Formal Specification of an Asynchronous Viterbi Decoder

Johanna Tuominen*†, and Juha Plosila†
*Turku Centre for Computer Science, Finland
†Dept. of information Technology, University of Turku, Finland
{joeltu|juplos}@utu.fi

***Abstract*— Conventionally, the correctness of functional and non-functional properties of hardware components is ensured during design process by simulation. Moreover, different description languages are needed during development phases. Thus, by adopting the Action Systems, we are able to use the same formalism from specification down to implementation. Recently, we have been exploiting possibilities to formally model power consumption. That is the purpose is to develop a formal power estimation flow which can be used to monitor the power consumption from abstract level down to the gate level implementation. In this paper, we present a formal model for asynchronous Viterbi decoder, which will be used as a case study for the power estimation flow in the future.**

I. Introduction

Formal methods provides an environment to design, analyze, and verify digital hardware with the benefits of rigorous mathematical basis. In this study, the Action Systems formalism is applied [1]. It is a framework for specification and correctness preserving development of concurrent systems, and it is based on an extended version of Dijkstra's language of guarded commands [3]. Development of the action system is done in a stepwise manner within the $refinement\ calculus$ [2]. The specification of a hardware system is transformed into an implementation using correctness preserving transformations. In conventional Action Systems, only the logical correctness of the system is verified, while non-functional properties, like time, power, and area are not validated.

Convolutional encoding and Viterbi decoding are widely used in modern communication systems, such as digital satellite TV, and digital mobile radios [6]. To satisfy the demands caused by the developments of the modern telecommunication, high-speed, low-power, and low-cost Viterbi decoders are required. In this paper, we present a formal model of an asynchronous Viterbi decoder. The asynchronous approach is chosen for the implementation because of its potential for low-power, and low-noise behavior [9].

Currently, we are exploiting the possibilities to formally model power consumption [7] [8]. The purpose is to develop a formal power estimation flow from initial specification down to implementation. To estimate the power consumption, there is a trade-off between the accuracy and the abstraction level of detail which the system is analyzed. The more detailed the description, the more accurate the simulation will be. But on the other hand, the more time consuming it will be. Moreover, the designer wants to make decisions as early as possible in the design flow to avoid design backtracking. Thus, the purpose is to use the asynchronous Viterbi decoder as a case study for the power estimation flow. That is, to estimate the power consumption of the decoder at different development phases. For instance, starting from the formal description presented here, and finally from the gate-level description.

II. Action Systems

An action A is defined by (for example):

$$
\begin{array}{ll}
A ::= abort & (abortion, non-termination) \\
\quad | \ skip & (empty\ statement) \\
\quad | \ A_1 [\!] \ ... \ [\!] \ A_n & (non-deterministic\ choice) \\
\quad | \ A_1; \ ... \ ; A_n & (sequential\ composition) \\
\quad | \ x := e & ((multiple)\ assignement) \\
\quad | \ g \ \rightarrow \ A & (guarded\ command)
\end{array}
$$

where A_i, $i = 0, ..., n$, are actions; x is a variable or a list of variables; x_0 is a value(s) of the variable(s); e is an expression or a list of expressions; g is a predicate.

Semantics of actions. Action is considered to be $atomic$, which means that only the initial and final states are observed by the system. Thus, when selected for execution, the action is completed without any interference from other actions. Atomic actions may be represented by simple assignments or by more complex action compositions, such as the atomic sequence. $Non-atomicity$ means that an action outside the composition can execute between two component actions of the construct, which is not possible in the $atomic$ composition structures. The notation differs whether the composition is atomic or not, for instance, the sequential composition is noted by ; ($atomic$), and **;** ($non-atomic$).

The actions are defined using weakest precondition for predicate transformers [3]. For instance, the correctness of an action A with respect to predicates P and Q (precondition and postcondition) is denoted by: $\{P\}A\{Q\} = P \Rightarrow wp(A, Q)$. The $wp(A, Q)$ is the weakest precondition for the action A to establish the post-condition Q. The $guard\ gA$ of an action A is defined by $gA = \neg wp(A, false)$. An action is enabled when its guard evaluates to $true$, otherwise disabled.

A. Action System

An action system $\mathcal{A}$ has a form:

sys $\mathcal{A}\ (g)\ [par]$
|[
type t
const c
var v
actions A
subsys S_A
init "$initialization\ of\ the\ variables\ g\ and\ v$"
exec
do "$composition\ of\ actions\ A$" **od**
]|

Three different parts can be identified from the action system description: $interface$, $declarations$, and $iteration$.

The interface part specifies global variables g, that is, variables that are visible outside the action system. In other words, global variables are accessible by other action systems. If an action system does not have any interface variables, it is a $closed$ action system otherwise it is an $open$ action system. The declaration part consists of type (t), variable (v), constant (c), and action (A) declarations. Furthermore, type definitions and initializations are described in the declaration

part. Using the items introduced in the interface and declarative parts the operation of the system is described in the iteration section; in the **do** $-$ **od** loop.

The operation of an action system is started by initialization in which the variables are set to predefined values. Actions are selected for execution based on the composition operators and the enabledness of the actions. The operation is continued until there are no actions to enable, which temporarily aborts the system. Thus, the operation continues if some action enables it.

Quantified constructs Any action-level operator $\bullet \in [\!], ; (atomic), \mathbf{;}(non-atomic)$, and the system-level operator $[\!]$ can be quantified using the notation defined as follows:

$$[\bullet\ 1\ \leq\ i \leq\ n\ :\ A(i)] \widehat{=} A(1)\ \bullet\ ...\ \bullet A(n)$$
$$[[\!]\ 1\ \leq\ i \leq\ n\ :\ \mathcal{A}(i)] \widehat{=} \mathcal{A}(1)\ [\!]\ ...\ [\!]\ \mathcal{A}(n)$$

Composing Action Systems Consider two hierarchical action systems $\mathcal{A}$ and $\mathcal{B}$ with distinct local variables, local procedures, subsystem instances, and actions. The parallel composition of such systems is denoted by $\mathcal{A}\ [\!]\ \mathcal{B}$. It is defined to be another action system whose global and local identifiers (procedures, variables, subsystem instances, actions) consist of the identifiers of the component systems and whose **exec**-clause has the form: *do* $A\ [\!]\ B$ *od* $[\!]\ S_A\ [\!]\ S_B$. Here A and B denote the action compositions, and S_A and S_B the subsystem compositions in $\mathcal{A}$ and $\mathcal{B}$, respectively. The definition of the parallel composition is used inversely in system derivation to decompose a system description into a composition of smaller separate systems or internal subsystems.

III. Asynchronous Viterbi Decoder Design

A. Viterbi algorithm

Viterbi decoders [10][11] are used to decode convolutionally encoded data. A message encoded using a convolutional encoder follows a trellis diagram, which shows the different states of the encoder as well as the path taken to encode an arbitrary message. Despite of the possible errors in the stream, the Viterbi algorithm tries to reconstruct this correct path based on the received stream. This is accomplished by reconstructing the trellis diagram and allocating a weight to each branch and node of the reconstructed trellis per each time slot. These weight defines the likely branches and nodes used by the encoder. By tracing back through the reconstructed trellis, the decoder can detect and correct errors in the receiver stream. In other words the Viterbi algorithm finds the sequence of symbols in the given trellis that is closest in distance to the received sequence of noisy symbols, which is the global most likely sequence. By adopting the $Euclidean$ distance as a distance measure, the algorithm is the optimal maximum likelihood detection method, when the sequence of received symbols is corrupted by the additive white Gaussian noise (AWGN) [4].

B. Asynchronous Viterbi architecture

The proposed decoder is a soft-decision 64-state $1/2$- rate Viterbi decoder. The generator polynomials used are the industrial standards $(171_8, 133_8)$ [6]. A simplified block diagram of the decoder is shown in Figure 1.

The decoder consist of three units: Branch Metric Unit (BMU), Path Metric Unit (PMU), and the Trace Back Unit (TBU). The BMU generates the branch metrics, which measure the difference between the received symbol, and the symbol that causes the transition in trellis. Path Metric Unit (PMU) consists of two parts: The Add-Compare-Select Unit (ACSU) and the State Metric Memory (SMM). To find the survivor path for each state, the branch metric of a given

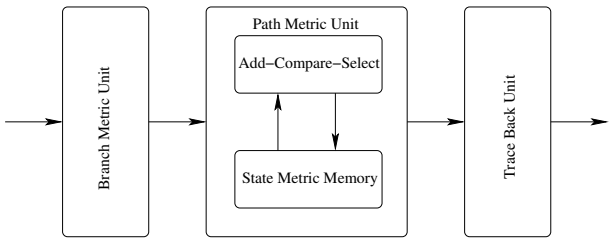

Fig. 1. A simplified block diagram of the Viterbi decoder

transition is added to the recent path metric value stored in the state metric memory. This new path metric is then compared with other path metrics, which are entering to that state. The transition with the minimum path metric is chosen to be the survivor metric. The path metric of the survivor path of each state is updated and stored back into the state metric memory. Trace Back Unit (TBU) stores the survivor paths and performs the trace back operation. Finally, it outputs the decoded sequence.

IV. Formal Specification

The formal description of the Viterbi decoder is modeled as a hierarchical Action System. Thus, the top level definition is fairly simple. It consists of control variables, and three subsystems: Branch Metric Unit (BMU), Path Metric Unit (PMU), and Trace Back Unit (TBU). The decoder is enabled when there is data available from the encoder by setting the $enable_{decoder}$ variable to $true$. Then the BMU is enabled $(V1)$. On the contrary, when the there is no data to process the decoder is disabled by setting the $enable_{decoder}$ to $false$, and the BMU is disabled $(V2)$.

sys *ViterbiDecoder* $(enable_{decoder}, enable_{bmu} : bool)$
$[\!]$
const $states := 64;$
$depth := 2;$
$L := 30;$
$mem_{depth} = 2 * (L + 1)$
$D_{max} = 4;$
type $bit : bool;$
$array : bit[0..states - 1][0..D_{max}];$
$bvect : bit[0..1];$
subsys $BMU, PMU, TBU;$
init $enable_{decoder}, enable_{PMU} := F;$
actions $V1 : enable_{decoder} \rightarrow enable_{bmu} := T;$
$V2 : \neg enable_{decoder} \rightarrow enable_{bmu} := F;$
exec
do $V1\ [\!]\ V2$ **od** $[\!]\ BMU\ [\!]\ PMU\ [\!]\ TBU$
$]\!]$

For simplicity, most of the constant variables are defined in the top level description. Moreover, we define a type bit, which is of type $boolean$. The value $true$ indicates the logic '1', and the value $false$ indicates the logic '0'.

The branch Metric is the squared distance between the received noisy symbol Y_n, and the ideal noiseless symbol of that transition $C_{i,j}$. That is, the branch metric from state i to state j at stage n is (Euclidean distance): $B_{i,j,n} = (Y_n - C_{i,j})^2$. Moreover, a multi-bit quantization is assumed for the input bits, all though not described here. The next state table, shown in Table I, of the trellis for the (2,1,7) convolutional encoder is modeled as of type $array$, and the BMU receives the table as a parameter ($metrics$). The system model for the BMU is defined by:

TABLE I
LOOK-UP TABLE OF THE OUTPUTS FOR THE PROPOSED DECODER

Current State	Output (input '0')	Output (input '1')
S0	00	11
S1	11	00
S2	01	10
S3	10	01
—	—	—
S62	11	00
S63	00	11

sys $BMU\ (din : bvect, bm0, bm1 : array, enable_{pmu} : bool)[metrics]$
|[
var $bm_{ready} : bool;$
proc $dist_{calc}(metrics[i,j], din) :$
$(bm0[l,(l,0..states-1] :=$
$(din - metrics[0,i,(i,0..states-1)])^2;$
$bm1[l,(l,0..states-1] :=$
$(din - metrics[1,j,(j,0..states-1)])^2);$
init $enable_{pmu}, bm_{ready} := F$
actions $B1 : enable_{bmu} \wedge \neg bm_{ready} \rightarrow dist_{calc};$
$bm_{ready}, enable_{pmu} := T;$
$B2 : pmu_{ready} \rightarrow bm_{ready} := F;$
$B3 : \neg enable_{bmu} \rightarrow enable_{pmu} := F;$
exec
do $B1 \parallel B2 \parallel B3$ **od**
]|

The incoming symbols din from the encoder are defined as of type bit vector $bvect$. When both the $enable_{bmu}$ is set to $true$, and the bm_{ready} is set to $false$, the calculation of the Euclidean distance is carried out by the procedure $dist_{calc}$ $(B1)$. Moreover, the variables bm_{ready}, and the $enable_{pmu}$ is set to $true$. This indicates that the BMU has processed the data, and the PMU is enabled. After the PMU is ready to accept new data, it sets the pmu_{ready} signal to $true$. Then the bm_{ready} is set to $false$, which indicates that the BMU is ready to accept new symbol from the encoder $(B2)$. The action $(B3)$ disables the PMU when there is no data to process.

Path metric unit is defined by:

sys $PMU\ (enable_{pmu}, pmu_{ready}, enable_{tbu})$
|[
type $smem : bit[0,...,states-1][0,...,2D_{max}+1];$
var $SMM := smem;$
$enable_{acsu} := bool;$
proc $update(SMM) : SMM[i,(i,0..states-1)] :=$
$pm_{out}[i,(i,0..states-1)];$
subsys $ACSU[i];$
init $enable_{acsu}, pmu_{ready}, enable_{tbu} := F;$
actions $P1 : enable_{pmu} \wedge \neg pmu_{ready} \rightarrow$
$enable_{acsu} := T;$
$P2 : \neg enable_{pmu} \rightarrow enable_{acsu} := F;$
$enable_{tbu} := F;$
$P3 : acsu_{ready} \rightarrow update; pmu_{ready} := T;$
$P4 : pmu_{ready} \rightarrow acsu_{ready}, pmu_{ready} := F;$
$enable_{tbu} := T$
exec
do $P1 \parallel P2 \parallel P3 \parallel P4$ **od**$\parallel [\ \parallel 0 \leq i \leq states-1 : ACSU[i]]$
]|

The PMU consists of the state metric memory SMM, control logic, and the 64 Add-Compare-Select Units (ACSU). The state metric memory is modeled as of type $smem$, which is an array. It stores the local winner for each state, which is used in the path metric calculation for the next calculation cycle. The size of each cell is defined by $2D_{max}+1$, where the D_{max} is the maximum possible difference in the path metrics. For instance, by assuming that the length of the each path metric is four ($D_{max} = 4$), then the size of the each cell in the given vector variable will be 9 bits. Thus, by adopting this approach the extra calculation needed for the normalization operation is avoided [5].

The ACS units are connected as a butterfly network, illustrated in Figure 2.

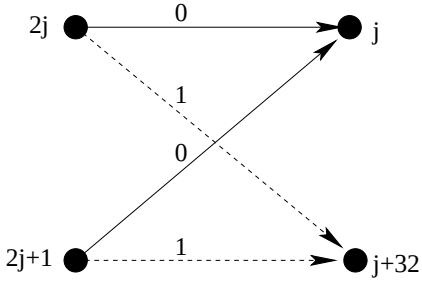

Fig. 2. State relationship for the path metrics computation.

Since the ACS units have a similar structure, it is modeled as a single subsystem $ACSU$. The 64 ACS units are generated from that model by using the quantified composition. The ACS units are indexed by the state number as follows: the first ACS unit is $ACS(0)$, the second one is $ACS(1)$ and so on. The PMU is enabled whenever there is valid data from the BMU to process, and the PMU is ready to accept new data $(P1)$. Then, the PMU enables the ACS units. The ACS unit is defined by:

sys $ACSU\ (acsu_{ready} : bool, pm_{out} : smem, D : bit)[i, SMM]$
|[
var $pm_{new1}, pm_{new0} : smem;$
$dv, select, acsu_{ready} : bool;$
proc $dec_{bit}(pm_{new1}, pm_{new0}) : (pm_{new1} < pm_{new0} \rightarrow D := 1;$
$pm_{new0} < pm_{new1} \rightarrow D := 0);$
proc $min(pm_{new1}, pm_{new0}) :$
$(pm_{new1} < pm_{new0} \rightarrow pm_{out} := pm_{new1};$
$pm_{new0} < pm_{new1} \rightarrow pm_{out} := pm_{new0})$
init $acsu_{ready}, select, dv := F;$
actions $A : enable_{acsu} \rightarrow pm_{new1}, pm_{new0} :=$
$(bm1[i,(i,0..states-1)]+$
$SMM[j,(j,(j,0..states-1]),$
$(bm0[i,(i,0..states-1)]+$
$SMM[j+32,(j,(j,0..states-1]);$
$dv := T;$
$C : dv \wedge \neg acsu_{ready} \rightarrow D := dec_{bit}(pm_{new1}, pm_{new0});$
$select := T;$
$S : dv \wedge select \rightarrow pm_{out} := min(pm_{new1}, pm_{new0});$
$acs_{ready}, select := T, F;$
$U : \neg enable_{acsu} \rightarrow dv, acsu_{ready}, select := F;$
exec
do $A \parallel C \parallel S \parallel U$ **od**
]|

The ACS unit consists of two adders, which calculates the sum of the incoming branch metrics, and the previous path metrics from the state metric memory SMM (A). The comparison operation finds the most likely path that has the minimum metric. Thus, the decision bit D is calculated by the action (C) using procedure dec_{bit}, and the smallest path metric is chosen by the selector (S), which is then stored into the state metric memory. By assuming that the total transition in the trellis is M, and the number of states is N, a maximum of $(M-N)$ comparison operations are required, and $2N$ sums are required to initialize the metric for each state.

Trace back unit stores the survivor paths for each state, and performs the trace back operation. Moreover, it outputs the decoded bit. The data structure of the traceback memory is shown in Table II.

TABLE II
TRACEBACK UNIT DATA STRUCTURE PER STAGE

State	Path Metrics	Decision bit
0	PM0	D0
1	PM1	D1
—	—	—
62	PM62	D62
63	PM63	D63

At time t, the path metric, which is the local winner, and the corresponding decision bit in each state is stored into the path metrics, and decision bit category, respectively. Then, the same procedure is repeated at time $t+1$. Thus, the state numbers are used as pointers, that is 0 corresponds to state $S0$, and so on. For simplicity, we use integer representation for them, however in the lower level implementations, the length of the pointers will be $(log_2(64) = 6)$ 6 bits. The minimum value for the length of the survivor path is $L = 5(log_2(N)$, where the N is the number of states [5]. In other words, the number of stages to store in the memory before the trace back operation can begin is $L+1$. In this case we define that, the number of stages that has to be stored before trace back is 31, and the overall length L for the trace back memory is 62. Thus, we can carry out the traceback from the memory location 30 down to 0, and at the same time write new data to memory locations from 61 down to 31. The trace back unit is defined by:

sys *TBU* $(decbit : bit)$
$|[$
type $tracemem : [0..states-3][0..2D_{max}+1];$
$decisionmem : [0..states-3];$
var $tr_{start}, inc, traceback : bool;$
$count : integer$
proc $gmin(trmem[k(k, 0..states-1), tr_{index}]) :$
$PM_{min} := trmem[0, tr_{index}];$
proc $trmem[k(k, 0..states-1), tr_{index}] \leq PM_{min} \rightarrow$
$PM_{min} := trmem[k, tr_{index}]; cstate := PM_{min}(k);$
subsys $TBU_{control};$
init $trmem := tracemem; decmem := decisionmem;$
$tr_{start} := F$
$count := 0, inc := F;$
actions $T1 : enable_{TBU} \rightarrow$
$trmem[i, count, (i, 0..states-1)] :=$
$PM_{out}[i, (i, 0..states-1)];$
$decmem[i, count, (i, 0..states-1)] :=$
$D[i, (i, 0..states-1)]; inc := T;$
$T2 : traceback \rightarrow$
$gmin(trmem[k, tr_{index}, (k, 0..states-1)]);$
$tr_{start} := T;$
$T3 : traceback \wedge tr_{start} \rightarrow$
$dec_{bit} := decmem[cstate, tr_{index}];$
$bit_{out} := decbit; tr_{index} := tr_{index} - 1; tr_{start} := F$
exec
do $T1 \; [] \; T2 \; [] \; T3$ **od**$[] \; TBU_{control}$
$]|$

The $TBU_{control}$ is a subsystem that controls the memory operations.

sys $TBU_{control}$
$|[$
actions $C1 : enable_{tbu} \wedge inc \rightarrow count := count + 1; inc := F;$
$C2 : count = L + 1 \rightarrow tr_{index}, traceback := L + 1, T;$
$C3 : count = 2 * (L + 1) \rightarrow$
$tr_{index}, traceback := 2 * (L + 1), T; count := 0;$
$C4 : tr_{index} := 0 \vee trace_{index} := L \rightarrow traceback := F;$
exec
do $C1 \; [] \; C2 \; [] \; C3 \; []C4$ **od**
$]|$

The TBU is enabled by the PMU when there is data to be stored. From each state, the TBU stores the smallest metric (local winner), and the corresponding decision bit $(T1)$. The $TBU_{control}$ is enabled to count the number of stages stored in the memory $(C1)$. When the number of stages reaches the survivor depth, that is 31 stages, the $TBU_{control}$ enables the trace back operation $(C2)$. The trace back is carried out by calculating the global winner from stages 30 down to 0, or (61 down to 31). That is the smallest metric from the local winners per each stage. This is carried out in the procedure $gmin$. The procedure returns the pointer to the global winner, that is the state number $cstate$. Then the decision bit corresponding the global winner state is read from the decision memory $decmem$, and then outputted as a decoded bit $(T3)$. This is carried out as long as the tr_{index} is either 0 or L $(C4)$. In parallel with the trace back operation the incoming path metrics are written into the memory locations from 31 to 61. Thus, the trace back is carried out alternately with the memory write operation, and therefore the TBU is enabled as long as the PMU is enabled.

V. CONCLUSIONS AND FUTURE WORK

In this paper, we presented a formal model for asynchronous Viterbi decoder. The decoder is 64-state $1/2$- rate Viterbi decoder, and the generator polynomials used are the industrial standards $(171_8, 133_8)$. The asynchronous implementation is chosen due to its potential for low-power, and low-noise behavior. The formal model presented will be used as a case study for the formal power estimation model, which will be included into the Action system formalism in the near future. The purpose is to analyze power consumption starting from the formal model presented here down to the gate-level implementation of the decoder.

REFERENCES

[1] R. J. R. Back and K. Sere. *From Modular Systems to Action Systems*, in Proc. of Formal Methods Europe' 94, Spain, October 1994. Lecture notes on computer science, Springer-Verlag.

[2] R. J. Back and J. von Wrigth. *Refinement Calculus: A Systematic Introduction*, Springer-Verlag, April 1998.

[3] E. W. Dijkstra. *A Discipline of Programming*, Prentice-Hall International, 1976.

[4] G. D. Forney Jr. *Maximum-Likelihood Sequence Detection in the Presence of Intersymbol Interference*, IEEE Trans. on Information Theory, IT-18(3): 363-378, May 1972.

[5] H. -L. Lou. *Implementing the Viterbi Algorithm*, IEEE Signal Processing Magazine, September 1995.

[6] J. G. Proakis. *Digital Communications*, Third Edition, McGraw-Hill 1995.

[7] J. Tuominen, T. Säntti and J. Plosila. *Towards a Formal Power Estimation Framework*, Turku Center for Computer Science Technical Report Series, Number 672, March 2005, ISBN 952-12-1517-8.

[8] J. Tuominen and J. Plosila. *Formal Energy Estimation Framework*, Turku Center for Computer Science Technical Report Series, Number 696, June 2005, ISBN 952-12-1578-X.

[9] J. Tuominen, P. Liljeberg, and J. Isoaho. *Self-Timed Approach for Reducing On-Chip Switching Noise*, in Proc. IFIP SoC-VLSI 2003, December 2003, Darmstadt, Germany.

[10] A. J. Viterbi. *Error Bounds for Convolutional Codes and an Asymptotically Optimum Decoding Algorithm*, IEEE Transactions on Information Theory, April 1967; IT(2): pp. 260-267.

[11] A. J. Viterbi. *Convolutional Codes and Their Performance in Communication Systems*, IEEE Transactions on Communications Technology, October 1971; COM - 19(5): pp. 751-772.