LOGIC DESIGN AND SIMULATION, E. Hörbst (Editor)
© Elsevier Science Publishers B.V. (North-Holland), 1986

4 MIXED LEVEL MODELLING AND SIMULATION OF VLSI SYSTEMS

Franz J. Rammig

University of Paderborn
D-4790 Paderborn
Federal Republic of Germany

In this chapter various levels of abstraction for the description of VLSI systems will be discussed. Based upon this discussion basic external modelling concepts are identified. These concepts are mapped on only two internal ones. Finally a unified simulation technique serves to implement these two internal modelling concepts. The principles of this chapter are illustrated using the broadband hardware description language CAP/DSDL.

1. Levels of Abstraction

VLSI systems being highly complex systems it is necessary to describe them in a hierarchical manner on various levels of abstraction. It should be noted that hierarchy and various levels of abstraction are orthogonal concepts in this context. A hierarchical description may cover various levels of abstraction while within one level of abstraction a hierarchical description may be provided. A hierarchical description is constituted by recursively applying decomposition while a certain level of abstraction is identified by a specific conceptual view. It is not surprising that there does not exist a standardized system of abstraction levels. In the following such a system that seems to reflect some kind of common understanding is introduced.

Level 7: System Level

The basic view at this level is that of cooperating semiautonomous modules. Each of these modules are interpreted as a processor in a wider sense, i. e. an object that is capable to execute instructions on demand. Usual processors as well as DMA channels, memory systems, busses, peripheral controllers may serve as examples. From a more theoretical point of view these modules are modelled as Abstract Data Types (ADT). These are objects consisting of an internal carrier structure (invisible from the outside) and a set of allowed operations on this carrier. The implementation of these operations is invisible to the outside world.

So the system level may also be interpreted as a view where we have cooperating Abstract Data Types. The cooperation structure may be given in two different ways:

Either there is a hierarchically superior global communication control or control is distributed among the modules. In the first case the global control requests the individual modules to perform operations in an order that is defined by a concurrent algorithm. In the second alternative the concurrently operating modules are requesting operations mutually from each other.

In any case the timing model is reduced to a causality structure in most cases.

Level 6: **Algorithmic level**

Each module at the system level being a processor it is necessary to describe its interpretation algorithm for the instruction set. Typically this algorithm is a highly concurrent one. So it may be given using extended algorithmic constructs or by any appropriate notation, e. g. using interpreted Petri Nets. Included in this algorithm is not only the control structure but also the performed data manipulation. This is usually described by having operational modules of the Register Transfer level in mind, e. g. registers, busses, combinational logic etc..

The algorithmic level is also called Microprogramming level.

As timing model either a causality structure or a discrete time set (counting of clock signals) is used.

Level 5: **Register Transfer Level**

The RT-level is obtained from the algorithmic level by a certain kind of inversion. The imperative point of view (an algorithm is imperative by nature) is replaced by a reactive one. I. e. the system is composed of a (unordered) set of primitive objects that perform an action (usually a register transfer) whenever a certain condition becomes true. Included in the action performed is a modification of the global condition space. By this other objects may become active.

It should be noted that the algorithm to be implemented is still present (of course) but no longer explicitly visible.

The primitive modules are registers, memories, combinational logic, busses. A distinction is made between variables with storing capability (registers) and such without (terminals).

The timing model is in most cases a discrete time set (counting of clock signals). But real time is handled in some cases, too. By this also the asynchronous behaviour can be described.

Level 4: **Gate level**

In order to be implemented the modules of the RT level have to be expanded to gate circuits. By this deabstracting procedure the information about what are control signals and what are data signals is completely lost. Gate level descriptions are purely structural ones. The behaviour is hidden. It can be constructed from the (known) behaviour of the primitive objects (gates) and the interconncection structure. This reconstruction of the behaviour is exactly what is carried out by a gate level simulator. (The gate level simulator only shows the input/output behaviour, the symbolic behaviour, i. e. the algorithm has to be constructed by analytical methods.) For a pure gate level only logic gates are allowed as primitive objects. Usually more general "macro gates" are included in gate level discriptions. So what really constitutes this level is the point of view of defining a system by the interconnection structure only. The timing model is usually given by a continuous time set as the exact time behaviour is of interest at this level.

***Level 3*: Switch level**

The basic view at the Switch level is the same as at the gate level. The Switch level is obtained by further expanding the logic gates to circuits of transistors where the transistors are viewed at as discrete (ideal) switches. As the implementation of gates usually makes use of different signal strengths (ratio logic), bidirectional transmission gates and capacitances for data storage, these features must be describable at this level. The main difference to the gate level is introduced by the bidirectional transmission gate as gate level descriptions are unidirectional by nature.

Various models for this level of abstraction have been proposed [3, 6, 8, 12]. The timing model ranges from unit delay assumptions [3, 6] to a continuous time set [12].

***Level 2*: Symbolic Layout**

Concerning behaviour (and that is what we are interested in within the scope of this paper) this level is the same as the switch level. The information added with respect to the switch level is of structural nature: The relative position of the devices and the layer of interconnection. As no information is known about dimensions and about the fabrication process, the analogue behaviour is still unknown.

***Level 1*: Layout**

Again concerning behaviour this level does not constitute an own nature. From this point of view it is equivalent to the electrical level. The layout level abstracts from the process information but a certain layout has to be constructed with a fixed process in mind. Therefore a well defined analogue behaviour can be attached to a description at the layout level.

***Level 0*: Electrical Level**

At this level of abstraction the detailed analogue behaviour of the circuit is described. Usually this is done by a system of differential equations. There exist precise models of the devices that can be parametrized by a couple of parameters. Primitive objects now are capacitors, resistors, inductors. Of course a continuous time set is used at this level.

Nr.	Level	Model	primitive elements	observable values	timing model
7	System	net of processors	abstract data types	type dependent	causality
6	Algorithmic	concurrent algorithm	algorithmic constructs registers, logic, busses	bit string with interpretation	causality or discrete time set
5	Register Transfer	conditional register transfers	registers, logic, busses	bit strings	discrete time set
4	Gate	Boolean equations	gates	bits	continuous time set
3	Switch	Multivalued equations	switches, capacitors, resistors	discrete, multi-valued "bits"	continuous time set or discrete one
2	Symbolic Layout	↑	↑	↑	↑
1	Layout	↓	↓	↓	↓
0	Electrical	differential equations	capacitors, resistors, inductors	bounded real values	continuous time set

***Table 1**: Levels of abstraction*

Various languages exist for the description of digital systems at the different levels of abstraction. In most cases they are dedicated to a single level. So languages like SIMULA [2] or OCCAM [13] may be used for the system level. ISPS [1] is a typical language dedicated to the algorithmic level while CDL [4] or DDL [5] are well known as languages for the RT level. For the levels below the language aspect is of minor interest. So in most cases only the simulation system is mentioned which of course has a special description language. TEGAS [20] may serve as an example at the gate level, MOSSIM [3] for the switch level and SPICE [21] for the electrical level.

2. An Introduction to CAP/DSDL

During the design process various descriptions at the different levels of abstraction have to be produced (at least if the design process is carried out in a systematic top down manner). For this purpose three main approaches may be followed up:

- The dedicated languages approach.

  This is the classical solution. It has the advantage that relatively simple languages can be used that are easy to learn, easy to understand and can be processed efficiently.

  On the other hand this is a completely chaotic approach with a lot of friction losses at the interfaces between the languages. In practice dedicated languages are used only within design processes with a limited scope.

- The language family approach.

  A possible idea to overcome the problems of the dedicated languages approach is to construct a family of languages. Each language again is dedicated to a certain level of abstraction but the members of the family are interrelated with respect to syntax and semantics. By this solution not so heavy friction losses are to be expected. But they are still present. On the other hand the individual languages tend to become a little bit more complicated than in the pure dedicated languages approach. CONLAN [16] is a typical example of this idea.

- The broadband language approach.

  Using a broadband language is just the opposite to a set of unrelated dedicated languages. As the designer has to work only with one language it is very easy to describe his design at various levels of abstraction and to transform descriptions to such ones at lower levels of abstraction. By the same reason it is much easier to produce mixed level descriptions and to process (e. g. simulate) them. On the other hand of course there is the danger that a language dinosaur is made. To avoid this problem the modelling concepts have to be investigated carefully and reduced to a minimal set of necessary concepts. This approach will be demonstrated using the broadband language CAP/DSDL [19].

CAP/DSDL stands for "Concurrent Algorithmic Programming Language/Digital Systems Description Language". It has been designed to cover the bandwidth between System Level and Switch Level. Since 1981 it is very sucessfully in use at Siemens AG, Munich and a couple of German universities. Various design projects have been supported by this system.

CAP/DSDL has been designed to be as compatible to PASCAL [10] as possible. As (at least in Europe) every younger hardware designer is familiar with PASCAL it is rather simple for him to learn and to use CAP/DSDL. In addition an integrated hardware/software design process is facilitated. By the broadband nature of the language the user is not fixed in a "bed of Procrustes" - like manner to a certain style of design and description. An important feature included in the language is that of Abstract Data Types. So it is easy for the designer to create the basic objects that are useful for his problem, adapt to the changes of technology and to design in a highly modular way. In fact CAP/DSDL also includes some ideas from MODULA 2 [22]. CAP/DSDL allows to describe asynchronous systems as easily as synchronous ones. Arbitrary mixtures are allowed as well. The timing can be described more exactly as in most dedicated Gate Level languages. Finally CAP/DSDL is based upon very few semantical concepts. So it is a relatively slim language though being rather universal.

2.1 Objects in CAP/DSDL

Like PASCAL in CAP/DSDL a distinction is made between (object-) types and variables (object instantiations). Unlike PASCAL (but similar to SIMULA [2], MODULA 2, ADA [9]) in CAP/DSDL also types of complex objects with own control structure (type *procedure* or *function*) are possible. E. g. if one wants to describe a homogeneous multiprocessor system the single processor template is given by a type definition while the instantiations of this types, i. e. the set of processors in the system are created by declaration of variables of this type:

```
procedure multi-processor
    .
    .
    .
type CPU = procedure CPU (....);
        {procedure body}
    .
    .
    .
var CPU1, CPU2, CPU3, CPU4: CPU
```

In this context a special role is played by "export procedures". They allow the implementation of Abstract Data Types (ADT). They are procedures having a local data structure serving as carrier structure for the ADT to be implemented and the specification of operations allowed on the structure to be defined. These specifications are given using procedures and functions that are "exported".

```
export (operation_1, ..., operation_n) procedure xy
   {declaration of carrier-structure, may also be an ADT}

   procedure operation_1 (....);
     {implementation of operation_1}

   procedure operation_n (....);
     {implementation of operation_n}
end
```

Any procedure, function or export procedure may be a generic object. The following example explains this concept:

(Implemented ADT "stack", implemented on array, generic with attributes type of "stackelement" and "size" of stack).

```
type stack =
   export (flush, pop, push) procedure stack
         [stackelement:type; size:const];
      var stackcarrier:array [0:size] of stackelement;
      procedure flush;
         {procedure body}
      procedure pop (out data:stackelement; out empty:bit);
         {procedure body}
      procedure push (in data:stackelement; out full:bit);
         {procedure body}
   end;
.
.
.

var largebytestack: stack [bit (8), 10 000];
   smallarraystack: stack [array [0:16] of bit (32), 64];
```

Any CAP-procedure includes a monitor mechanism that ensures that concurrent references are scheduled in a certain manner so that the procedure is active only once at every point of time. I. e. a piece of hardware is described that exists once but may be referenced concurrently serving only one request per point of time (time shared resources).

The basic data object offered by CAP/DSDL is the bitstring of arbitrary length: **bit** (n). By certain operators it is interpreted as unsigned integer or as integer in two's complement representation. So arithmetic of arbitrary length is describable in CAP/DSDL as well. Special elementary objects are offered for interrupt signals and for places of Petri Nets. Out of elementary object composite ones may be constructed using the contructors known from PASCAL: **array** of arbitrary dimension and **record** of arbitrary depth. In order to allow different interpretations of a common physical object (a frequently used technique in hardware design) a PL/I - like overlay mechanism is offered.

2.2 Assertions

The analysis of large protocols of simulation results is a cumbersome task. In order to liberate the designer a little bit from this clerical work, in CAP/DSDL it is possible to formulate error-conditions that are not allowed to occure during the entire simulation run. So, if the user knows what he is looking for in the simulation protocol he can formulate this and delegate this analysis to the simulator. The designer also can formulate what has to happen as reaction to a violation of such an assertion. Usually just an error message is produced.

Example:

```
assertions
   clear & preset = > error ('illegal input values at module XYZ');
   not (write & (uptime (write) - uptime (data_ready) > = setup ))
    = > error ('setup-condition between write and data_ready violated');
   accu (16) = "1" = > stop (overflowcheck, 'overflow at register accu')
```

The last condition says that not only an error message has to be produced but also the simulation has to be stopped, provided that the switch "overflowcheck" has been enabled by the simulator-environment.

2.3 Datamanipulation in CAP/DSDL

The most elementary datamanipulation is the assignment of an expression's value to one or more variables or subparts/concatenations of them. CAP/DSDL as a hardware description language offers a rich set of logical operators together with numerous built in functions. Other operators are offered for arithmetic, permutations (shift/rotate), string manipulation and relation. Entire structured data objects may serve as source or destination of assignments as well. Unlike PASCAL but similar to PL/I also restricted expressions are allowed as assignment targets.

Example:

Assume the following declarations:

```
var
   register_bank_1, register_bank_2: array [0 : 15] of bit (32);
   mbr : record
      instr : bit (8);
      adr : bit (24);
   memory: array [0 : "(4) EFFFFF"] of bit (32);
               {nondecimal constant hexa digit following}

   mar : bit (24);
   indirect_bit : bit ;
```

With these declarations the following assignments are allowed:

Array to array assignment:

```
register_bank_1 : = register_bank_2;
```

Conditional assignment of array-element to entire record:

```
mbr : = if indirect_bit then memory [ memory [ mar] ]
                        else memory [mar];
```

Assignment of record component to concatenation of substrings of record components:

```
register_bank_1[0].(15:0) || register_bank_1[1].(7:0) : = mbr.adr;
```

2.4 Control Structures

Usually hardware behaves in a highly concurrent manner. Therefore CAP/DSDL allows to describe concurrent processes very easily (and of course by this also sequential ones). A distinction is made between structured control structures and general ones. The first ones are preferable as long as possible. It has been shown in [18] that they are suffiently general. In certain tricky situations however the use of general control structures (Petri Nets in our case) is more adequate.

2.5 Structured Control Structures

Structured control structures reflect structured hardware. So they are to be prefered whenever possible. They are made by nesting statements of the following kind:

- empty statement
- assignment statement

- compound statement
- while - statement
- repeat - statement
- for - statement
- case - statement
- if - statement

Any statement can be prefixed by an **at**-clause or a **when**-clause in order to describe clocked systems.

By the compound statement it is specified whether the included statements have to be executed sequentially, concurrently (i.e. not synchronized a priory) or in parallel (synchronized a priory)

Example:

Sequential execution:

```
seqbegin
      A : = B and C;
      B : = shl (B, 10)
end
```

Concurrent execution of a sequence of two assignments and a single one:

```
conbegin
     seqbegin
          A : = B and C;
          B : = shl (B,10)
     end;
     F : = G
end
```

Synchronized parallel execution, no data conflict:

```
parbegin
      A : = B;
      B : = A
end
```

The **while**-statement and the **repeat**-statement are used to describe cyclic activities. They have the same meaning as in PASCAL.

Example:

```
while not stop_bit do
    seqbegin
        instruction_fetch;
        operand-fetch;
        execute
    end
repeat countup until overflow
```

In contrary to PASCAL the **for** - statement describes a loop only if a sequential sweep over the index values is specified. Otherwise a shorthand notation for a replication is meant.

Example:

Roundshift on a buffer array, 10 assignments performed in parallel:

```
for I : = 0 parto 9 do
     buffer [ (I + 1) mod 10] : = buffer [i]
```

The **case**-statement and the **if**-statement are used to describe alternative control flows. These statements have the same meaning as in PASCAL.

Example:

Control by FSM:

```
case state of
      0: case input of
            0: state : = 1;
            1: parbegin
                  state : = 0;
                  output : = 1
                 end;
      1:  ...
            .
            .
            .
end
```

All the above mentioned statements serve to describe logical control structure. In the case of an asynchronous implementation this may directly reflect the realization. In most cases, however, the physical realization is carried out using a clocking scheme. In order to specify this the **at**-prefix and the **when**-prefix is offered by CAP/DSDL.

A statement prefixed with **at**-condition is executed if it has to be executed due to the control flow it is embedded in **and** (after this) the specified condition has become true. The **at**-prefix describes the edge trigged approach. The **when**-prefix serves for the same purpose in level oriented techniques.

Example: (pipelined instruction cycle, synchronized by raising edge of clocksignal mainclock)

```
conbegin
   at up (mainclock) do instruction_fetch;
   at up (mainclock) do operand_fetch;
   at up (mainclock) do execution
end
```

2.6 General Control Structrures

In CAP/DSDL it is also possible to specify control structures directly via a (modified) Petri net. In addition structured control specifications may be embedded.

The places of the Petri net are represented by special variables of type **place** while the transitions are described using constructs of the form **on**_("input places") **do mark** ("output places") as prefix of arbitrary CAP/DSDL statements. Such a statement is executed whenever the attached transition fires.

Example: (same as above, but without clocking)

```
var start, if, of, ex, ifdone, ofdone, exdone :place;
on (start) do mark (if & of & ex);
on (if) do mark (ifdone) instruction_fetch;
on (of) do mark (ofdone) operand_fetch;
on (ex) do mark (exdone) execution;
on (exdone & ifdone & ofdone) do mark (start);
```

2.7 Data driven Control

CAP/DSDL is not restricted to a procedural point of view. It is also possible to describe systems in a nonprocedural manner, i. e. without an explicitly given control structure. This is done using "implicit variables", assignments to such variables and "implicit control activations".

Implicit variables model pieces of hardware without storing capabiliity. They are single assignment variables. The value of such a variable is defined by a single expression being assigned to. This expression is evaluated continuously (e.g. output of a combinational circuit). Optionally the assignment can be conditioned, i.e. carried out as long as a certain condition is true. If not, the value assigned last is maintained (e.g. D-latch).

Example: (description of an asymetric clock generator and a D-latch)

```
var clock : implicit bit; D, Q : bit;
impdef
   clock : = not clock delay (up 10, down 20);
   when clock do Q : = D;
```

Implicit control activations are CAP/DSDL statements with an **at**-prefix embedded in the impdef part. Such a statement is initiated whenever the condition of the prefix becomes true.

Example:

```
impdef
   at down (clock) or change (ext_signal) do action_1;
   at up (clock) do action_1;
```

It should be noted that the relative ordering of the statements within an impdef part has no semantical meaning.

2.8 Modularization

CAP/DSDL is a blockoriented language like PASCAL in the sense that there are nested namespaces, organized as in PASCAL. On the other hand, of course, all objects are static ones that exist and keep their values independently from the activation of the block they are embedded in. Blocks are constituted by procedures, functions and export procedures. Each block may contain own resources which are not accessable from the outside. In addition a block has access to every object of the block it is embedded in, provided it has no object of the same name. External blocks have no access to global objects.

A block describes a once existing resource (piece of hardware) that may contain an own control. This piece of hardware may be requested concurrently from different sides. Therefore an arbitration mechanism is included.

The interface between a block and its static environment is given by access to global resources. The interface to the dynamic environment is described by a pair (formal parameter list, actual parameter list). There is a distinction between input-, output- and bidirectional signals. Their correct usage is checked by the compiler. Checked too is the type compatibility of the actual and formal parameters.

Example:

```
function abs (in something: bit (128)) : bit (128);
    begin
        abs : = if something > = 0 then something
                                   else something
    end;

procedure rsflipflop (in R,S : bit : out Q.NQ : bit);
    var state : bit;
        x : bit (32);
    assertions R & S  = > error ('bloody misuser!')
    seqbegin
        Q, state : = case R || S of
                          0 : state;
                          1 : "1";
                          2 : "0";
                          3 : (randint (0, 1, x)).(0)
                          end;
        NQ : = not Q
    end
```

Procedures, functions and export procedures can also be declared as types. Objects of such a type can be instantiated using usual **var** declarations. In this case also generic objects are allowed.

Example: (abs-function for arbitrary data types)

```
type abs = function abs [t : type]
                (in something : t) : t;
      begin
          abs : = if something > = 0 then something
                                     else - something
      end;
    .
    .
    .

var long_abs : abs [ bit (7938) ]
      array_abs : abs [ array [32:1, 15:0] of bit (16)];
```

2.9 Interrupts

The concurrent control structure of CAP/DSDL may be superimposed by an additional level.

Interrupt signals are special objects with domain {set, not set}. They can be set either internally by special functions or externally. Interrupt signals correspond to interrupt service routines (ISR) that can be declared in every block. If an ISR is activated by its corresponding interrupt signal only the block it is declared in is interrupted. The remaining part of the entire description remains unaffected, i. e. proceeds normally. The interrupted block is resumed after the termination of the ISR.

Example: (Skeleton of a handshaking protocol between concurrently active procedures module_1 and module_2)

```
procedure int_demo
   var interseq, intack : interrupt (1)

   procedure module_1;              priority
   interrupts
      on interrupt (intseq) do begin sint (intack) end
        .
        .
        .

   conbegin module_1; module_2 end
```

2.10 Timing

Empty statements and assignments may be delayed. In the case of an empty statement this means that the initiation of a sequentially following statement is delayed. In the case of an assignment statement the arguments are evaluated immediately while the target variables get the value to be assigned after the specified delay. This delay specification may contain arbitrary expressions in order to describe state dependent delay. Additionally intervals of uncertainty may be specified. In the case of **bit** (1) assignments different rise/fall times may be given.

Example:

```
output : = arg1 & arg 2 delay (20);
A : = not A  delay (up 20, down 15 to 17);
sum : = arg1  +  arg2 delay (if arg 2 > 15 then 20 to 25,
                                        else 10 to 12);
```

3. External Modelling Concepts

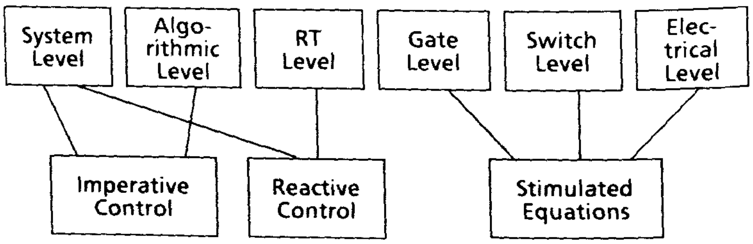
As explained above, descriptions at the system level usually are given by a set of abstract data types, their instantiations and a communication structure between them. This communication between the components of a system (i.e. the instantiated ADTs) may be specified either using a global algorithm or in a distributed way. This reflects basic architectural concepts.

In the case of CAP/DSDL we observe these two different approaches either as the existence of a global, concurrent algorithm or an interrupt structure. It can be observed that both the imperative point of view and the reactive one are present at the system level.

The algorithmic level is uniquely characterized by a single modelling concept: The imperative point of view. Even if we have (as usual in hardware descriptions) a highly concurrent control structure there is still the single central control that governs the entire system.

At the register transfer level we have a single concept, too. But now the imperative point of view is completely inverted to a reactive one. The system now is described by passive components that react only on demand, i.e. perform a certain action whenever a specific condition is true. A third modelling concept is present at the gate level. Here we have an unordered set of equations. These equations may be disturbed from their equilibrium by external (or internal) events. After being disturbed they try to restabilize in equilibrium. So we have the concept of stimulated equations at this level. The same model is present at the lower levels. Only the domain of the variables is enriched. At the switch level a finite set of classes of conductance and capacitance is used as domain for variables while at the electrical level this finite set is expanded to an infinite one.

To sum up: We observed that the six different (concerning behaviour) levels of abstraction are covered by only three external modelling concepts. This is illustrated in the following figure



4. Internal Modelling Concepts

4.1 Petri Nets

The above discussion showed that concurrent algorithms are used at the system level (in the case of a centralized global control) and at the algorithmic level. Furthermore it can be observed that there are various notations. This entire variation however can be reduced to one single concept:

Timed Interrupted Petri Nets

Definition 3.1

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

Definition 3.2

$PN = (PG, m_0, R)$ is called ***Petri Net***: $<=>$
$PG = (P, T, E)$ Petri Net Graph
$m_0 \in M = \{m \mid m: P \rightarrow \mathbb{N}_0\}$ (initial marking)
$R \in \{r \mid r: T \rightarrow f_T\}$ with $\forall t \in T$: $(f_t : M \rightarrow M)$ (firing rule of t).

Places are used to model conditions. If a place contains a token the associated condition is assumed to be true. Actions are modelled by transitions. A transition is firable if a certain condition on its input places is true (e.g. all input places marked). By firing it manipulates the marking of its input places and output places (e.g. demarks all input places and marks every output place). By technical reasons we use a heterogeneous set of firing rules, but this is not essential.

Definition 3.3

$IPN = (PN, I, D)$ is called ***Interpreted Petri Net***: $<=>$
$PN = ((P, T, E)\ m_0, R)$ Petri Net
$I \in \{i \mid i: T \rightarrow \sigma \cup \{\lambda\}\}$ with $\sigma = \{o \mid o: \mathrm{dom}(o) \subset XD \rightarrow \mathrm{codom}(o) \subset XD\}$
D many-sorted set (of "data objects").

Interpreted Petri Nets are obtained by attaching a datamanipulation $o(t)$ to transition t. Whenever such a transition fires its attached operation is performed. We call this an ***interpreted firing***.

Definition 4

$TIPN = (IPN, \nabla)$ is called ***Timed Interpreted Petri Net***: $<=>$
$IPN = (((P,T,E),m_0,R),I,D)$ Interpreted Petri Net
$\nabla \in \{\delta \mid \delta : T \to \tau\}$ with $\tau = \{0' \mid 0' : \mathrm{dom}(0') \subset XD \to \mathbb{R}\}$

A ***timed interpreted firing*** of a transition is defined as follows:

Assume transition t becomes firable at time point t_0. At this time point the attached operation (if existent) $i(t) = o$ is initiated. That means the values of dom(o) at this time point are evaluated. At the same time point the delay function $\delta(t) = 0'$ is evaluated based on the values of dom(0') at time point t_0. The result of 0' may be K. Then at time point $t_0 + K$ the values calculated by o are stored in codom(o) and the firing (i.e. the token game) takes place.

CAP-nets are Timed Interpreted Petri Nets with a heterogeneous set of firing rules. In the following we will denote the set of input places of a transition t by ${}^\bullet t = \{p \in P \mid (p,t) \in E\}$. Similarly we define $t^\bullet = \{p \in P \mid (t,p) \in E\}$ and for places p, ${}^\bullet p = \{t \mid (t,p) \in E\}$; $p^\bullet = \{t \mid (p,t) \in E\}$.

Definition 5

Let $PN = ((P,T,E),m_0,R)$ be a Petri Net, $t \in T$.

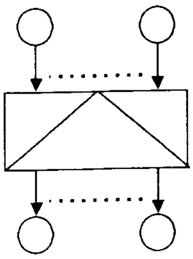
Transition t is called an **AND-*transition*** : $<=>$
1) t is firable under marking m: $<=>$ $\forall p \in {}^\bullet t$: $m(p) > 0$
2) $f_t : M \to M$ is called firing of t : $<=>$

$$
\begin{aligned}
f_t(m(p)) &= m(p) - 1 && \text{iff } p \in {}^\bullet t \wedge t \text{ firable}\\
&= m(p) + 1 && \text{iff } p \in t^\bullet \wedge t \text{ firable}\\
&= m(p) && \text{else.}
\end{aligned}
$$

Let $\boldsymbol{A(PN)} \subset T$ denote the set of all AND-transitions of a Petri Net PN.

The AND-transition is the single type of transition commonly used in Petri Nets.

Symbol



Definition 6

Let be $PN = ((P,T,E),m_0,R)$ a Petri Net, $t \in T$.

Transition t is called an **OR-*transition*** : < = >

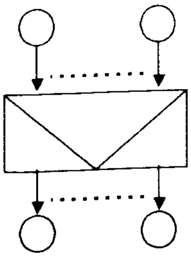
1) t is firable under marking m: < = > $\exists p \in {}^{\bullet}t$: $m(p)>0$

2) $f_t : M \rightarrow M$ is called firing of t : < = >

$$\begin{aligned} f_t(m(p)) &= m(p) - 1 && \text{iff } p \in {}^{\bullet}t \wedge m(p)>0 \wedge t \text{ firable} \\ &= m(p) + 1 && \text{iff } p \in t^{\bullet} \wedge t \text{ firable} \\ &= m(p) && \text{else} \end{aligned}$$

Let $\boldsymbol{O(PN)} \subset T$ denote the set of all OR-transitions of a Petri Net PN.

Symbol



Definition 7

Let $IPN = (((P,T,E),m_0,R),I,D)$ be an Interpreted Petri Net, $t \in T$, ${}^\bullet t = \{p_i\}$, $t^\bullet = \{p_{false}, p_{true}\}$, $i(t) = d \to d$, $i(t)(d) = d$, $value(d) \in \{true, false\}$.

Transition t is called a **DECIDER-*transition*** : < = >

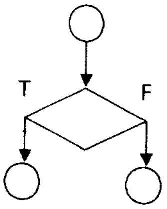
1) t is activated under marking m: < = > $m(p_i) > 0$

2) $f_t : M \to M$ is called firing of t : < = >

$f_t(m(p_i))$	$= m(p_i) - 1$	iff t firable
$f_t(m(p_{false}))$	$= m(p_{false}) + 1$	iff t firable $\wedge$ d = false
$f_t(m(p_{true}))$	$= m(p_{true}) + 1$	iff t firable $\wedge$ d = true
$f_t(m(p)$	$= m(p)$	else

Let $\boldsymbol{D(IPN)} \subset T$ denote the set of all DECIDER-transitions of an Interpreted Petri Net *IPN*.

Symbol



Of course this definition can be generalized to an n-branch decider in a straight forward manner.

Definition 8

Let $IPN = (((P,T,E),m_0,R),I,D)$ be an Interpreted Petri Net, $t \in T$, ${}^\bullet t = \{enable, req_i \mid i = 0{:}n\}$, $t^\bullet = \{run, ret_i \mid i = 0{:}n\}$, $i(t) = d \to d$, $i(t)(d) = d$, $value(d) \in \{\{req_i \mid i = 0{:}n\} \to [0,n]\}$.

Translation t is called **BLKHD-*transition*** : < = >

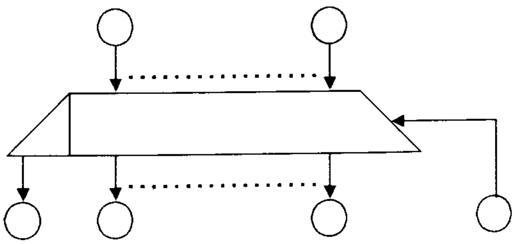
1) t is firable under marking m: < = > $m(enable) > 0 \wedge \exists p \in \{req_i \mid i = 0{:}n\}$: $m(p) > 0$

2) $f_t : M \to M$ is called firing of t : < = >

$f_t(m(enable))$	$= m(enable) - 1$	iff t firable
$f_t(m(run))$	$= m(run) + 1$	iff t firable
$f_t(m(req_i))$	$= m(req_i) - 1$	iff t firable
$f_t(m(ret_i))$	$= m(ret_i) + 1$	iff $value(d)(req_i) = max\{(value(d)(req_j) \mid m(req_j) > 0\}$

Let $\boldsymbol{B(IPN)} \subset T$ denote the set of all BLKHD-transitions of an Interpreted Petri Net *IPN*.

Symbol



It should be noted that by definition a BLKHD-transition includes an arbitration mechanism between input places req_i, $i = 0{:}n$.

Definition 9

Let $IPN = (((P,T,E),m_0,R),I,D)$ be an Interpreted Petri Net, $t \in T$, $^\bullet t = \{finished, ret_i \mid i = 0{:}n\}$, $t^\bullet = \{enable, back_i \mid i = 0{:}n\}$.

Transition t is called **BLKEND-*transition*** $:<=>$

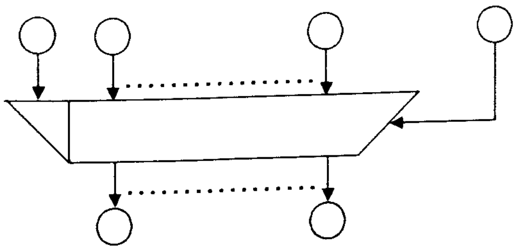
1) t is firable under marking m: $<=>$ $m(finished) > 0 \wedge \exists p \in \{ret_i \mid i = 0{:}n\}$: $m(p) > 0$

2) $f_t : M \rightarrow M$ is called firing of t : $<=>$

$$\begin{array}{lll} f_t(m(finished)) & = m(finished) - 1 & \text{iff } t \text{ firable} \\ f_t(m(enable)) & = m(enable) + 1 & \text{iff } t \text{ firable} \\ \left.\begin{array}{l} f_t(m(ret_i)) \\ f_t(m(back_i)) \end{array}\right. & \left.\begin{array}{l} = m(ret_i) - 1 \\ = m(back_i) + 1 \end{array}\right\} & \text{iff } t \text{ firable} \wedge m(ret_i) > 0 \end{array}$$

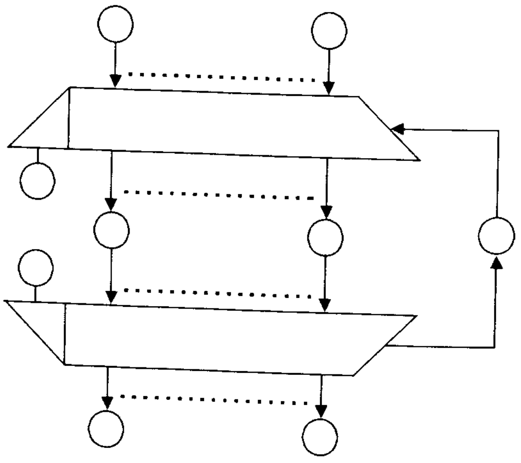
Let $\mathbf{N(IPN)} \subset T$ denote the set of all BLKEND-transitions of an Interpreted Petri Net *IPN*.

Symbol



BLKHD- and BLKEND-transitions are only allowed to be used pairwise. In such a pair the places called *ret*$_i$ and *enable* are identified. Between *run* and *finished* arbitrary nets with exactly one input place and one output place can be connected.

Symbol



Definition 10

Let $PN = ((P,T,E),m_0,R)$ be a Petri Net, $t \in T$, $^\bullet t = \{synch, ord\}$, $^\bullet t = \{out\}$.

Transition t is called **AT-*transition*** :< = >

1)t is firable under marking m:< = > $m(synch) > 0$

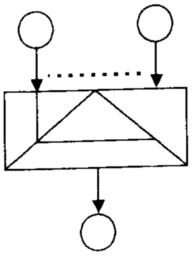
2) $f_t : M \rightarrow M$ is called firing of t : < = >

$$f_t(m(synch)) = m(synch) - 1 \quad \text{iff } t \text{ firable}$$

$$\left.\begin{aligned} f_t(m(ord)) &= m(ord) - 1 \\ f_t(m(out)) &= m(out) + 1 \end{aligned}\right\} \quad \text{iff } t \text{ firable} \wedge m(ord) > 0$$

Let ***Y(PN)***$\subset T$ denote the set of all AT-transitions of a Petri Net *PN*

Symbol



The AND-transition is the basic transition of Petri Nets. The OR-transition has been introduced in order to model backward conflicts by transitions while the DECIDER-transition models the foreward conflict. BLKHD/BLKEND pairs allow a hierarchical description of nets with encapsulated subnets while with the aid of the AT-transition a synchronization with a cyclic external event (typically the clock) can be described. Instead of a direct definition as used here, the semantics of the different types of transitions can also be given in terms of ordinary Petri Nets.

We restrict ourselves to a subclass of such nets:

Definition 11

Let $IPN = (((P,T,E),m_0,R),I,D)$ be an Interpreted Petri Net.

IPN is called ***SRCN*** (***Safe Restricted CAP Net***) $:<=>$

1) $T = A(IPN) \cup O(IPN) \cup D(IPN) \cup B(IPN) \cup N(IPN) \cup Y(IPN)$
2) $\forall p \in P{:}\ |{}^\bullet p| = |p^\bullet| = 1$
3) $\forall p \in P\ \forall m \in M{:}\ m(p) \in \{0,1\}$
4) $\forall t \in T{:}\ (t \notin D(IPN) \vee t \notin B(IPN) \vee t \notin A(IPN) \cap \{t \in T \mid |{}^\bullet t| = |t^\bullet| = 1\}) \Rightarrow i(t) = \lambda$

4.2 Algorithmic Constructs of CAP/DSDL Expressed by SRCNs

In order to demonstrate the power of SRCNs we shortly demonstrate how the algorithmic constructs of CAP/DSDL are transformed to SRCNs. Similarly comparable constructs of ADA (rendevous), SIMULA (coroutine) or OCCAM (CSP !/?) can be transformed.

Procedures and Functions

Let S be a <compound statement>, F a <formal parameter list>, T a <data type>.

The construct

procedure pname; ...S

is equivalent to

procedure pname **mark**(p_1); ...
net on(p_1) **do return** S **end**

The construct

procedure pname(*F*); ...S

is equivalent to

procedure pname **mark**(p_1)(*F*)
net on(p_1) **do return** S **end**

The construct

function pname: T; ...S

is equivalent to

function pname **mark**(p_1): T; ...
net on(p_1) **do return** S **end**

The construct

function pname(*F*):T; ...S

is equivalent to

function pname **mark**(p_1)(*F*): T; ...
net on(*p* sub 1) **do return** S **end**

Compound Statements

Let S_1 to S_n be a <statement>, M a construct of the form **return**, **nomark** or **mark**(<mark list>).

The construct

on(p_1) **do** M **seqbegin** S_1...S_n **end**

is equivalent to

on(p_1) **on mark**(p_2) S_1
on(p_2) **do mark**(p_3) S_2
.
.
.
on (p_n) **do** M S_n

The construct

on(p_1): M **conbegin** S_1...S_n **end**

is equivalent to

on(p_1) **do mark**(p_{2i} &...& p_{ni})
on(p_{2i}) **do mark**(p_{2t}) S_1
.
.
.
on(p_{ni}) **do mark**(p_{nt}) S_n
on(p_{2t} &...& p_{nt}) **do** M

Case Distincton

Let E be an <expression>, S_1 to S_n a <statement>, M of the form **return**, **nomark** or **mark**(<mark list>).

The construct

on(p_1) **do if** E **then** S_1 **else** S_2

is equivalent to

on(p_1) **do if** E **then mark**(p_{1t}) S_1 **else mark**(p_{1f}) S_2
on(p_{1t} | p_{1f}) **do** M

Loops and Replications

Let E be an <expression>, S a <statement>, M of form **return**, **nomark** or **mark**(<mark list>).

The construct

on(p_1) **do** M **while** E **do** S

is equivalent to

on(p_1|p_y) **do if** E **then mark**(p_y) S **else** M

The construct

on(p_1) do M repeat S until E

is equivalent to

on(p_1) do mark(p_s|p_y) do if E then mark(p_y) S else M

Guarded Commands

Having investigated a common concept for all algorithmic aspects we are now looking for such a unifying approach for the nonprocedural concepts as well. This approach is given in a natural way by guarded commands:

$$\{ (<condition> \; : \; <action>)_i \mid i = 1,\dots,n \}$$

with the semantics that $<action>_i$ has to be performed whenever $<condition>_i$ becomes true. The nonprocedural point of view is expressed by the set nature of this approach. By properly restricting the nature of the conditions this definition corresponds directly to our different kinds of guarded register transfers. Similarly the interrupt concept can be covered by the same technique (it can be covered by Petri Nets as well [7] so that we have two options in this case). Finally the stimulated equations point of view as used at the gate level and switch level fits into this framework as well. We just have to replace the *<condition>* by the constant condition "true".

4.3 Summary of Internal Modelling Concepts

We have to cover five levels of abstraction:

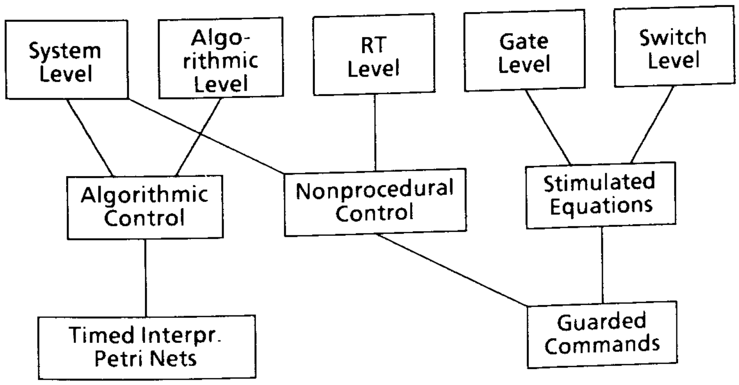
- System level
- Algorithmic level
- Register Transfer level
- Gate level
- Switch level

We observed three main concepts for modelling these levels:

- Algorithmic control (imperative control)
- Nonprocedural control (reactive control)
- Stimulated equations (hidden control)

These three external modelling concepts can be covered by two internal concepts:

- Timed Interpreted Petri Nets
- Guarded Commands



The mapping from external concepts to the internal one typically is perfomed by a compiler. So it is done in the case of CAP/DSDL.

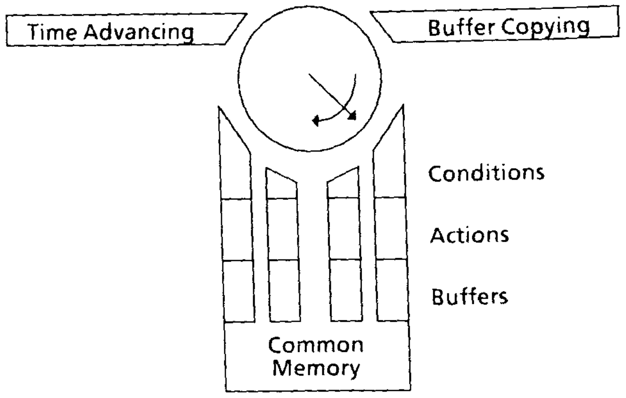
4.4 Simulation Techniques

The simulation algorithm has to map the internal modelling concept to the architecture of the host computer. This can be carried out either by interpretation ("table driven simulation") or by generating proper code ("compiled node simulation"). The basic concept is the same for both purposes. Similarily this basic concept is valid for different host architectures. The actual algorithms however are completely dependent on the actual host architecture. Here we will assume a sequential v. Neumann architecture. Based on this assumption two simulation techniques will be discussed: "Equitemporal Iteration" and "Critical Event Scheduling".

4.4.1 Equitemporal Iteration

This technique has a global point of view. The idea is that a sweep over the entire model of the system takes place iteratively. After each sweep the global time is increased by a step wich may vary from iteration to iteration but is always the same for all components visited.

Each component of the system to be simulated is modelled by a triplet (c, a, d). Component c stands for the executability condition attached to the component. In the case of simulating a Petri Net this may be part of the firing rule while in the case of Guarded Commands this may be the guard. Assume that c is true in an iteration. Then action a is performed. As a consequence some variables get new values. This assignment is not carried out directly to the target variables but to buffers in order to make sure that the components visited next during the actual sweep are not affected by this assignment. After the entire sweep all buffers are copied into the target variables stored in a common memory. Graphically this algorithm may be represented in the following way:



This simulation technique is very simple and easy to implement. Therefore it has been very popular for gate level simulation and is still popular at the RT-level. Unfortunately it is very inefficient in most cases. The reason is that typically at a certain point of time more than 95 percent of a circuit is stable. But this means that the probability that an operation other than the identity has to be performed for a certain component during a given iteration is less than 0.05.

4.4.2 Event Scheduling for Petri Nets

First of all we have to decide what are the events in a Petri Net. It seems to be the most natural approach to treat the firing of a transition as an event. As we have timed Petri Nets, i.e. timed firings we in fact have two subevents per event: The initation of the firing and the termination. CAP-nets are defined in such a way that the firing of a transition is initiated exactly at the time point where the firing conditions of this transition become true. At the same time point the attached data operation is initiated and based on values of this time point the proper delay is calculated. The data operation is terminated, i.e. the assignments take place, after the calculated delay time has elapsed. At the same time point the firing terminates, i.e. the token game according to the firing rule of the transition takes place. A closer view at this model shows that only the termination subevent has to be scheduled by the algorithm.

Definition 12

Let $IPN = (((P,T,E),m_0,R),I,D)$ be an Interpreted Petri Net; $t, t' \in T$.

The transition t is called ***predecessor*** of transition t' (and t' ***successor*** of t) : < = >
$t^\bullet \cap {}^\bullet t \neq \varnothing$

A predecessor of a transition is also called an ***influencer*** of this transition while a successor of a transition is also called an ***influence*** of this transition.

Corollary 1
The firing of a transition t is initiated at time point r = > the firing of at least one of its predecessors is terminated at the same time point r.

Proof
This is obvious as a transition can become firable only if the marking of its input places change its value. This can happen only due to the firing of at least one of its predecessors.

This corollary allows us to treat only the termination of a firing as an event. The initiation is treated as part of the execution of the firing event of that predecessor that caused the initiation. So the basic algorithm for event scheduling can be modified in the following way:

```
begin
   time : = 0;
while time ≦ final_time and queue ≠ empty do
   begin
      extract event $E_t$ with $t$ minimal from queue;
      assign values of data operations;
      modify marking due to firing rule;
      for all influencees do
          if firable then
             begin
                calculate hatching time for firing;
                initiate data operation;
                insert $E_i$ properly in queue
             end
   end
end
```

4.12 Critical Event Scheduling

We are looking for a simulation algorithm that skips all unneccessary computations. This algorithm has to be useful for both of our internal modelling concepts. By examining these models we observe:

• The time of the next occurrence of an event is predictable.
In the case of Petri Nets this is true as the firing time of a transition t is determined by the time of the activation of t and the associated delay which is known at activation time. In the case of guarded commands the variables of the guard are known and whenever an assignment to such a variable is initiated at this time point the associated delay is known.

• If the time of the next occurrence of an event is not predictable this event does not take place until it becomes predictable by the occurrence of other events.
If we look at Petri Nets a transition t becomes firable only if the marking of its input places change. This can happen only if another transition t' fires, i.e. another event takes place before. The same is true in the case of guarded commands. A guard can change its value only due to an assignment, i.e. the execution of another command.

• Between two succeeding events there happens nothing influencing the system.
This is true as the global marking is stable if no transition fires and all guards are stable if no assignments are carried out.

From these three statements the critical event simulation algorithm can be deduced directly. This technique is a local one in contrary to equitemporal iteration.

The following skeleton illustrates the algorithm:

```
begin
   time := 0;
while time ≤ final_time and queue ≠ empty do
   begin
      extract event $E_t$ with $t$ minimal from queue;
      execute event $E_t$;
      for all predictable events $E_1,\ldots,E_n$ influenced by $E_t$ do
         begin
         calculate hatching time of $E_i$;
         insert $E_i$ properly into queue
         end
   end
end
```

Obviously this algorithm reduces the number of components to be visited and the operation to be executed drastically. The price to be payed is the overhead of keeping the queue sorted. But this is neglectable as the queue typically holds less than 5 percent of all components of a model.

5. Event Scheduling Techniques for Internal Modelling Concepts

Event scheduling has been considered as the most powerful simulation technique. The above general discussion showed that this technique can be used for Petri Nets and Guarded Commands. However it has to be investigated which are the most adequate implementation methods for the two models.

It can be observed that a scan over all successors of a transition is included in the action to be carried out at a firing event of this transition. Fortunately in practice this is not so bad as typically

- a transition has only very few output places in most cases and
- a transition has only very few input places in most cases.

By the first oberservation only few transitions have to be visited by a scan. The second overservation makes the probability that an influenced transition has become firable due to the firing of the actual transition relatively high. In addition an uncertainty exists only in the case of visiting transition of type AND, BLKHD, or AT.

5.1 Event Scheduling for Guarded Command

Again the basic decision has to be made what is an event. The execution of a guarded command seems to be the most obvious selection. But again such an event is composed of two subevents: The initiation of the execution and its terminition. In the case of CAP/DSDL the operation may either be an assignment or an entire algorithm. In the case of an algorithm the initiation of the operation means that the attached algorithm has to be started. This means nothing else than marking the single input place of the Interpreted Petri Net that represents this algorihtm. And this also terminates the operation. Therefore we restrict ourselves to assignments in this section. In this case at the initiation time point the delay is calculated based on values at this time point. The operation to be carried out is based on values at this time point, too. The assignment of values, however, takes place after the calculated delay time. This assignment terminates the event.

As in the case of Petri Nets only the termination subevent has to be considered (scheduled).

Definition 13

Let $C_i(c_{i1},\dots,c_{in})$ be an expression with argument variables $c_{i1},\dots,c_{in}$; $A_i(d_{i1},\dots,d_{iK};a_{i1},\dots,a_{im})$ be an assignment with destination variables $d_{i1},\dots,d_{iK}$ and argument variables $a_{i1},\dots,a_{im}$
Let $G_1 = C_1 : A_1$ and $G_2 = C_2 : A_2$ be guarded commands.

G_1 called ***influencer*** of G_2 (and G_2 ***influencee*** of G_1) $:<=>$
At least one destination variable d_{1s} of A_1 is argument variable c_{2r} of C_2.

It should be noted, that the guarded commands G_1 and G_2 may be identical, i. e. a guarded command may be its own influencer (and influencee).

Corollary 2

A guarded command $G = C : A$ can be initiated only if at least one guarded command terminates at the same point of time.

Proof

This is obvious as for an initiation it is necessary that the value of the guard is changed. This can happen only if at least one of its variables gets a new value. And this can happen only due to an assignment carried out as part of an influencer's termination.

When stimulated equations are modelled by guarded commands the same is true. The variables in the guard just have to be replaced by the variables of the assignment. So the following modification of the basic event scheduling algorithm is obtained:

```
begin
   time := 0;
   while time $\leq$ final_time and queue $\neq$ empty do
      begin
         extract event $E_t$ with $t$ minimal from queue;
         assign values;
         for all influencees do
            if guard becomes true then
               begin
                  calculate hatching time for value assignment;
                  initiate data operation;
                  insert $E_i$ properly in queue
               end
      end
end
```

Here too we have a scanning over all influencees of a guarded command. Fortunately in this case too it can be observed that usually the fanout of a variable is relatively small.

Further improvements can be obtained by a selective trace mechanism that of course is included in the case of CAP/DSDL. Selective trace means that only new values that are different from the old ones are considered when scanning through the influencees.

6. Event Scheduling for Switch Level Simulation

Having reduced the bandwidth of hardware descriptions to two internal concepts and having investigated how these concepts can be handled by a common and efficient simulation algorithm, the major problems seem to be solved. Unfortunately the switch level causes some additional problems. The main reason is that the algorithm of event scheduling is local and unidirectional (influencer - influencee - relations!) by nature while switch level models are bidirectional.

So it is not surprising that the "classical" switch level approach (especially [3, 8, 11] but also [6]) has an equitemporal iteration in mind. This approach is bound to a unit-delay assumption (or zero-delay). Therefore both, model fidelity and algorithm efficiency may cause problems. Here we will discuss shortly a technique for switch level simulation by event scheduling. This technique has been developed for CAP/DSDL and published first in [12]. Later Kawai and Hayes proposed a similar approach [11].

All switch level models need an extension of the domain of variables (observation points) in order to describe different signal strength. As the basic principles remain the same independently from the cardinality of the domain we here restrict ourselves on a "minimal" set of three different signal strengths:

So the following set of values is obtained:

L0:	low impedance zero	(e.g. GND)
L1:	low impedance one	(e.g. V_{DD})
M0:	medium impedance zero	(e.g.through pulldown)
M1:	medium impedance one	(e.g. through pullup)
H0:	high impedance zero	(charging zero)
H1:	high impedance one	(charging one)
Z:	high impedance	

This domain allows to argue about ratio logic and circuits without charge sharing. If charge sharing occurs charing values at different strength levels are needed, of course. This can be included in the model without problems.

Following obvious arguments we can define a partial ordering on the set V = { L0, L1, M0, M1, H0, H1, Z }

In order to handle uncertain signals usually a third value is added at every signal strength. E.g., following [8], our domain (V, < =) would be transformed to

$$\begin{tikzcd}
 & LU \arrow[dl, dash] \arrow[dr, dash] & \\
LO \arrow[dr, dash] & & L1 \arrow[dl, dash] \\
 & MU \arrow[dl, dash] \arrow[dr, dash] & \\
MO \arrow[dr, dash] & & M1 \arrow[dl, dash] \\
 & HU \arrow[dl, dash] \arrow[dr, dash] & \\
HO \arrow[dr, dash] & & H1 \arrow[dl, dash] \\
 & Z &
\end{tikzcd}$$

This is a nice lattice but unfortunately causes a couple of problems. They originate from the fact that strength and (logical) value are two orthogonal aspects of a signal while uncertainty is introduced only for the aspect (logical) value.

Therefore it is preferable to represent uncertain values just by an enumeration of the possible values. So 2^V (the powerset of V) serves as the basic domain in the model. The behaviour of components has to be described as mapping $F: (2^V)^n \rightarrow (2^V)^m$ where for convenience we identify $\alpha \in V$ with $\{\alpha\} \in 2^V$.

Definition 14

Let $F: (2^V)^n \rightarrow (2^V)^m$; $A_1, \ldots, A_n \in 2^V$

F is called ***well defined*** : $<=>$

(i) $F(A_1, \ldots, A_n) := \cup \{F(a_1, \ldots, a_n) \mid a_i \in A_i\}$

(ii) $F(A_1, \ldots, A_n) = \emptyset \;<=>\; \exists\, 1 \leq i \leq n \;\; A_i = \emptyset$

By (i) all functions can be defined on singletons while by (ii) it is possible to exclude the empty set. The behaviour of devices is defined using two basic functions:

Definition 15

$\forall a,b \in V$: sup(a,b): = if $a \leqq b$ then b
else if $b \leqq a$ then a
else $\{a, b\}$

$\forall A,B \in 2^V$: sup(A,B): = $\cup$ sup { (a,b) | $a \in A \wedge b \in B$ }

The function sup(A,B) describes the value of a region that is connected with two signals A and B.

Definition 16

$\forall a \in V$: $\downarrow(a) :=$ if $a \in \{H1, M1, L1\}$ then H1
else if $a \in \{H0, M0, L0\}$ then H0
else Z

$\forall A \in 2^V$: $\downarrow(A) := \bigcup \{\downarrow(a) \mid a \in A\}$

The function $\downarrow(A)$ models what happens when a charged region becomes isolated. The following propositions can be proved easily:

Proposition 1

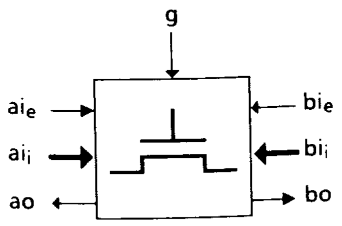
(i) $\forall A \in 2^V$: $\sup(A,A) = A$

(ii) $\forall A,B \in 2^V$: $\sup(A,B) = \sup(B,A)$

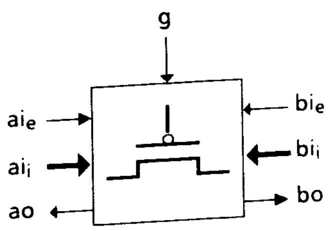
(iii) $\forall A,B,C \in 2^V$: $\sup(A, \sup(B,C)) = \sup(\sup(A,B),C)$

(iv) $\forall \in A \in 2^V$: $\downarrow(\downarrow(A)) = \downarrow(A)$

In order to model transistors we use devices with one gate input, two inputs for drain and source each and one output for drain and source each.



Positive Switch
(nMOS Transistor)



Negative Switch
(pMOS Transistor)

Here ai_e and bi_e denote what is offered to the tranistor when the tranisitor's influence is neglected, while ai_i and bi_i include the transistor's influence. The outputs *ao* und *bo* describe what is offered by the transistor.

Definition 17

Let be $PS{:}(2^V)^5 \rightarrow (2^V)^2$.

PS is called ***positive switch*** : < = >

(i) $\forall g \in V{:}\ \forall ai_e, ai_i, ao, bi_e, bi_i, bo \in 2^V{:}$

$$PS(g, ai_e, ai_i, bi_e, bi_i) = (ao, bo) \quad <=>$$

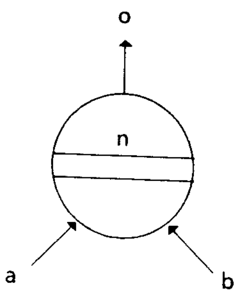
$$(ao, bo) = \text{if } g \in \{L0, M0, H0, Z\} \text{ then } (\downarrow(ai_i), \downarrow(bi_i)) \text{ else } (ai_e, bi_e)$$

(ii) $\forall G, ai_e, ai_i, ao, bi_e, bi_i, bo \in 2^V{:} \quad PS(G, ai_e, ai_i, bi_e, bi_i){:} = \cup \{PS(g, ai_e, ai_i, bi_e, bi_i) \mid g \in G\}$

The negative switch pMOS transistor is modelled in the same way.

A little more complicated is the definition of a node. By a node we mean a certain region that either may be connected to neighboured nodes or be isolated. When isolated it is capable to store a value for a finite period of time. This period is called "***charge decay time***".

The value behaviour over a certain period of time is represented by a sequence of values (i.e. a finite resolution of the observation equipment is assumed). Based on these assumptions a node with two neighboured nodes is defined in the following way:



Definition 18

Let be $nd{:}((2^V)^2)^n \rightarrow 2^V$.

nd is called ***node with charge decay time*** n : < = >

(i) $\forall a, b \in V{:} \quad nd(a_{t-n}, b_{t-n}; a_{t-n+1}, b_{t-n+1}; \ldots; a_t, b_t) =$

$$\text{if } (\forall t' \in [t-n, t] : \sup(a_t, b_t) \in \{H1, H0, Z\}) \text{ then } Z \text{ else } \sup(a_t, b_t)$$

(ii) $\forall A, B \in 2^V{:}\ nd(A_{t-n}, B_{t-n}; \ldots; A_t, B_t) = \cup \{nd(a_{t-n}, b_{t-n}; \ldots; a_t, b_t) \mid a_{t-i} \in A_{t-i} \wedge b_{t-i} \in B_{t-i}\}$

By the above discussion the model offers representations for nMOS transistors, pMOS ones, and for regions in between. Resistors (depletion mode transitors) are modelled simply by offering values of medium strength to nodes.

Of course the modell makes sense only if proper interconnection rules are formulated. For this purpose a graph grammar (being a little bit complicated) seems to be most adequate [14]. Within this paper an informal definition may be more helpful:

Definition 19

Given a transistor circuit consisting of transistors, pullups, pulldowns, regions, V_{DD}, *GND* and terminals in the following way:

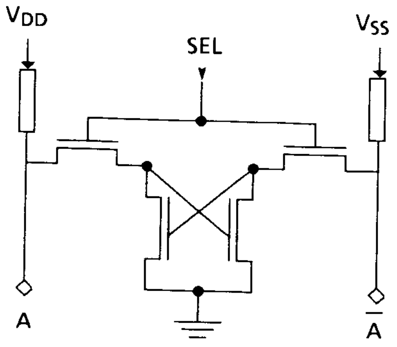
- For each transistor use a proper switch
- If two transistors send into one region:
  - introduce one node for this region
  - connect crosswise xo and xi_e-terminals
  - connect xi_e-outputs to the node
  - connect the node-output to the xi_i - inputs
- Connect the gate-inputs of switches with the proper node-outputs
- If a pullup or pulldown is connected to a region introduce a node and offer a medium strength constant.
- If more than two transistors (or pulldowns/pullups) are connected to a region
  - for each transistor t_i introduce a node n_i with the xo-outputs of all other transistors as inputs
  - collect these node outputs by an additional node n_g
  - connect the xi_e-input of the switch for transistor t_i with the output of node n_i
  - connect all xi_i-inputs with the output of node n_g
- Bidirectional terminals are handled like additonal transistors.

This model works very well as long as no circles are included in the circuit to be simulated. In this case some additional modifications of the model have to be carried out. They are a little bit complicated and shall not be discussed in this paper. However it has been observed that circles occure rather rarely if some typical situations like CMOS-transmission gates can be excluded by a special preprocessing.

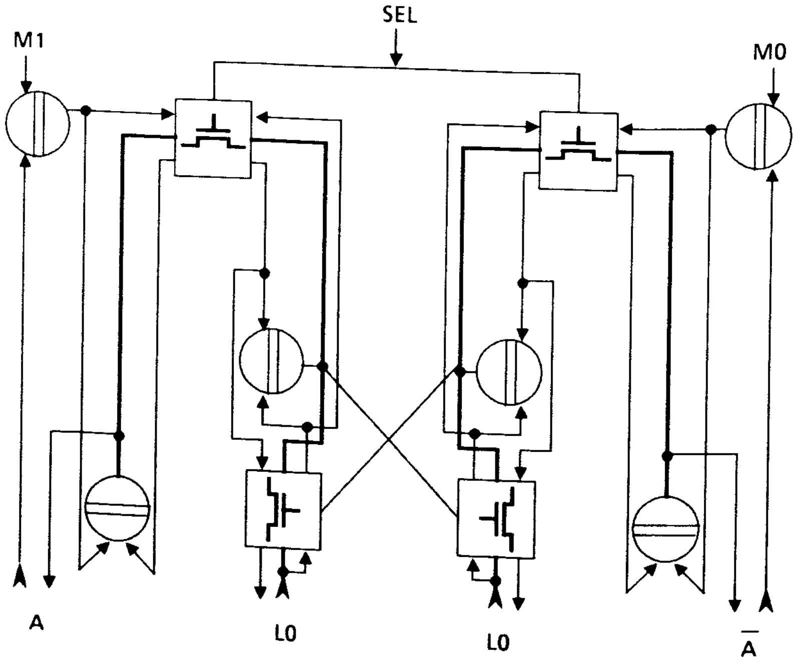
Example:

Just in order to show how a transistor circuit is transformed a simple memory cell is used. The benefit of the approach is obvious if it is assumed that there is an array of these cells. The simulation algorithm being a local one only the addressed (by SEL) cell is processed per operation. By experimenting with the model it can easily be seen that even the behaviour of the cell in the case where the refresh cycle is ommitted is modelled correctly.

The following RAM cell in nMOS technology has to be modelled:



Using the rules of definition 19 the following model is obtained:



References

[1] Barbacci, M.R.,Instruction Set Processor Specification (ISPS): The Notation and its Application, Dept of Computer Science, Carnegie Mellon University (1979).

[2] Belsnes, O., The Use of SIMULA for Real-Time System Implementation, Norwegian Computing Center, Oslo (1978).

[3] Bryant, R.E., MOSSIM: A Switch-Level Simulator for MOS-LSI, in:Proceedings 18th Design Automation Conference (1981).

[4] Chu, Y., Introducing CDL, IEEE Computer, Dec. 1979.

[5] Duley, J.R. and Dietmeyer, D.L., A Digital system Design Language (DDL), IEEE Transact. Comp. C-24, No. 2 (1975).

[6] Gordon, M.J.C., Register Transfer Systems and their Behaviour, in: Proceedings CHDL '81 (Sept. 1981).

[7] Groening, K., Zur Darstellung und Simulation von Interrupts in Parallelen Systemen durch Petri Netze (Representation and Simulation of Interrupts in Parallel Systems by Petri Nets), in German, Diploma Thesis, CS Dept., Univ. Dortmund (1979).

[8] Hayes, J.P., A Unified Switching Theory with Applications to VLSI Design, Proceeding of the IEEE, 70, No.10 (1982).

[9] Ichbiah, J.D. et. al., Preliminary ADA Reference Manual, ACM Sigplan Notices, 14, No.6 (June 1980).

[10] Jensen, K. and Wirth, N., Pascal User Manual and Report, (Springer, Berlin, 1978).

[11] Kawai, M., and Hayes, J.P., An Experimental MOS Fault Simulation Program CSASIM, in: Proceedings ACM IEEE 21st Design Automation Conference (1984).

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

[13] May, M.D., OCCAM, ACM SIGPLAN Notices, 18,No. 4(Apr. 1983).

[14] Muliyanto, E., Logiksimulation fuer MOS-Schaltungen (Logik Simulation of MOS Circuits), in German, Diploma Theses, CS Dept., Univ. Dortmund (1984).

[15] Noe, J.and Nutt, G., Macro E-Nets for Representation of Parallel Systems, IEEE Transact. Comp. C-22, No. 8 (1978).

[16] Piloty, R., Barbacci, M., Borionne, D., Dietmeyer, D., Hill, F. and Skelly, P., CONLAN Report (Springer, Berlin, 1983.

[17] Peterson, J.L., Petri Nets, ACM Computing Surveys (1977).

[18] Rammig, F.J., Structural Parallel Programming with a Highly Concurrent Programming Language, in: Atti di Congresso Annuale AICA '80 (1980).

[19] Rammig, F.J., Preliminary CAP/DSDL Language Reference Manual, Forschungsbericht der Abt. Informatik, Universität Dortmund, No. 129 (1981).

[20] Szygenda, S.A., TEGAS 2 - Anatomy of a General Purpose Test Generation and Simulation System for Digital Logic, in: Proceedings Design Automation Workshop (1972).

[21] Vladimirescu, A. and Liu, S., The Simulation of MOS Integrated Circuits Using SPICE, Memo VCB/ERLM 80/7, Univ. of Calif., Berkeley (1980).

[22] Wirth, N., Programming in Modula 2, (Springer, Berlin, 1982.

[23] Zeigler, B.P., Theory of Modelling and Simulation, (Wiley & Sons, 1976).