. Based on the cell header information, the design stores the payload data into a particular area of RAM (Figure 2a). Figure 2b shows the ATM cell format. The input data arrives bytewise to the design via input `data_in`. The input signal `cell_start` indicates the starting byte of each arriving ATM cell of interest. During the following 53 clock cycles, the receiver processes the cell. First, the header information is extracted and then the payload data is written to the RAM at an address calculated from the VPI and VCI fields of the cell header.

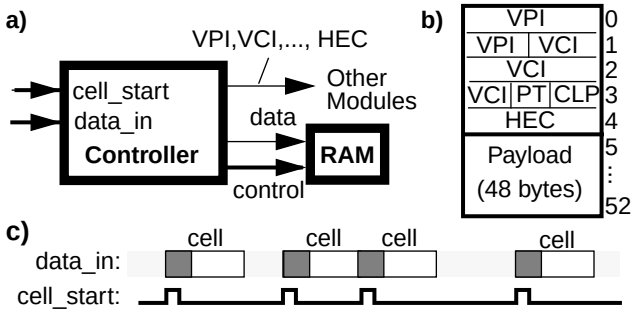

Figure 2. ATM Example: a) Block Diagram, b) ATM Cell, and c) Data Stream

Figure 3 shows the hierarchy of frames describing the example controller. In Dali, frames are definitions which describe cycle-level behaviors. Frames may be composed of simpler frames, in a similar manner to regular expressions, by means of sequencing, alternatives, repetition, and condition matching.

Frame `ATM_RX` forms the root of the hierarchy and consists of a composition of several frames. One of these frames is the frame `atm_cell`. The frame definition `atm_cell`, in turn, consists of the frames `header_extraction`, `header_processing`, and `payload`. This frame hierarchy is modeled closely to the input specification thus allowing an easier understanding, entry, and debugging of the design.

The `atm_cell` frame is defined as the *sequence* `header_extraction` followed by the *alternatives* `header_processing` and `payload`. All branches of an alternative are executed in parallel. So, after reading the header, header processing and payload processing are executed in parallel (frame `header_processing` is not expanded for brevity). *Alternatives* are also commonly used for parallel searches of different patterns.

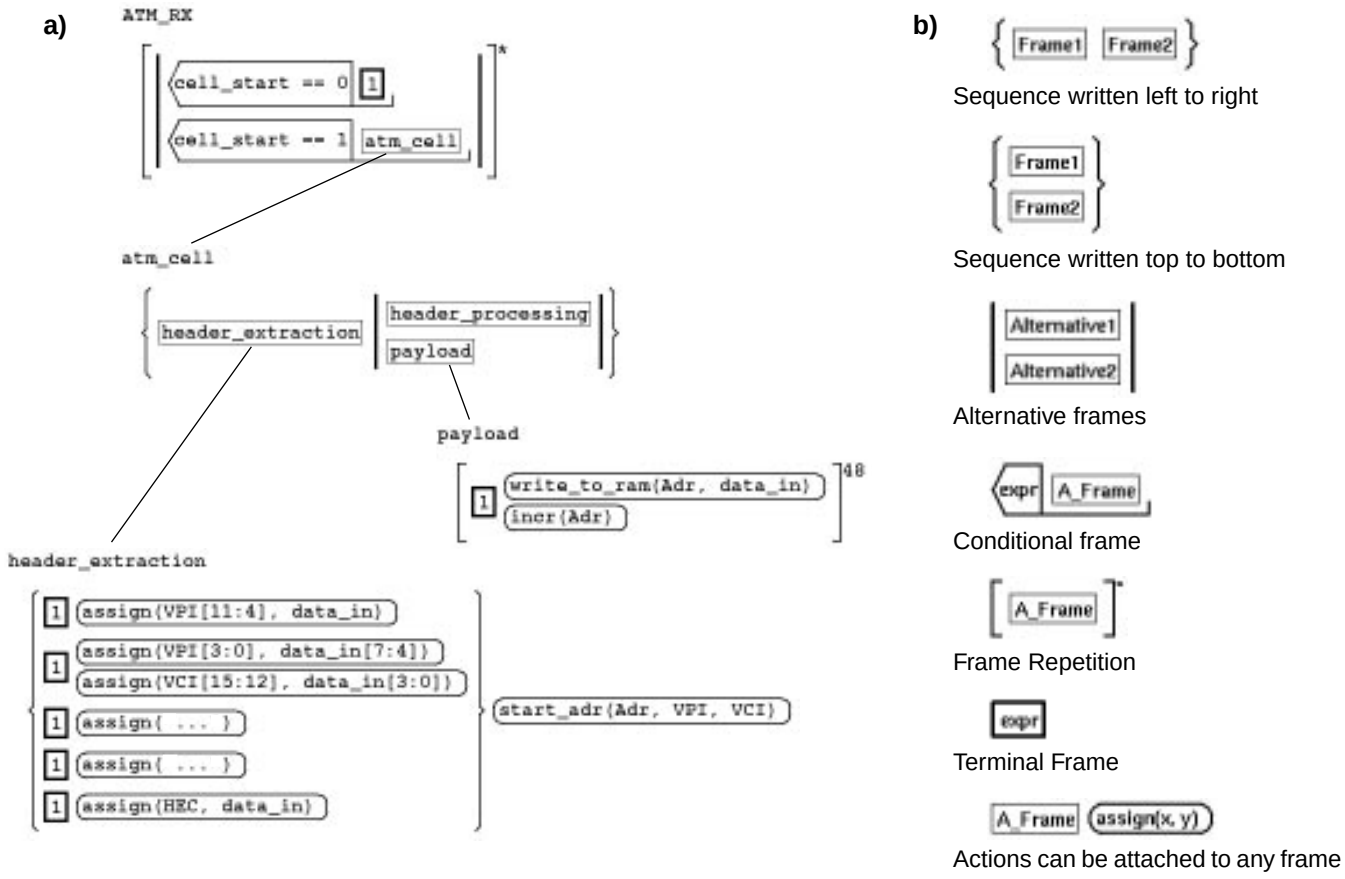

Figure 3. a) ATM Cell Example Frame Hierarchy b) Legend

The frame `header_extraction` reads the five bytes forming the header of the ATM cell and stores their values into the appropriate registers. This is performed by the sequence of five *terminal frames* and their attached actions. A *terminal frame* is executed for exactly one clock cycle when its associated Boolean expression is checked. If the expression is fulfilled, the terminal *accepts* and all attached actions are executed. The Boolean expression may be arbitrarily complex using input ports and internal variables.

Terminal frames are used to recognize specific input patterns as well as to describe a specific delay. In the case of the `header_extraction` frame, all five terminal frames have the condition "1" which is always fulfilled. Therefore they always execute in five consecutive clock cycles.

When the first terminal frame accepts, its attached action `assign(VPI[11:4], data_in)` is executed. This action simply assigns the second parameter to the first. According to the ATM format, the first byte of the header contains a part of the VPI field. During the next four clock cycles all of the other header fields are processed.

A frame *accepts* if the behaviors that it describes are satisfied. For a terminal frame, this means that the condition is fulfilled during one clock cycle. A sequence accepts together with its last frame. The execution of the sequence is aborted with the first non accepting frame. An alternative accepts together with the first accepting branch. A frame is said to be *active* if any of its sub-frames is accepting.

Actions can be associated with the frames at all levels of the hierarchy and they execute when the associated frame hierarchy accepts. Actions may be selected from a small set of *built-in actions* such as: `assign`, `clear`, `set`, `incr`, and `decr` or they may be *user-defined* allowing great flexibility.

The built-in actions are automatically translated directly into the "host" language (Verilog or VHDL) during code generation. A subroutine call is generated for each user-defined action. The designer specifies the body of the user defined actions in an action library file coded in the host language. Dali only needs to know the interface of the user-defined actions.

In this example, built-in actions are used in header extraction process and user-defined actions are used to write

received payload bytes into the RAM (`write_to_ram`) and to determine the starting address (`start_adr`).

Frame `payload` receives the 48 bytes of the payload data and writes it into the RAM. The frame is defined as a *repetition* of the inner terminal frame 48 times. As mentioned before, the combination of the actions and frames describes the desired behavior (Action part: write the received input bytes into RAM and increment the address. Frame part: for 48 clock cycles).

The continuous processing of the stream of cells is described by the *repetition* specified with the "*" superscript in the top-level frame. Recall that the beginning of a valid ATM cell is signaled by a pulse of the signal `cell_start`. After processing an ATM cell, the receiver must wait for the starting of the next cell. The top level frame describes a repeating operation of the alternatives: "waiting for the start of the next cell" and "processing a cell". We distinguish these two cases by the two alternative *conditional frames*. If the condition of a conditional frame is fulfilled, its inner frame is executed. For example, if `cell_start` is low (upper branch), then the terminal frame is executed and consumes one clock cycle. This causes this alternative to accept and the process starts over. If the condition of the lower conditional frame is fulfilled then the processing of the ATM cell starts. When `atm_cell` finally accepts (after 53 clock cycles) the process is started over again. In other words, the design will repeatedly: a) wait one clock cycle if `cell_start` is low, or b) process an ATM cell if `cell_start` is high.

4. Compilation

In a Dali design, frames and actions implicitly define a controlling FSM without the designer having to explicitly describe states or transitions. This is one of the major benefits of Dali. Although the designer does not have to handle the details of the underlying FSM, its is important to understand how the Dali FSM operates.

4.1 Controller Execution

The execution of the Dali described controller is cycle-based. At the start of each cycle, the external and internal inputs to the design are sampled by the controller and the current state of the controller is read. Next a control logic block is evaluated to determine which actions will occur in the cycle and also to compute the next state of the controller. In addition, to support debugging, the control logic block also computes a set of debugging signals that describe which frames are actively executing and which frames are accepting in the design. These debugging signals are used for back-annotating the state of the controller onto the input specification allowing debugging of the design at the high level.

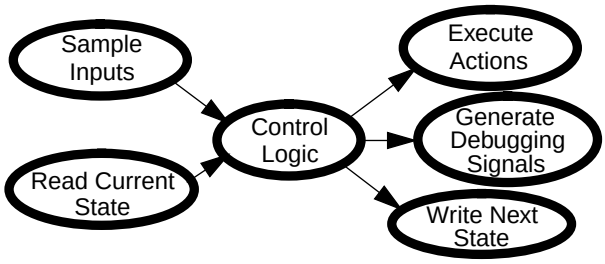

Figure 4. Controller Execution

4.2 Controller Synthesis

Figure 5 shows an overview of the synthesis steps. The input to synthesis is represented as a collection of tree structures representing the graphical operators and frame hierarchy. Each of the trees corresponds to one frame definition. The root of each tree describes the frame definition, each node of the tree describes one of the above operators, and the node's children are the operands of the operator. Leaves of the tree describe terminal frames or references to other frame definitions. The information about ports, variables and actions is also stored.

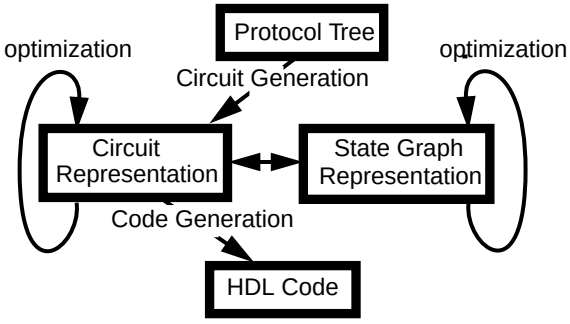

Figure 5. Controller Synthesis

The design is then checked for syntax errors including missing clock and reset signals, undefined frames, undeclared actions, and so forth. After this, the set of trees is elaborated into a fully instantiated single hierarchical *Protocol Tree*. Further checks, such as detecting recursive frame definitions, are also applied.

An initial circuit describing the controller as a set of Boolean next-state and output functions is created during elaboration. Starting with the simple formulas for the terminal frames, circuits for the parent nodes are composed based on the type of operation required until the root of the tree is reached as described in [Sea94a], [Sea94b]. Additional debug functions for each node of the tree are calculated in order to back-annotate active and accepting frames.

The circuit created for the root of the elaborated tree represents the complete controller. Inputs to the circuit are

the input ports, internal variables, and the state information. Outputs from the circuit are the action triggers, the next states, and debug functions.

Once an initial controller circuit is created it can be optimized in several ways. The initial circuit representation can be transformed into a state graph representation, or on the other hand, the circuit representation itself may be optimized. Based on compilation directives, Dali can perform a suite of optimizations on each of these two basic representations. This allows the selective application of optimization algorithms that are best suited for the two basic representations: circuit and state graph. Implicit optimization algorithms are performed on the circuit representation to analyze the circuit and remove redundant state registers [Ber90] [Tou93]. State graph algorithms are provided for state minimization and state encoding.

This allows, for example, designers to start with an initial circuit, remove redundant state variables and remove redundant states, convert it to a state-graph, choose an encoding that best fits their needs, convert it back into a circuit, and still have a circuit representation that is equivalent to the original one.

Following the above steps, we were able to optimize the *mismatch* example, a pathological regular expression, described in [Kar83]. On the circuit level, 6 state registers were found to be redundant, the state-graph representation was reduced to 1721 states from 8062 states in the original representation. The authors do not know of any other system that has been able to reduce this machine. [Sea94a] were not able to produce a state-graph representation for this circuit using SIS [Sen92]. It took 59 seconds on a Sun Sparc 20 to remove the redundant state registers and 486 seconds to produce the 1721 state machine.

4.3 Verification

To prove that the above optimizations and conversions were performed correctly on the examples, (i.e. produced equivalent circuits), a formal verification approach was undertaken. Given an initial controller circuit *C1* and an optimized version of the controller circuit *C2*, if the optimization steps do not change the behavior then *C1* must be equivalent to *C2*. To prove this, the product machine *C1xC2* is constructed and formally verified by showing that C1 and C2 produce the same output sequence for any input sequence. For example, the above minimized 1721 state machine was formally verified in 147 seconds on a Sun Sparc 20.

4.4 HDL Code Generation

HDL code generation is performed as a backend process. Dali generates either synthesizable Verilog or VHDL code, depending on the host language. The output code can also be generated with or without debugging information. Verilog code generation, for instance, produces a Verilog module describing the controller circuit. The module implements the behavior of the circuit including the logic of the controller and the triggers for actions. Note in Figure 6 that the actions are used to manipulate outputs and variables.

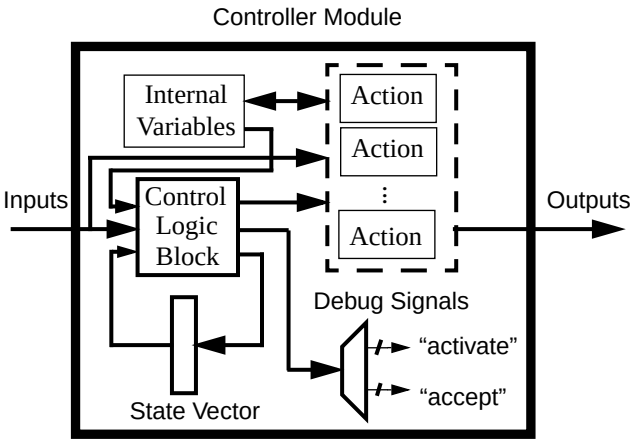

Figure 6. Block Diagram of Generated Code

5. Debugging

A testbench can instantiate the generated controller and other entities for simulation of the complete system. The Dali simulation interface provides a simulator independent mechanism for allowing Dali to interface to a variety of commercial HDL simulators for debugging. Dali debugging information for the controller is back-annotated in the Dali Frame Editor. The other signals in the design can be watched via an ordinary waveform viewer. In the Frame Editor, the designer sees which frames in the design are "accepting" and "active" in each cycle. The designer can also set breakpoints to stop the simulation when a frame is activated or when a frame accepts.

In a typical design flow, the design is first synthesized with the debugging information. The back-annotation to the specification during simulation makes debugging much easier so that design problems are uncovered earlier. Once the results are satisfactory, an optimized controller is compiled and the generated HDL code is synthesized using down stream synthesis tools.

6. Results

The Dali system has been used to build controllers for real life applications such as ATM and MPEG protocols. These are:

- A receiver for the ATM protocol. This design is simi-

lar to the example design shown before but performs more tasks during the header processing.

- A receiver for the MPEG system layer. Special actions are used to cope with cells of varying length.
- A receiver for the HDLC protocol. It accumulates incoming data bits into 32 bit words and performs CRC checking.

Table 1 shows the results. The example design is also included to allow a better interpretation of the values. The number of frames and actions allows one to judge the complexity of the protocol. Dali generated the FSM (see number of states and state bits) and the HDL output (all results are for VHDL). The shown times (running on a Sparc20) include FSM synthesis as well as HDL generation with all optimizations turned off. Short compilation times are important to allow a fast debugging cycle.

FSM synthesis time, however, can be much longer when optimizations are turned on which is relevant after finishing the debugging cycle. The table shows the results for removing redundant states.

The HDL output was then run through a commercial logic synthesizer, while distinguishing between the pure controller part and the gates required for actions. In the MPEG design, actions play an active role during frame recognition. Therefore, these actions are not separated from the controller part.

7. Summary and Conclusions

This paper has described Dali and its novel graphical input format that has been developed to closely match typical high level design specification diagrams for structured data processing controllers. This frees the designer to work at the level of the specification, instead of thinking of control in terms of an FSM. Three key benefits to this approach over the traditional FSM approach are: 1) design changes at the specification level are easier to handle, 2) debugging the design is easier since simulation results are back-annotated onto the high level graphical specification, and 3) it decreases the design cycle time.

Several examples have been shown in the paper to give the reader an idea of the scope and depth of the Dali system. The authors believe that Dali is the first system that has produced a state machine for the mismatch [Kar83] example that uses fewer than 8000 states (Dali used 1721 states). The other examples were chosen based on problems that are currently of commercial interest. Designer feedback indicates that the overall design cycle can be reduced by a factor of three and incorporating specification changes into a Dali design can take well less than half the time to implement compared to traditional design approaches.

References

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

[Ber90] C. Berthet, O. Coudert, J. C. Madre, “New Ideas on Symbolic Manipulations of Finite State Machines’’, in Proc. of ICCD’90, Cambridge MA, USA, September 1990.

[Buc94] J. T. Buck, S. Ha, E. A. Lee, and D. G. Messerschmitt, “Ptolemy: A Framework for Simulating and Prototyping Heterogeneous Systems,” in *International Journal in Computer Simulation*, vol. 4, no. 2, 1994, pp. 155-182.

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

[Kar83] A.R. Karlin, H.W. Trickey, and J.D. Ullman, “Experience with a regular expression compiler”, in Proc. of ICCD’83, pp. 656-665, 1983

[Sea94a] A. Seawright and F. Brewer, “Clairvoyant: A Synthesis System For Production-Based Specification,” in *IEEE Trans. on VLSI Systems*, June 1994, pp. 172-85.

[Sea94b] A. Seawright, “Grammar-Based Specification and Synthesis for Synchronous Digital Hardware Design,” Ph. D. Thesis, University of California at Santa Barbara, June 1994 (UMI order #9500298).

[Sen92] 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”, Electron. Res. Lab. Memo, No. UCB/ERL M92/41, May 1992

[Tou93] H. Touati and G. Berry, “Optimized Controller Synthesis Using Esterel”, in *Proc. International Workshop on Logic Synthesis IWLS’93*, Lake Tahoe, 1993.

Table 1: Design Results

Design	Input Specification		Generated FSM + HDL						Circuit Size	
	Frames	Actions	Time [s]	Bits	Time[s]	Redundant bits	states	lines VHDL	Ctrl [gates]	Actions [gates]
Example	26	14	0.6	10	1.0	1	10	191	197	403
ATM	12	17	0.9	20	1.6	3	19	602	183	609
MPEG	183	28	1.8	68	300	0	344	657	1416	
HDLC	17	10	0.9	21	1.6	6	38	216	403	1097