Compiled-Code-Based Simulation with Timing Verification

W. Hahn, A. Hagerer, C. Herrmann

Faculty of Mathematics and Computer Science, University of Passau
Innstr. 33, 94032 Passau, F.R. Germany

Abstract

Due to the complexity of today's systems, prototyping by simulation must be based on simulation-engine-like performance. It is proved by implementations that compiler-driven strategy is the best approach for high-performance simulation engines. However, proponents of the table-driven strategy for simulation engines claim that concerning timing simulation the advantage of the compiler-driven strategy only exists for zero-delay and unit-delay simulation but there would be no competition concerning nominal-, fixed-, and precise-delay simulation. It is the intention of this paper to contradict this claim by proving that all types of timing models can be defined in an algebraic way and compiled into code for the Munich Simulation Computer, an event-flow computer for high-performance compiler-driven logic simulation. Experimental results are presented and discussed concerning code size as well as code execution overhead.

Keywords

Timing Verification, Compiler-Driven Logic Simulation, Update-Dataflow (Event-flow) Computing

Introduction

Simulation engines are the only answer to the simulation bottleneck of system simulation [1]. They can be classified into systems that accelerate the table-driven simulation strategy, e.g., ZYCAD´s family of System Evaluators, and into systems that accelerate the compiler-driven simulation strategy, e.g., IBM´s Engineering Verification Engine, the Munich Simulation Computer MuSiC [2, 3, 4], and others. These two types of simulation engines strictly differ in their performance potential. Table-driven simulation [5] can only exploit algorithmic parallelism and is restricted by a centralized time-wheel mechanism that handles the progress of simulation time. Compiler-driven simulation [6], however, can massively exploit a design's data parallelism. Due to this fact, the performance potential of compiler-driven simulation is orders of magnitude higher.

Proponents of table-driven simulation engines, however, claim that concerning timing simulation this advantage of compiler-driven simulation engines only exists for zero-delay and unit-delay simulation. As an answer, this paper shows how to define and to code all common types of timing models at gate-level as well as at functional-level for compiler-driven simulation without sacrificing too much of the superior performance potential.

The paper is organized in the following way: A first section introduces event-flow computation as an efficient principle for design simulation. The next two sections offer a new way to express timing behavior of design components for compiled-code gate-level and higher-level simulation by boolean equations and an evaluation of the two approaches concerning code size overhead as well as code execution overhead.

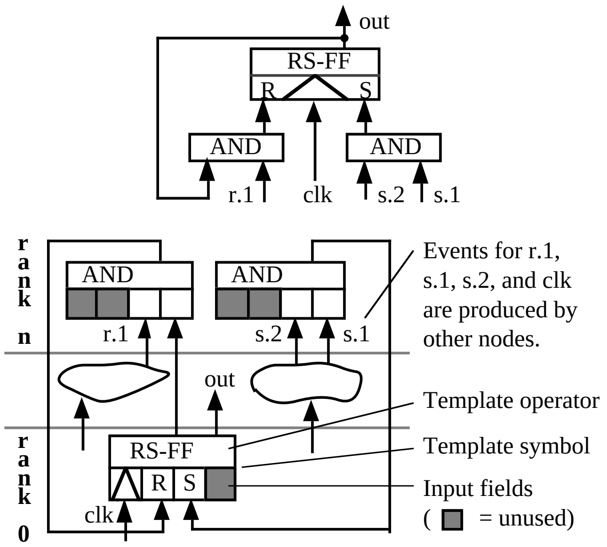


Figure 1: Transformation of a design into a rank-ordered event-flow graph

1. Event-flow computer architecture

The architecture principle of the event-flow approach [2, 3, 4] used to accelerate simulation of digital systems is compiling a system's description written in a common hardware description language into a directed graph, the system's program graph. The nodes of the graph represent system elements. The functionality bound to a node depends on the description level, e.g., transistor-functionality at switch-level, gate-functionality at gate-level, functionality of combinational networks (functional operators) at register-transfer-level, and so on up to user-specific algorithms at functional-model level. Arcs between nodes reflect interconnecting wires. They specify which nodes produce results that are used as arguments of a successor node's function.

In an implementation of the event-flow architecture, a node and its arcs to succeeding nodes are represented by an instance called a template. Each consists of an operator, up to four input fields, and an arbitrary number of references to input fields of succeeding templates according to the graph's connectivity. The number of input fields is restricted since the implementation aims at a simulation engine. Template operators define functions on operands in four-valued logic, i.e., over the values 0, 1, Z (high-impedance), and ? (unknown). As for any compiled-code simulator and as shown in Figure 1, templates are rank-ordered due to their data dependencies where storing elements are placed into rank 0 and primary inputs in rank 1.

The following three rules define the event-flow model of computation, i.e., the way a rank-ordered program graph is executed:

- Selection rule: Only templates are selected at which no further data can arrive at an input field, i.e., all templates placed on the same rank become selected.
- Event-flow firing rule: A template that passes the selection rule must be evaluated, i.e., the template's operation as defined by its operator is computed using the input data as arguments, if it received at least one data value at an input field after its last evaluation.
- Execution and result distribution rule: Template evaluation produces data that will be visible to other templates if and only if the operation's result is different from the result of the last evaluation. This datum is stored in the template as "last result". For each reference to a succeeding template, i.e., for each arc, the result is transmitted to update the referenced input field.

An evaluation cycle, i.e., a simulation step, starts with selecting rank 1 for execution, progresses via rank 2, 3, and so on, up to the rank with the maximum rank number n and ends after execution of rank 0. Whenever rank 0 becomes computed, simulation time is incremented by a fixed time unit that is independent of the evaluation cycle number and determined before graph execution starts.

Event-driven simulation by the event-flow model of computation is highly efficient since the selection scheme prevents template evaluation as long as new input data can arrive and the firing rule in combination with the result distribution rule reduces the set of templates to be evaluated to those templates that may contribute to a new design state due to a changed input situation.

MuSiC is an implementation of this architecture for multi-level logic simulation. To speed up rt-level, functional-level, and functional-model-level simulation, input fields of templates can store vectors of up to eight logic values and the template's operator is applied either on all values in parallel or in case of operators like 'increment' (INC), 'shift&concatenate' (SH&CAT), 'select' (SEL), etc. by handling all values as single objects.

By an operational rt-model of an 8-processing-unit version [7], MuSiC performance was evaluated as a function of event probability p_e per signal and different workloads concerning zero-delay simulation. Performance extrapolation to a 256-processing-unit version matches earlier given performance predictions [3, 4]: MuSiC can offer simulation performance of up to some 10^{10} gate evaluations per second (g/s). All experiments show that MuSiC's performance potential is better than that of table-driven simulation engines [e.g., 8]. The question, however, is how to make this superior performance potential available for timing verification?

2. Gate-level timing verification

Logic simulation computes the state of each design element as a function of time using a model of the design. For doing this correctly, delays within an element and on wires must be considered. In the event-flow architecture, templates, i.e., MuSiC instructions, do not model the delay behavior of the element they represent. Since simulation time is increased by a fixed time-unit ΔT after computation of rank 0, the following conception holds: design elements that are represented by templates on rank 0 are modeled considering a delay of ΔT while all other elements are modeled disregarding their timing behavior. This means that MuSiC directly supports zero-delay simulation where combinational elements, coded by templates in ranks 1 to n, have no delay and only registers, coded by templates in rank 0, have a delay of $\Delta T=1$.

In reality, each element p shows an element-specific propagation delay ΔT_p: if an event at an input of the element occurs at time t, then the effect of this event will not occur at the output until $t+\Delta T_p$. This can be modeled by splitting the element into a purely functional component with output $O_{fct}(t)$ followed by a separate delay component that computes the element's output O(t) from its input $I(t) = O_{fct}(t)$ due to the specific timing model.

2.1 Defining functions for delay components

Instead of scheduling events and managing lists of events, in the event-flow architecture a delay component is modeled in the same parallel and distributed way that is applied for computation of the functional behavior. The output of a delay component is computed by a function f_{delay} that transforms a time-based stream of values that consists of the recent value I(t) and the signal history of previously computed values I(t-1), …, $I(t-\Delta T_p)$ into a single value O(t). Modeling delay components for execution by MuSiC now means to define the component's function f_{delay} and to represent the function in form of a graph.

The definitions of f_{delay} use the following notations:

Delay specifications:

ΔT	propagation delay,
ΔR	propagation delay of a rising edge,
ΔF	propagation delay of a falling edge,
Δm	minimum propagation delay,
ΔM	maximum propagation delay,

ΔRm	minimum propagation delay of a rising edge,
ΔRM	maximum propagation delay of a rising edge,
ΔFm	minimum propagation delay of a falling edge,
ΔFM	maximum propagation delay of a falling edge,

Operator symbols:

NOT	the not-operator in four-valued logic,
AND	the and-operator in four-valued logic,
OR	the or-operator in four-valued logic,
<>	a comparison operator, c = a <> b, defined by:

c =	a = O	a = 1	a = Z	a = ?
b = O	O	?	?	?
b = 1	?	1	?	?
b = Z	?	?	Z	?
b = ?	?	?	?	?

Furthermore, let the repeated application of an operator OP, i.e., AND, OR, <>, on a history of logic values from time t - a to time t - b be written as:

$$\mathop{\mathrm{OP}}\limits_{i=a}^{b} I(t-i) = I(t-a)\ \mathrm{OP}\ I(t-(a+1))\ \mathrm{OP}\ \ldots\ \mathrm{OP}\ I(t-b)\,.$$

Then, the behavior of delay components can be expressed by the following boolean functions:

1. Zero-delay model
   $\Delta T = 0$: $O(t) := I(t)$
2. Unit-delay model
   $\Delta T = 1$: $O(t) := I(t-1)$
3. Nominal-delay model
   $\Delta R = \Delta F = \Delta T$: $O(t) := I(t - \Delta T)$
4. Fixed-delay model
   Since it may hold $\Delta R<\Delta F$ as well as $\Delta R>\Delta F$, two different equations are necessary:
   $\Delta R < \Delta F$: $O(t) := \mathop{\mathrm{OR}}\limits_{i=\Delta R}^{\Delta F} I(t-i)$ $\Delta R > \Delta F$: $O(t) := \mathop{\mathrm{AND}}\limits_{i=\Delta F}^{\Delta R} I(t-i)$
5. Precise-delay model I
   The less accurate version of the precise-delay model assigns the same (minimum/maximum) propagation delay to the rising edge and to the falling edge:
   I. $\Delta Fm = \Delta Rm = \Delta m < \Delta FM = \Delta RM = \Delta M$:
   $$O(t) := \mathop{<>}\limits_{i=\Delta m}^{\Delta M} I(t-i)$$
6. Precise-delay model II
   The accurate version of the precise-delay model distinguishes the (minimum/maximum) propagation delay of a rising edge from the propagation delay of a falling edge. Where only the relation of two propagation delays had to be considered for the fixed-delay model, now all relations of two intervals $\Delta Rm \le \Delta RM$ and $\Delta Fm \le \Delta FM$ are important. Sixteen different situations may occur but they can be mapped on just seven relations where the first relation is that of the precise-delay model I. Therefore, additional equations are only necessary for the six remaining relations. This leads to the following equations:
   II. $\Delta Rm \le \Delta RM \le \Delta Fm \le \Delta FM$:

$$O(t) := \mathop{\mathrm{OR}}\limits_{i=\Delta Rm}^{\Delta Fm} I(t-i) <> \mathop{\mathrm{OR}}\limits_{i=\Delta Rm}^{\Delta FM} I(t-i) <> \mathop{\mathrm{OR}}\limits_{i=\Delta RM}^{\Delta Fm} I(t-i) <> \mathop{\mathrm{OR}}\limits_{i=\Delta RM}^{\Delta FM} I(t-i)$$

III. $\Delta Rm \le \Delta Fm < \Delta RM \le \Delta FM$:

$$O(t) := \mathop{\mathrm{OR}}\limits_{i=\Delta Rm}^{\Delta Fm} I(t-i) <> \mathop{\mathrm{OR}}\limits_{i=\Delta Rm}^{\Delta FM} I(t-i) <> \mathop{\mathrm{AND}}\limits_{i=\Delta Fm}^{\Delta RM} I(t-i) <> \mathop{\mathrm{OR}}\limits_{i=\Delta RM}^{\Delta FM} I(t-i)$$

IV. $\Delta Rm < \Delta Fm < \Delta FM < \Delta RM$:

$$O(t) := \mathop{\mathrm{OR}}\limits_{i=\Delta Rm}^{\Delta Fm} I(t-i) <> \mathop{\mathrm{OR}}\limits_{i=\Delta Rm}^{\Delta FM} I(t-i) <> \mathop{\mathrm{AND}}\limits_{i=\Delta Fm}^{\Delta RM} I(t-i) <> \mathop{\mathrm{AND}}\limits_{i=\Delta FM}^{\Delta RM} I(t-i)$$

V. $\Delta Fm < \Delta Rm < \Delta RM < \Delta FM$:

$$O(t) := \mathop{\mathrm{AND}}\limits_{i=\Delta Fm}^{\Delta Rm} I(t-i) <> \mathop{\mathrm{AND}}\limits_{i=\Delta Fm}^{\Delta RM} I(t-i) <> \mathop{\mathrm{OR}}\limits_{i=\Delta Rm}^{\Delta FM} I(t-i) <> \mathop{\mathrm{OR}}\limits_{i=\Delta RM}^{\Delta FM} I(t-i)$$

VI. $\Delta Fm \le \Delta Rm < \Delta FM \le \Delta RM$:

$$O(t) := \mathop{\mathrm{AND}}\limits_{i=\Delta Fm}^{\Delta Rm} I(t-i) <> \mathop{\mathrm{AND}}\limits_{i=\Delta Fm}^{\Delta RM} I(t-i) <> \mathop{\mathrm{OR}}\limits_{i=\Delta Rm}^{\Delta FM} I(t-i) <> \mathop{\mathrm{AND}}\limits_{i=\Delta FM}^{\Delta RM} I(t-i)$$

VII. $\Delta Fm \le \Delta FM \le \Delta Rm \le \Delta RM$:

$$O(t) := \mathop{\mathrm{AND}}\limits_{i=\Delta Fm}^{\Delta Rm} I(t-i) <> \mathop{\mathrm{AND}}\limits_{i=\Delta Fm}^{\Delta RM} I(t-i) <> \mathop{\mathrm{AND}}\limits_{i=\Delta FM}^{\Delta Rm} I(t-i) <> \mathop{\mathrm{AND}}\limits_{i=\Delta FM}^{\Delta RM} I(t-i)$$

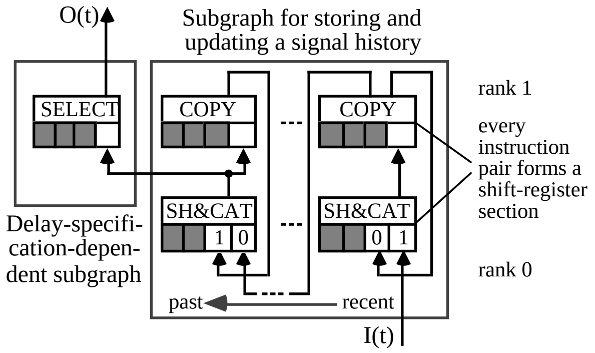

O(t)
Subgraph for storing and updating a signal history
SELECT
COPY
COPY
SH&CAT
1
0
SH&CAT
0
1
rank 1
every instruction pair forms a shift-register section
rank 0
Delay-specification-dependent subgraph
past
recent
I(t)

Figure 2: Coding scheme for fixed- and precise-delay models (demonstrated is nominal-delay)

2.2 Transforming delay equations into program graphs

For timing verification a gate is modeled concerning its functional behavior by a single MuSiC instruction and concerning its timing behavior by a program graph by which one of the previous delay equations is coded according to the required timing-verification accuracy.

MuSiC's program graph execution scheme supports coding of zero-delay models directly and coding of unit-delay models straightforwardly by placing all instructions into rank 0 that represent elements with unit delay (for zero-delay simulation only register instructions, for unit-delay simulation additionally all gate instructions). However, copy instructions are necessary to keep instructions of rank 0 data-independent.

Coding the other delay equations requires storing and updating of signal histories beyond one time increment. This is done by SH&CAT-instructions (they shift the data vector of input field 2 one position to the left and concatenate the left-most position of the data vector of the right-most input field 1). Since SH&CAT-instructions have to be considered as storing elements and are therefore placed into rank 0, input histories are updated whenever simulation time progresses.

As indicated by Figure 2, coding of the remaining delay equations is done by combining two subgraphs. The first subgraph stores and updates the particular signal history based on SH&CAT-instructions and the other subgraph evaluates the recent output value. The latter subgraph results from transforming f_{delay} into a graph by using MuSiC instructions. In Figure 2, however, this subgraph simply consists of a "select"-instruction since the delay specification to be implemented is nominal-delay with $\Delta R = \Delta F = \Delta T$. In this case, the value that corresponds to $I(t–\Delta T)$ has to be se lected out of the past-most section of the signal history as the signal at position $\mathrm{mod}(\Delta T/v)$, where $v = 8$ is the vector length of input fields. Obviously, the complexity of delay-specification-dependent subgraphs increases with the accuracy of the timing model.

2.3 Efficiency of gate-level delay models

There are two different criteria concerning code overhead: on the one hand, static code overhead due to the number of instructions additional to zero-delay code, and on the other hand, dynamic code overhead due to the number of instructions to be executed additionally to zero-delay code.

Since subgraph construction for delay equations follows a strict scheme for every timing model, the mean number of additionally necessary instructions can closely be precalculated [10] as shown in Table 1. The accuracy of this calculation was verified by analyzing generated code of example designs and comparing the results with calculated results.

ΔF (time units)	ΔR (time units)	Instructions	Executed instructions event-flow	Executed instructions data-flow
$1 \le \Delta F \le 8$	$1 \le \Delta R \le 8$	4	25	32
$9 \le \Delta F \le 16$	$9 \le \Delta R \le 16$	6	49	96
	$1 \le \Delta R \le 8$	8	59	128
$17 \le \Delta F \le 24$	$17 \le \Delta R \le 24$	8	73	192
	$9 \le \Delta R \le 16$	10	83	240
	$1 \le \Delta R \le 8$	12	93	288

Table 1: Number of delay-specification-dependent instructions for coding a fixed-delay model of a single gate

According to the event-flow execution scheme, the dynamic code overhead depends on event probability p_e per signal. An example is also given in Table 1, that, by the way, proves the advantage of event-flow computation over data-flow computation for logic simulation.

Due to the subgraph construction scheme and to a regular flow of events within a subgraph, for every time increment a lower bound and an upper bound for the number of event-activated instructions can be calculated [10]. First, calculation is based on the assumption that all instructions representing a logic element's function, produce an event only once per design's clock period. In a signal history of width ΔT, this single event causes exactly ΔT updates till the signal history reaches a steady state. The corresponding number of executed instructions activated by these updates are used to calculate the lower overhead bound. On the other hand, timing models serve in logic simulation to make spikes and hazards visible. Due to this, instructions that represent a logic element's function can produce an event more often than once per clock period. Therefore, the second assumption is that a logic element receives an event at each input at each simulation step and that each input event results in an output event. Calculating the execution overhead based on this assumption gives an upper overhead bound.

These calculations were applied to a set of example designs and the results are shown in Figure 3. It holds that the dynamic code overhead for fixed-delay and precise-delay simulation is in the extreme range from 2 for very small p_e-values to 90 for p_e-values close to one. Usual values for p_e, however, are less than 15 %. As marked in Figure 3, for these p_e-values, dynamic code overhead is less than 12, i.e., performance degradation is even for the most complex timing models not more than one order of magnitude.

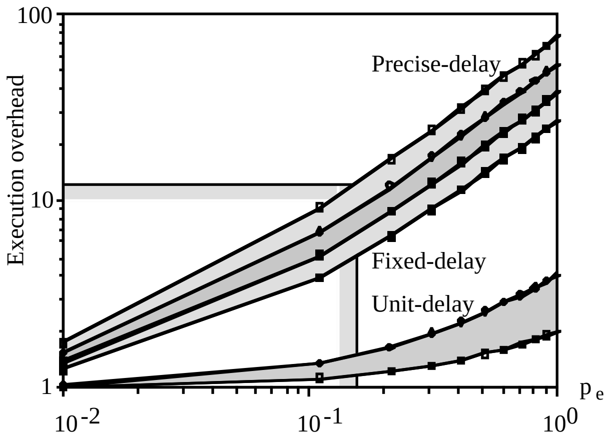


Figure 3: Dynamic code overhead (executed instructions of the timing model code / executed instructions of zero-delay code) for different timing models as a function of event probability p_e per signal

3. High-level timing verification

Unfortunately it holds that timing simulation of gate-level representations of rt-level elements and of functional level elements, e.g., flipflop, adders, etc., often yields too pessimistic results [9]. For some simulation steps the simulator computes a value "uncertain" for the element's state, although the real system's state is certain as shown by continuous simulation of the whole functional element. In logic simulation each gate's behavior is simulated independently of the interactions between the full set of gates that build an element of higher functionality [11]. Therefore, new algorithms for compiled-code simulation of complex timing behavior are necessary.

3.1 Defining the delay model

Generally, due to their transformation function ζ, functional elements transform a vector I of n logic values into a vector O of w logic values. Like gates, functional elements can be modeled by separating timing and functional behavior. Concerning timing behavior, modeling consists of generation of a modified input situation, classification of conditions fulfilled by this modification concerning the element's input and output signals, and synthesis of the delay behavior of output signals according to fulfilled conditions [12]. Therefore, as depicted in Figure 4, the model of a functional element's transformation function ζ is based on a modification function μ, on a function φ describing the pure functional behavior of the element, on an analysis function α and on a set of synthesis functions $\sigma_0, \ldots, \sigma_{w-1}$, one for each component of the element's output. Each transforms vectors of logic values into one vector.

The modification function μ has to change the vector I of input signals to model inertial delays or to react to violated time constraints. E.g., if a timing condition of a functional element *d-latch* is not fulfilled, a change to the state '?' can be enforced by modifying the input signal *clock* to '1' and the input signal *data* to '?'. Therefore, μ computes a vector I_{mod} of modified input signals and a vector C of input condition signals. These signals indicate if specific input situations, i.e., specific combinations of input and output signals, occur. By using I_{mod} the function φ mimics the behavior of the functional element by computing a signal vector O_{fct} without considering delays.

The function α analyzes the input vector I, the modified input vector I_{mod}, the input condition signals C, and the signal vector O_{fct}. The result is a vector Q of q condition signals. A condition signal Q[k](t), $0 \le k < q$, indicates if a specific delay situation k that is described by the designer occurs at time t. For a component O[i] of the output vector O a delay situation k specifies a minimum delay $\Delta m_{i,k}$ and a maximum delay $\Delta M_{i,k}$ according to the data sheets.

A synthesis function σ_i for the output's component i determines the minimum and the maximum delay that has to be applied to the value of O_{fct}[i] to compute ζ(I). The minimum delay δ_i^m is the minimum of the minumum delays of those delay situations k that possibly (Q[k]='?') or certainly (Q[k]='1') occur. The maximum delay δ_i^M is determined analogously. Based on the delays δ_i^m and δ_i^M, on the value O_{fct}[i](t) of the signal O_{fct} at time t and, on a prediction of the future values of signal O[i] that was determined in the previous simulation step and therefore beginning at time t-1, the synthesis function σ_i predicts all future values of O[i] beginning at time t. Assuming that the delay situation will not change after time t the signal O[i] will be stable at least after $\delta_i^{max} := \max(\{\Delta m_{i,k}, \Delta M_{i,k} \mid 0 \le k < q\})$ time units. Therefore, it is sufficient to predict $\delta_i^{max}+1$ values: For a time t+j between $t+\delta_i^m$ and $t+\delta_i^M-1$ the value '?' is predicted. In this case, no certain information exists about the signal's value after δ_i^m time units and before δ_i^M time units. For a time after $t+\delta_i^M$ the value O_{fct}[i](t) is scheduled. Therefore, at time t it is assumed that the signal will be stable with the value d after the maximum delay. For time t+j, $j<\delta_i^m$, the value is chosen from the prediction that was calculated at the simulation time t-1. The signal does not change it's value before the minimum delay. However, considering the case that the element's delay situation does not change, the computation can be seen as a shift register since the value of the component j at time t is shifted through components with indices lower than j, until it is placed at time t+j in component 0, i.e., until it forms the value of the element's output signal O[i].

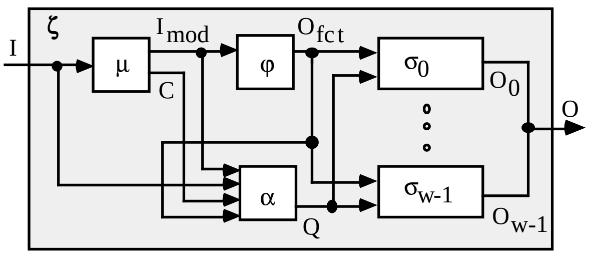
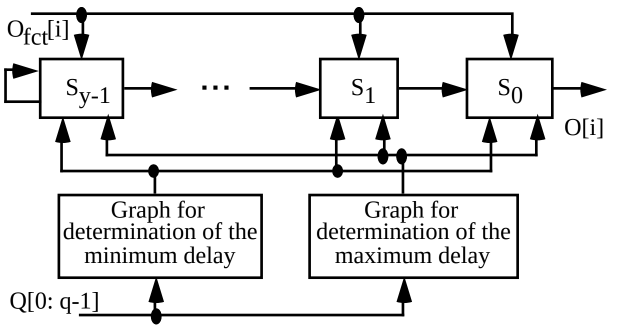


Figure 4: Representation of the functional and timing behavior of a functional elements by a set of functions

3.2 Transforming delay models of functional elements into program graphs

All functions of a model describing the timing behavior of a functional element are specified by the designer, e.g., by using a specification language based on functions. Coding of timing specifications is not only a problem of mapping them on MuSiC's instruction set but also a problem of placing instructions into specific ranks: if a specification function's result at time t depends on the result at time t-1, e.g., functions like EVENT, RISE, etc., the result has to be offered by an instruction that is placed into rank 0. Other instructions necessary for computation of the function are placed into higher ranks in order to have access to a value computed in the previous simulation cycle.

Transformation of a functional element's timing behavior is performed in two steps. First, subgraphs that code boolean functions are generated according to the specifications of the modification function μ and the analysis function α. Second, the graphs for computing the synthesis functions are created. As depicted in Figure 5, the graph of every σ_i consists of a subgraph to compute the prediction of the future values and two subgraphs to compute the minimum delay and the maximum delay according to the delay situation that is determined by the result of the analysis function. Due to the limitation of the operand's vectorization, the graph computing the future values consists of $y = \lceil (\delta_i^{max}+1)/8 \rceil$ slices. Each slice S_j, $0 \le j < y$, internally computes the future values O(t+8·j)-O(t+8·j+7) and offers the component O(t+8·j) as a result to other slices. The slice S_j uses the signal O_{fct}[i], two control signal vectors indicating if $t+8\cdot j < \delta_i^m$ resp. $t+8\cdot j > \delta_i^M$, and the result of slice S_{j+1} as arguments. Since no result can be offered to the slice S_{y-1} a feedback loop is used to realize the replication of $O[i](t+\delta_i^{max})$.



$O_{fct}[i]$
S_{y-1}
...
S_1
S_0
O[i]
Graph for determination of the minimum delay
Graph for determination of the maximum delay
Q[0: q-1]

Figure 5: Structure of the program graph for a synthesis function

Design and experiment characteristics	D-register (8 bit)	CLA-adder (8 bit)
No. of inputs / No. of outputs	9 / 8	16 / 8
No. of time constraints	9	32
No. of delay cases	8	48
Maximum delay (U time units)	6	8
Gate complexity (G)	40	84
Static no. of instructions (sI)	240	401
Static no. of instructions per gate (= sI/G)	6.0	4.8
Average dynamic no. of instr. (dI)	648	840
Event rate (=dI/(sI·U)) in % per time unit	45	26
∅–no. of instr. exec. per gate (= dI/G)	**16.2**	**10.0**

Table 2: Overhead characterization for high-level delay models of two functional elements

3.3 Efficiency of high-level delay models

The size of a program graph generated according to the timing specification, the static code overhead, depends on the size of the input and output vectors of the functional element. Furthermore, it depends on the complexity of the functions μ and α. It is also influenced by the range of the delay values (size for the graph of the synthesis functions). As indicated by Table 2, the static code overhead slightly increases with increasing complexity of the functional element and its timing model since common subexpressions in the delay specification are represented by only one subgraph.

Dynamic code overhead is again the number of instructions to be executed additionally. It is here also reduced by the event-flow principle. It holds that the amount of these instructions is far less than the amount necessary for coding the timing behavior of a gate-level representation gate by gate. Additionally, when simulating the delays of a whole functional element, coarser simulation steps can be chosen without loosing precision compared with simulating every gate separately.

Experimental results show that for complex functional elements the static as well as the dynamic overhead tends to be better than the overhead when simulating a gate-level representation of the functional element for timing-verification purposes [12]. Again, compared with simulation at gate-level in zero-delay mode the performance is reduced by only one order of magnitude.

Conclusion

For timing verification of digital systems, a new approach for compiler-driven simulation was presented. All types of timing models could be included into program graphs to be executed by the Munich Simulation Computer or by any other compiler-driven (programmed) simulator which functionality is recently restricted to zero-delay and unit-delay mode. Although for the most accurate precise-delay simulation execution overhead may decrease MuSiC's performance more than that of table-driven simulation engines, simulation speed remains superior to these engines and, in case of MuSiC, in the range of billions of gate evaluations per second.

References

[1] Blank, T.; 1984: "A Survey of Hardware Accelerators Used in Computer-Aided Design", *IEEE Design & Test of Computers* 1, (3), 21-39.

[2] Hahn, W., Fischer, K.; 1985: "An Event-Flow Computer for Fast Simulation of Digital Systems", *Proc. 22nd ACM/IEEE Design Automation Conference*, 338-344.

[3] Hahn, W.; 1986: "Event-Flow Computation as Key to Fast Digital Design Simulation", *Microprocessing and Microprogramming - The Euromicro Journal* 18, 27-38.

[4] Hahn, W.; 1987: "The Munich Simulation Computer: Design Principles and Performance Prediction", In *Hardware Accelerators for Electrical CAD.* Ambler, T., Agrawal, P., Moore, W. (Eds.). Adam Hilger, Bristol, U.K.

[5] Szygenda, S.A., Thompson, E.W.; 1975: "Digital Logic Simulation In a Time-Based, Table-Driven Environment - Part 1. Design Verification", *IEEE Computer,* (3), 34-37.

[6] d'Abreu, M.A.; 1985: "Gate-Level Simulation", *IEEE Design & Test of Computers* 2, (6), 63-71.

[7] Hahn, W. Anger, H. Hagerer, A., Schuster, B.; 1988: "A Multi-Transputer-Net as a Hardware Simulation Environment", *Microprocessing and Microprogramming - The Euromicro Journal* 25, 291-298.

[8] ZYCAD; 1993: See most recent System Evaluator Specifications.

[9] Breuer, M.A., Friedman, A.D.; 1976: "Diagnosis and Reliable Design of Digital Systems", *Computer Science Press.*

[10] Eisenhut, M.; 1990: "Realisierung und Bewertung verschiedener Zeitmodelle der Simulation digitaler Systeme in Ereignisflußgraphen", *Diploma Thesis*, University of Passau, Fac. of Math. & Comp. Sci.

[11] Benkoski, J., Strojwas, A.J.; 1989: "Timing Verification by Formal Signal, Interaction Modeling in a Multi-Level Timing Simulator", *Proc. 26th ACM/IEEE Design Automation Conference*, 668-673.

[12] Herrmann, C.; 1993: "Simulation von Signalverzögerungen und Zeitbedingungen funktionaler Elemente auf MuSiCII", *Diploma Thesis* , University of Passau, Fac. of Math. & Comp. Sci.