# System Level Simulation Concepts

Franz J. Rammig
Paderborn University
Dept. of Mathematics and Computer Science
Paderborn, Federal Republic of Germany

Abstract

*In this paper special aspects of simulating digital equipment at the system level are addressed. After a short characterization of the different levels of abstraction the conceptual foundations of the system level are discussed. The more active point of view as represented by communicating processes is compared with the more reactive one of event triggered activities. CSP and Interpreted Petri Nets serve as examples for the different classes. These concepts are directly reflected by modelling concepts for the description of such systems. VHDL as a standardized traditional approach is compared with more innovative ones. As examples of such specification techniques object-oriented approaches and graphical languages are discussed. Not all simulation techniques are well suited for system level applications. Again reflecting the basic concepts, process-oriented techniques and event scheduling seem to be the most appropriate solutions.*

## 1. Levels of Abstraction

Today in the area of digital hardware design there is a rather widely accepted scheme of 6 abstraction levels [RA1]. This scheme is orthogonal to the different views a system is looked at. As example for this Gajsky [GA1] identifies three different views: behaviour, structure, and geometry. Additional views are possible, e.g. a test view. In the context of simulation the behaviour view is of main interest. So in this context the levels of abstraction are discussed only with this view in mind. To point out the features that define the system level a bottom-up presentation has been chosen. The lowest level (level 1) usually is called the **electrical level**. Here it is modelled, how electrical circuits built from resistors, capacitors, etc. behave over the time axis. This is done by a system of differential equations. I.e. both the time axis and the observable values are represented by a continuous domain. It should be noted that the geometrical view of this level is the (metric) layout which doesn't constitute an own level. There are some simulators at this level, SPICE in numerous versions [VL1] being the most widely used. The **switch level** (level 2) is the next abstract one. This level is rather accepted in digital MOS design but makes sense in other digital designs as well. The abstraction comes from modelling transistors as ideal on/off switches and the connections between as discrete capacitances. So the value domain is a discrete one now where a value is given by a pair consisting of a logical interpretation and a strength. Both components are from a finite domain. This abstraction introduces uncertain values. They are handled either by introducing additional "values" or by representing uncertainty by enumerating all possible values [LR1]. The time domain may still be a continuous one. Other approaches like MOSSIM [BR1] have a discrete one in mind (unit delay assumption). This leads to a concept to model switch level circuits by finite automata. Various switch level simulators exist, MOSSIM being the best known. The **gate level** (level 3) has a long tradition in digital systems design. It has a very nice mathematical background in Boolean algebra. However this models only the timeless behaviour. So some additional concepts have to be considered in order to cover the time axis as well. In the ideal case the value domain is restricted to Boolean values 0,1 while the time domain remains continuous. Again to solve the problem of uncertain values forces to introduce additional values. By this in most cases the underlying algebra is no longer Boolean. Even the concept of different strengths is carried over from the switch level in some cases. The operators however are always Boolean (logical) operators. This finally constitutes this level. If the value domain is restricted to 0,1 and the time model is unit delay then the modelling concept for this level is exactly a system of Boolean equations. Because of the long tradition there are numerous simulators for this level. They use different value domains and different timing models. HILO, CADAT, DISIM are some examples of commercially available simulation systems at this level. Today such simulators usually allow the user to define his own "macrogates" using so called "behavioural languages". These macrogates may become very complex. However the conceptual model is maintained, the oftenly cited "functional block level" is not really an own one. This is the case with the **register transfer level** (level 4). At this level a specific mode of operation is assumed. There are components that continuously observe their specific conditions. Whenever the condition of a component becomes true this component performs its specific operation. In any case such an operation may be interpreted as a transfer of data between registers where the data may be modified during this transfer. This point of view gives that level its name. Abstraction in this case originates from implicitely underlying a specific mode of operation. In addition the elementary components used at this level are more complex (e.g. registers, ALUs, etc.) abstracting from their implementation. The value domain at this level is given by (uninterpreted) bitstrings while the timing model is the counting of clock ticks. So the time domain now has become discrete as well. The register transfer level is very helpful in clean synchronous designs, it forces somehow to design in this manner. Therefore this level has been studied intensively in academic institutions but is much less popular in industry. Therefore nearly all of the numerous register transfer simulation systems (e.g. CDL, DDL, RTS, KARL, ERES) are in use rather only in universities. At the **algorithmic level** (level 5) the reactive point of view at the register transfer level is inverted to an imperative one. While at the register transfer level the system is looked at from the eyes of the individual components, at the algorithmic level the controller's point of view is taken. In contrast to ordinary algorithmic descriptions, however, concurrency plays an important role in hardware design and therefore also at this level of abstraction. Therefore highly concurrent algorithms usually are described. While at the register transfer level it is specified precisely what conditions cause operations to be carried out it is abstracted from this information at the algorithmic level. Only the logical point of time, when an operation has to be carried out is identified. All the remaining stuff is hidden away by assuming the imperative mode of operation of a system. The domain of values may be freely definable but usually is restricted to bitstrings with interpretations attached (e.g. two's complement integer, IEEE standard floating point number, ASCII character). The timing model is either still a counting of clock ticks or a purely logical one. In

this case simply a causality structure is assumed as in usual algorithmic languages. Algorithmic languages, and in combination with these, algorithmic level simulators have been a purely academic area for a long time. The increasing complexity of digital systems and caused by this the need for high level synthesis tools make this level more and more attractive for industry as well. VHDL approaching this level makes it even more visible for the industrial practice. Up to now there are only very few commercial simulators available for this level. DACAPO III, VERILOG, ENDOT, and partially VHDL may serve as examples. Finally at the **system level** (level 6) it is abstracted from the algorithmic implementation of the system's instruction set. At this level the entire system is viewed at as a set of cooperating processors. Here the term processor is used in a wider sense to denote a subsystem with an instruction set that enables it to export certain services. A usual processor is the most typical example but channels, device-controllers, etc. fall into the same class. Such a component is characterized by the functions performed by the instructions and the protocol to be used to request a service (an instruction) to be executed. In principle the initiative within such a protocol may be located at the serving device or at the requester. E.g. in the case of a usual processor this processor takes the initiative by fetching an instruction (the service it is requested to perform) from memory without explicitely being triggered to do so. In addition to describe the components of a system and their instruction sets plus protocols the global interaction of these semiautonomous objects has to be specified. Dependent on the kind of system to be described this may be done in a centralized manner or in a decentralized one. In the first case another highly concurrent algorithm serves to specify the global behaviour while in the second alternative in a totally distributed manner the different components decide due to certain states or events to request certain services from other components. So the system level can be interpreted as an abstraction of the algorithmic level (centralized alternative) or the register transfer level (distributed way). Both the value domain and the timing model are purely symbolic at this level. There are freely definable types with arbitrary semantics and time is interpreted only to be advanced by causality. The system level up to now is supported by very few commercial simulators. DACAPO III and VHDL are approaching this level while performane analysis tools like HIT [BS1] are addressing the system level as well.

# 2. System Level Modelling Concepts

## 2.1 Modelling the Component's Functionality

As noted above a component is characterized by its instruction set, i.e. the functions performed by the different instructions. From the implementation of these instructions it is abstracted as far as possible. An algebraically well investigated model for this purpose is the theory of abstract data types (ADT)[EM1, GH1]. An ADT **D(S,E)** is by a signature **S** and a set of equations **E**. A signature **S (sorts, ops)** is given by a set of **sorts** (domain identifiers) and a set **ops** of operations defined on these sorts. The signature specifies the syntax of the ADT. The semantics is given by a set of equations to be respected. Example: The following ADT that defines the Boolean algebra may also be interpreted as the specification of a processor with instruction set {T, F, not, and, or} that performs these operations in a way that the Boolean algebra is respected. How these operations are performed and how the arguments look like is completely abstracted from.

```
type Boolean is

  sorts Boolean

  opns T,F : Boolean;
   {nullary operations, i.e.  constants}
   not :  Boolean → Boolean; {unary operation}
   and, or :  Boolean, Boolean → Boolean;
   {binary operations}

  eqns or(a,F) = a;
     and(a,T) = a;

     or(a,b) = or(b,a);
     and(a,b) = and(b,a);

     and(a,or(b,c)) = or(and(a,b),and(a,c));
     or(a,and(b,c)) = and(or(a,b),or(a,c));

     and(a,not(a)) = F;
     or(a,not(a)) = T;

     or(a,or(b,c)) = or(or(a,b),c);
     and(a,and(b,c)) = and(and(a,b),c);

     or(a,a) = a;
     and(a,a) = a;

     or(not(a),not(b)) = not(and(a,b));
     and(not(a),not(b)) = not(or(a,b));

     not(not(a)) = a;
endtype.
```

## 2.2 Specification of the Global Behaviour

A very useful concept to model the global behaviour on a condition/event basis is that of Timed Interpreted Petri Nets. Petri Nets [PE2] have been defined by Carl Adam Petri [PE1] as a straight foreward extension of finite automata. They model a system as a set of actions (called **transitions**) controlled by conditions (called **places**). Each transition decides locally whether its local executability condition is fulfilled or not. The entire concept is very simple:

**Def. 2.2.1 (Petri Net Graph)**

$PG = (P, T, E)$ is called Petri-Net-Graph $:\Leftrightarrow$
$P$ finite set (of "places")
$T$ finite set (of "transitions")
$E \subseteq (P \times T) \cup (T \times P)$
$P \cap T = \emptyset$
$\forall x \in (P \cup T) : \exists y \in (P \cup T) : (x, y) \in E \vee (y, x) \in E$
$\diamond$

A Petri Net Graph is just a certain structure. Behaviour is introduced by adding markings of places and functions to manipulate these markings:

**Def. 2.2.2 (Petri Net)**

$PN = (PG, m_o, R)$ is called Petri-Net $:\Leftrightarrow$
$PG = (P, T, E)$ Petri-Net-Graph
$m_o \in M = \{m \mid m : P \rightarrow \mathbb{N}_0\}$ (initial marking)
$R \in \{r \mid r : T \rightarrow f_T\}$
with $f_T = \{f_t \mid t \in T\} \wedge \forall t \in T : (f_t : M \rightarrow M)$
(firing rule of $T$)
$\diamond$

In Petri Nets places are used to model conditions. If a place is marked by at least one token this condition is interpreted to be true. Actions are modelled by transitions. A transition can fire if certain conditions at its input- and output-places are true. By its firing it modifies the markings of its input- and output-places. All this is specified by the firing rule $f_t$ associated to each transition t. In ordinary Petri Nets there is only one single firing rule assiciated to all transitions stating that a transition is firable if all its input places are marked with at least one token. Interpreted Petri Nets are obtained by associating operations on an additional data domain to transitions. Whenever a transition fires, its associated operation is performed.
If it is neccessary to model time spent by performing operations this may be added easily:

**Def. 2.2.3 (Timed Interpreted Petri Net)**

$TIPN = (IPN, \Delta)$ is called timed Interpreted Petri-Net : $\Leftrightarrow$
$IPN = (((P, T, E), m_0, R), I, D)$ Interpreted Petri-Net
$\Delta \in \{\delta \mid \delta : T \rightarrow \tau\}$ with
$\tau = \{o' \mid o' : dom(o') \subseteq \mathrm{X}\ (\mathrm{D}) \rightarrow I\!R\}$

Here D denotes the data domain with interpretations I being defined on. A timed interpreted firing is defined as follows: Let transition t become firable at point of time $t_0$. Then at this point of time the associated operation is initiated based on argument values valid at this very point of time. At the same time the delay function d(t)=o' is calculated based on its argument values at this point of time. The result of this calculation may be k. Then at point of time $t_0 + k$ the results of the associated operation are assigned to their target variables.
Petri Nets have a nice graphical representation that makes them very readable. There is a rich theory about and various extensions have been studied intensively.

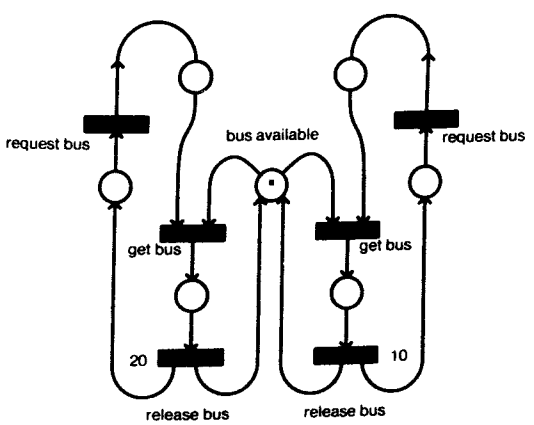


Fig. 1: Example of an Interpreted Petri Net (from [RA1])

Petri Nets are a completely event oriented approach while the model of Communicating Sequential Processes (CSP) [HO1] is based on the concept of processes. This model, too, is very simple though powerful. The entire system is represented by a set of concurrently active processes which internally are strictly sequential. The different processes have no shared resources and communicate only via messages where only synchronous communication is allowed.

**Def. 2.2.6 (Events and Processes)**
Objects to be specified are described by events. Such an event is looked at to be atomic. An example for such an event may be the assignment to a variable. The set of the events of an object is called the alphabet of this object. An arbitrary pattern of behaviour that can be described on this alphabet is a process.

**Def. 2.2.7 (Sequential Execution)**
Let be *a* an event and *P* a process. By *"seqbegin a ; P seqend"* it is denoted that first *a* has to happen and then *P* is started. It is defined that *a* is a process as well and also *"seqbegin a ; P seqend"*.

**Def. 2.2.8 (Recursion)**
Let *P* be a process. By *"while true do P"* it is denoted that *P* is repeated infinitely. If *P* is a process then also *"while true do P"*. Let *con* be a binary variable. By *"while con do P"* it is denoted that *P* is repeated as long as *con* is true. *"while con do P"* is a process as well.

**Def. 2.2.9 (Case Distinction)**
Let $P_1$,...,$P_n$ processes, *cnt* a variable with domain $\{c_1,...,c_n\}$. By *"case cnt of* $c_1$*:*$P_1$*;...*$c_n$*:*$P_n$ *caseend"* it is denoted that exactly this $P_i$ is executed whose $c_i$ is the actual value of *cnt*. If $P_1$,...,$P_n$ are processes then *"case cnt of* $c_1$*:*$P_1$*;...*$c_n$*:*$P_n$ *caseend"* is a process as well.

**Def. 2.2.10 (Input/Output)**
Let *chan* be a special variable of type channel and *var* an arbitrary variable. By *chan ! var* an output operation is denoted. The value of *var* is transferred via *chan*. The operation is not finished until the receiver (a concurrently active process, see Def. 2.2.12) has read the value. By *chan ? var* an input operation is denoted. It can be performed only if the channel *chan* is not empty. Initially any channel is empty. It can be filled only by an output operation. By an input operation the channel is emptied again. Input and output operations are interpreted as events.

**Def. 2.2.11 (Sequential Process)**
A sequential process is constructed by rules 2.2.6 - 2.2.10. Nothing else is a sequential process.

**Def. 2.2.12 (Concurrent Process)**
A sequential process is a concurrent one as well. Let *P* be sequential process and *C* a concurrent one. By *"conbegin P ; C concnd"* it is denoted that *P* and *C* have to be executed concurrently. *P* and *C* are initiated at the same time but then run independently (besides possible rendevous because of input/ output operations). The entire process *"conbegin P ; C conend"* is terminated if the last one of *P* and *C* is terminated. Concurrent processes must not share any resources besides of channels. If *P* is a sequential process and *C* is a concurrent one then *"conbegin P ; C conend"* is a concurrent process as well. Nothing else is a concurrent process.

### 2.3 Specification of Protocols

The protocols neccessary to request operations may be specified with each of the two concepts Timed Interpreted Petri Nets or CSP. It depends on the individual situation which concept is more appropriate.

## 3. System Level Modelling Languages

The system level is rarely supported by design languages. Recently by VHDL [VHD] a language has been standardized that is intended to cover the system level as well. From the above discussion this should mean that there are language features to specify abstract data

types and some mechanism for cooperating processes or something equivalent to Petri Nets. Processes are directly present in VHDL. A process in VHDL is initiated by a certain condition indicated in its "sensitivity list". In fact there is enumerated a list of variables the process reacts on value changes of. More than one process may be active concurrently. Once a process is activated it loops forever until it eventually reaches a suspension command. A process is a timeless and strictly sequential sequence of operations. However it may wait on specific events and may schedule future events to happen with a real-time axis in mind. Despite of originating from ADA, VHDL is not restricted to synchronous process interaction. In fact there is no primitive for message passing at all in the language. All communications is via shared "signals" (variables with a dimension in time). For such signals, as mentioned above, future values may be scheduled by a process and another process may wait on certain values of such signals. So the basic principle is asynchronous communications via shared resources. For a hardware description language this approach is fine as it is very general and by this imposes no restrictions on the systems to be designed. By proper use of the language the communications may be restricted to message passing and furthermore this message passing may be programmed in a synchronous way by always extending the simple sending to a handshake protocol. From this discussion VHDL seems to be nicely suited to support the system level from this point of view. On the other hand whenever time plays a role in a specification the time model of VHDL looks a little strange. A VHDL process being timeless even a sequence of operations that follow each other but all consume a certain amout of time cannot be described directly. In such a (rather usual situation) each elementary operation *Op* has to be replaced by a sequence:

i) schedule a virtual termination signal *t(Op)* for the intended termination time of *Op*

ii) *Op*

iii) wait on *t(Op)*.

Abstract data types are approached by VHDL by having imported the ADA package concept. This makes it possible to define the functionality of modules to be integrated into a system as packages to be used in the description. Unfortunately the software concept of shared code has been inherited without any change. So an arbitrary number of requests to services of such a package may be handled concurrently. In most cases this doesn't reflect the behaviour of real hardware components. The standard situation of a hardware component that can handle exactly one service at a time therefore has to be described in a rather complicated manner in VHDL. This situation is handled perfectly in DACAPO III [DAC]. This language offers a direct equivalent to implemented ADTs, called "export procedure" in this language. Such an export procedure exports (offers) a list of operations to its environment and makes sure that concurrent requests are handled properly. The basic principle of DACAPO is given by Timed Interpreted Petri Nets on the basis of which process models may be built as well. The time model is that in a sequence $Op_1$; $Op_2$ the operation $Op_2$ can start only after $Op_1$ has terminated. This means if some time consumption is attached to $Op_1$ then $Op_2$ waits for this time.

So both languages may be used for system level descriptions with DACAPO being more adequate in the behavioural aspects while VHDL offers more support in describing structure and configurations. For this purpose VHDL with its "port maps" and "configuration bodies" offers perfect support.

Object-oriented programming when extended by concepts for concurrency and for explicitely expressing protocols seem to be the most adequate description style for the system level. Up to now there are very few attempts into this direction. One example is POOL [OD1] a parallel object-oriented language which is intended for general purpose but has been used for high level hardware description as well. Another example is ODICE [MR1]. This experimental language has been defined and implemented to study a clean approach for integrating the concept of seperately specified protocols and of object- oriented programming into a conventional hardware description language. The experiment was made with DACAPO as host but may be possible with VHDL as well. ODICE uses the concept of object-oriented programming to define the module's functionality (the instructions interpreted as methods). Structures are generated by a generative concept similar to ZEUS [LK1] or MoDL [SA1]. Finally protocols are specified externally and bound to method-calls individually. Besides these concepts all basic expressing power of ODICE is inherited from the host language (DACAPO or VHDL).

With powerful graphical interfaces available today graphical specification languages for the system level become more feasable. Statecharts [HA1] and Structured Petri Nets [CK1] may serve as examples. Statecharts start with usual state diagrams of FSMs. In order to handle complexity, first of all a concept of hierarchy is added. So a state may be decomposed to an entire FSM and so on. This concept makes it neccessary to introduce means for specifying in which state such a macrostate has to start when activated. Similarly when leaving a macrostate it must be possible to store the last active microstate (history mechanism). As large systems usually are decomposed into cooperating automata, expressing power to describe concurrency has been added as well. In terms of Petri Nets by (nonhierarchical) cooperating automata the class of FSM decomposable Petri Nets is defined. Hierarchy seems to be added to Petri Nets as well. The most elegant way may be the approach of Structured Petri Nets [CK1]. In this approach a transition may be replaced by an entire Petri Net. Such a macro-transition becomes firable by the same condition as usual ones. The firing of such a macro-transition means that first the (always identical) initial marking is taken. Starting with this marking the (local) Petri Net becomes active and remains active as long as it is life. By this net becoming dead, the macro-transition plays its token game. When restricted to finite markings the Structured Petri Nets are equivalent to State Charts. While Statecharts may be characterized by "take FSMs, then add hierarchy, and finally add concurrency", Structured Petri Nets reverse the sequence of extensions: "take FSMs, then add concurrency, and finally add hierarchy".

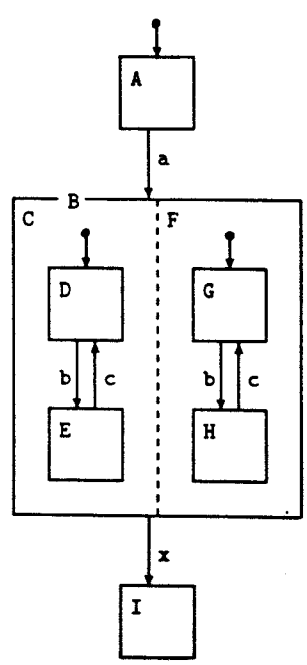


**Fig. 2: Example of a Statechart (from [HE1])**

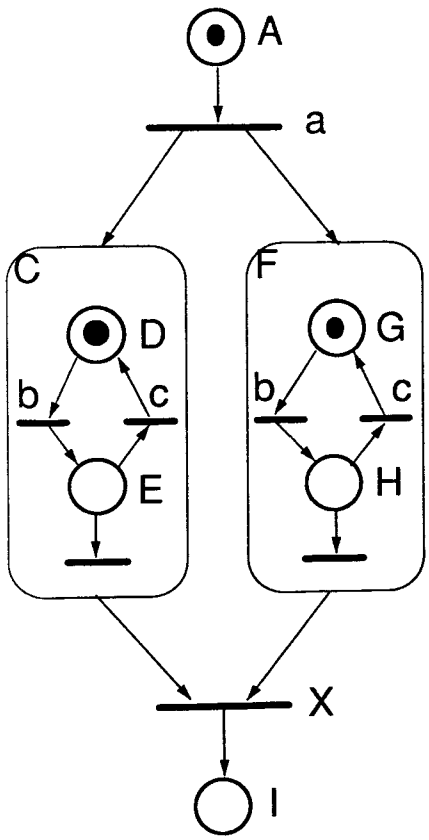


Fig. 3: Equivalent Structured Petri Net

## 4. System Level Simulation Techniques

The simulation technique to be used at system level depends on the major underlying modelling concept. If this concept is event-oriented then the well investigated technique of event scheduling is best suited. Today there are very efficient event scheduling algorithms available that can handle all levels of abstraction. This is due to the fact that there is a clean distinction between the scheduling mechanism and the events to be scheduled. These may be different at different levels of abstractions. At system level the typical event is a request to perform a certain service (operation). The action to be triggered by an event may be of arbitrary complexity. While at gate level it typically is the calculation of a Boolean function at system level it means calculation of the requested service. In both cases the action may cause aditional events to happen (infuencees). To increase performance in some recent simulators the model to be simulated is compiled into executable code as far as possible. Only the innermost kernel of the scheduling mechanism remains predefined. In principle this kernel looks like the following skeleton:

```
module main ;
  from circuit import
    circuit_size, word, data, influenee_nr, influencee,
                      executable, action, elapses ;
  const empty = 0 ;
  type  event = record
                  component_id  : integer;
                  event_time    : integer;
                  new_value     : word
                end;
  var   current_event, new_event : event  ;
        current_time, queue_fill : integer;
        changed                  : bit    ;

  begin
    time := 0 ;
    final_time := stop_time ;
    {stop_time to be supplied externally }
    while time <= final_time & queue_fill <> empty do
      begin
        current_event := event_queue . remove
        current_time  := current_event . event_time ;
        changed := data[current_event . component_id] <>
                          current_event . new_value ;
        data [current_event . component_id] :=
                            current_event . new_value ;
        if changed &
influencee_nr [current_event . component_id] > 0
  then begin
           for i :=
   1 to influencee_nr[current_event . component_id]
               do
                 begin
                   component :=
        influencee [current_event . component_id,i];
                    if executable ( component ) then
                  begin
                    new_event . component_id
                    := component ;
                    new_event . new_value :=
                    action (component) ;
                    new_event . event_time :=
                    current_time + elapses (component) ;
                    event_queue . insert ( new_event )
                  end
                                              else
                end
      end
   else;
     queue_fill := event_queue . test
   end
 end ;
end main .
```

System level simulators based on event scheduling may be supported properly by highly parallel simulation engines based on this principle. One of the most promising approaches seems to be the event-flow mechanism introduced by the MuSiC project [HF1]. This architecture modifies the data-flow paradigm to support event scheduling in a highly efficient way. Again there is a clean distinction between the event mechanism performed by the event-flow engine and the attached operations that are performed on general purpose processors (MC6800 in the first proposal, T800 in more recent ones). Process-oriented modelling concepts are easier to be simulated. A process mechanism is present on all nontrivial operating systems and is supported by a couple of languages including SIMULA or ADA. So the main task to be solved is the handling of real time and the fine grain

translation of the model's process concept to this one offered by the host system. In fact this may cause real problems. The problem of mapping broadcasting to a system that only knows point to point message passing like OCCAM may serve as an example. Process-oriented simulation systems may be parallelized in a straight forward way. Transputers seem to be a natural host system to do so. It should be noted, however, that there is no guaranteed linear or even near-linear speedup when going this way. Only if real process-oriented models are handled by such systems they perform properly.

## 5. References


[BR1] R.E. Bryant :
MOSSIM: A Switch Level Simulator for MOS-LSI
in: Proceedings 18th DAC, 1981

[Be1] H. Beilner :
Workload Characterization and Performance Modeling Tools
in: G. Serazzi(ed.): Workload Characterization of Computer Systems & Computer Networks, North Holland, 1986

[CK1] L.A. Cherkasova, V.E. Kotov:
Structured Nets
in: Proceedings MFCS'81, Springer LNCS 118, 1981

[DAC] DACAPO III System User Manual
DOSIS GmbH, Dortmund, 1987

[GA1] D.D. Gajski :
The Structure of a Silicon Compiler
in: Proceedings IEEE ICCD, 1987

[GH1] J. Guttag, J.J. Horning :
The Algebraic Specification of Abstract Data Types
Acta Informatica, 10, 1978

[EM1] H. Ehrig, B. Mahr :
Fundamentals of Algebraic Specification
in: EATCS Monographs on Theoretical Computer Science, Vol. 6, Springer, 1985

[HE1] P. Hennige:
Generierung hierarchischer Steuerwerke anhand von Spezifikationen durch modifizierte Statecharts
Diplomarbeit, Univ. Paderborn, FB 17, 1990

[HO1] C.A.R. Hoare :
Communicating Sequential Processes
Prentice-Hall, 1985

[LK1] K.J. Lieberherr, S.E. Knudsen :
ZEUS: A Hardware Description Language on VLSI
in: Proceedings 20th DAC,1983

[LR1] K.-D. Lewke, F.J. Rammig :
Description and Simulation of MOS Devices in Register Transfer Languages
in: Proceedings IFIP VLSI'83, North Holland, 1983

[MR1] W. Müller, F.J. Rammig :
ODICE: Object-Oriented Hardware Description in CAD Environment
in: Proceedings CHDL'89, North Holland, 1989

[OD1] E. Odijk, W. Bronnenberg :
Parallel Computing: the Object-Oriented Approach
in: Proceedings CONPAR 88, 1988

[PE1] C.A. Petri :
Kommunikation mit Automaten
Schriften des Rheinisch Westfälischen Instituts für instrumentelle Mathematik, Bonn, 1962

[PE2] J.L. Peterson :
Petri Nets
in: ACM Surveys, 1977

[RA1] F.J. Rammig :
Systematischer Entwurf digitaler Systeme
Teubner, 1989

[SA1] J. Smit et al.:
Definition of the Syntax an Semantics of the Modelling and Design Language MoDL
in: Dewilde (ed.): The integrated Circuit Design Book, Delft University Press, Delft, 1986