ISSN 0105-8517

SYNCHRONIZED TRANSITIONS

Anders P. Ravn
Jørgen Staunstrup

DAIMI PB – 219
January 1987

PB – 219 Ravn & Staunstrup: Synchronized Transitions

SYNCHRONIZED TRANSITIONS

Jørgen Staunstrup
Datalogisk Afdeling
Aarhus Universitet [*]

Anders P. Ravn
Institut for Datateknik
Danmarks tekniske Højskole [†]

Revised January 23, 1987

Abstract

This paper presents a high-level notation for designing a VLSI chip as a number of asynchronous state machines. The machines are asynchronous in that they make state transitions without reference to a common time standard such as a common clock. The notation allows high-level descriptions of the functional behavior of an electrical circuit. This makes it possible to explore and verify different design decisions at a very early stage of the design; before making a detailed implementation. By imposing certain constraints on the abstract description, it is possible to transform it into a layout or program by using so-called implementation rules.

[*]address: Ny Munkegade, DK-8000 Aarhus C, Danmark
[†]address: Bygning 343, DK-2800 Lyngby, Danmark

1 Motivation

A VLSI chip consists of registers and combinatorial functions. To structure the design, the registers are grouped in state components changed by the combinatorial functions. In the proposed notation state components are denoted by variables and functions by single assignment transitions. In an electrical circuit all parts run continuously, this is reflected by letting all transitions run in parallel. Synchronization of these transitions is expressed by predicates on state components without explicit reference to a clock. The notation allows high-level and abstract descriptions of the functional behavior of a chip. This makes it possible to explore and verify different design decisions before making a a detailed implementation. By imposing certain constraints on the abstract description, it is possible to transform it into a layout or program using so-called **implementation rules**.

The model underlying the notation describes a parallel computation by a number of asynchronous state machines. The machines are asynchronous in that they make state transitions without reference to a common time standard such as a common clock. Internal transitions, which only affect the state of a single machine, are performed completely independently of the other state machines. But one or more machines may perform a **synchronized transition** to communicate or otherwise coordinate their activities. During a synchronized transition, two or more state machines synchronize and make a simultaneous transition. After this transition is completed, the machines are again independent and can then make internal transitions or participate in other synchronized transitions.

The notation grew out of a number of attempts to describe a VLSI design as a collection of communicating processes in a CSP like notation. These descriptions all seemed unnecessarily complicated to construct and to read. Furthermore, all attempts to find a systematic way of transforming the descriptions to a VLSI design failed. The major problem was finding an efficient realization of the synchronous communication commands which is the cornerstone of CSP and all its derivatives. In a given state many synchronous communications may be possible and the choice of one particular communication does affect the choices available for other processes. The approach taken in SYNCHRONIZED TRANSITIONS is to replace input guards with predicates on state components. Thus, synchronization between asynchronous processes is modelled directly by a conditional change of common state components (e.g. a wire). Viewed this way, it is possible to formulate so-called **implementation rules** which are conditions (restrictions) that leads to efficient implementations. Viewed another way, implementation rules describe *special cases* which allows us to transform a description into an efficient VLSI design. So, the intention with SYNCHRONIZED TRANSITIONS is to find a notation which makes it easy to

- write simple high-level descriptions of the functional behavior and to
- systematically derive efficient VLSI implementations.

In this paper, we describe one notation for writing synchronized transitions; there are no doubt many other way of specifying a number of cooperating state machines. The notation is illustrated on three examples from VLSI chips which have been designed at Aarhus University. The second half of this paper discusses how descriptions using SYNCHRONIZED TRANSITIONS may be implemented.

2 Notation

The VLSI chip is viewed as a network of asynchronous state machines. Each of which performs some well defined part of the computation.[1] The network of state machines is described as a set of nodes and a set of transitions. A **node** is passive and consists of a data structure. This data structure represents the state of one state machine.

```
node = RECORD
         s: BOOLEAN;
         b: register;
       END;
```

Transitions describe state changes by single assignments where the list of values on the right hand side are assigned to the variables on the left hand side.

```
TRANSITION transfer(a, b: node);
< NOT b.s AND a.s -> b.b, b.s, a.s := a.b, TRUE, FALSE >
```

The transition `transfer` can only take place in a state where the two nodes a and b satisfy the predicate `NOT b.s AND a.s`. The transition will change the state so that:

```
b.b = a.b,   b.s = TRUE,   and   a.s = FALSE:
```

More generally a transitions is written as follows:

```
TRANSITION name(n1, n2, ... );
< B(n1, n2 ...)  ->  S(n1, n2 ...)  >
```

Here, `n1, n2 ...` are the nodes involved in the transition.

- `B(n1, n2 ...)` is the **precondition** of the transition; it is a boolean condition on the states of `n1, n2 ...` The transition can only be performed when its precondition is satisfied.

- `S(n1, n2 ...)` is a description of the state transformation made by this transition. It may only change the states of `n1, n2 ...`

A transition is atomic as indicated by `< ... >` [5]. This means that the transition appears to be performed indivisibly; this may, however, not be the way it is implemented. Transitions are repeated indefinitely once they are initiated. A transition is **enabled** when its precondition is satisfied.

Some transitions involve one node only, and these can be performed completely asynchronously with transitions involving other nodes only. But, the transitions affecting more than one node involve synchronization or communication. Such transitions

[1]Considering the processes as asynchronous on an abstract level does not preclude a completely synchronous implementation.

are synchronized with other transitions involving the same nodes to make transitions *atomic*. This means that the computations which can be described using the notation are sequences of transitions (an interleaving model).

There is no restriction on how many nodes a transition may involve. At the highest levels of abstraction, implementation efficiency is not directly represented; thus, it is possible to describe designs for which efficient implementations do not exist (or cannot readily be derived).

Large designs should be composed out of smaller components. Such decompositions are as essential when using SYNCHRONIZED TRANSITIONS as with any other notation. Constructs for modularization or decomposition are therefore important, this is discussed in further detail in Section 6, after a few explanatory examples are given.

Descriptions using atomic actions have been in use since the first papers on concurrency and synchronization. The key idea in SYNCHRONIZED TRANSITIONS is to focus on the interaction between processes rather than on the processes themselves. A similar viewpoint, called the "Global perspective" has recently been advocated as an alternative to the "Process-Eye view" [6]. Again it should be stressed that the motivation for this work is finding a notation which allows a smooth and systematic transformation from high-level descriptions to low level VLSI designs. At this stage, the transformations are manual, whether some or all can be automated has not been considered (yet).

3 A fifo queue

A fifo queue is a data structure capable of holding a sequence of elements. There are two fundamental operations on a fifo queue, insertion and removal. Elements are always removed in the same order as they are inserted (First In First Out). Such a queue may be implemented by a sequence of state machines, called nodes, each of which is capable of holding a single queue element. The state of a node is represented by the boolean variable s and a register b; where $s = FALSE$ means that the node is empty and $s = TRUE$, that it is full. The register may hold a queue element.

```
node = RECORD
          s: BOOLEAN;
          b: register;
       END;
```

When an element is inserted in the queue, it is given to the first node in the sequence (if this is empty). Whenever a node is empty and its predecessor is full, the element is handed from the predecessor to the empty node. This means that queue elements move down the sequence of nodes until it meets a node which is full. Several elements may be travelling down the sequence simultaneously. After a period without any operations on the queue all elements will have moved as far down the sequence as possible. In the following diagram empty nodes are shown in small letters, s, and full nodes in capital, S.

```
 in  ->  s1 -> s2  -> ... si -> SI+1...  -> SN -> out
```

When an element is removed (from out) this leaves an empty node which is filled from the left, leading to a shift of all the remaining elements. The movement of elements is described by transitions where:

- `transfer` moves an element from a node to successor, if it does not already hold an element,
- `input` inserts an element into the queue if there is room,
- `output` removes an element from the queue if there is any.

Finally, each node has a transition `init` which is used for resetting the queue (to empty).

```
    VAR (* external signals *)
      in, out: register;
      reset: BOOLEAN;
    TYPE
      node = RECORD
               s: BOOLEAN;  (* full <=> s *)
               b: register; (* holds one element *)
             END;
    VAR
      q: ARRAY [1..n] OF node;

    TRANSITION transfer(pred, succ: node);
    < NOT succ.s AND pred.s AND NOT reset
                  -> succ.b, succ.s, pred.s := pred.b, TRUE, FALSE >

    TRANSITION input
    < NOT q[1].s AND NOT reset -> q[1].s, q[1].b := TRUE, in >

    TRANSITION output
    < q[n].s ->  q[n].s, out := FALSE, q[n].b >

    TRANSITION init(i: node);
    < reset -> q[i].s := FALSE >

BEGIN
   ||$_{i=1}^{n}$ init(i) (* To make reset possible *)
   || input || output ||
   ||$_{i=2}^{n}$ transfer(q[i-1], q[i])
END;
```

Note that the transition `init` is enabled by an external signal `reset`. The manipulation of this signal is not shown in the description since it assumed to be done externally. Similarly the manipulation of the input/output registers is not shown.

The operator `||` initiates transitions. All initiated transitions run indefinitely and concurrently.

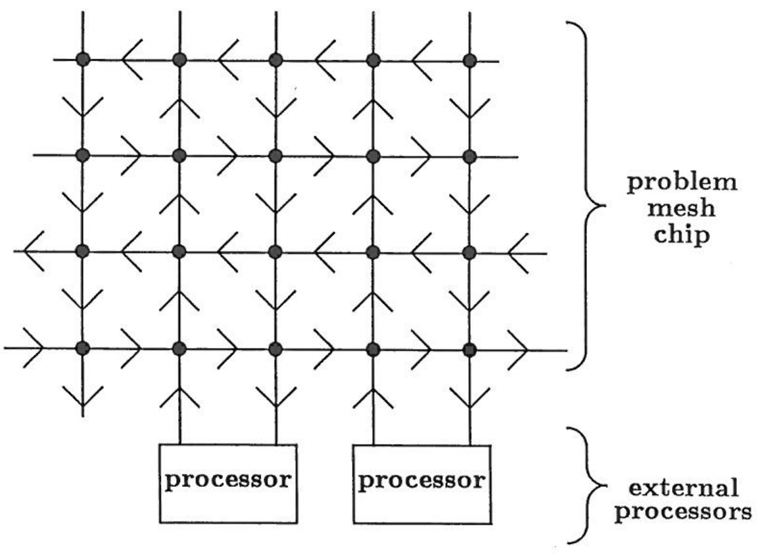


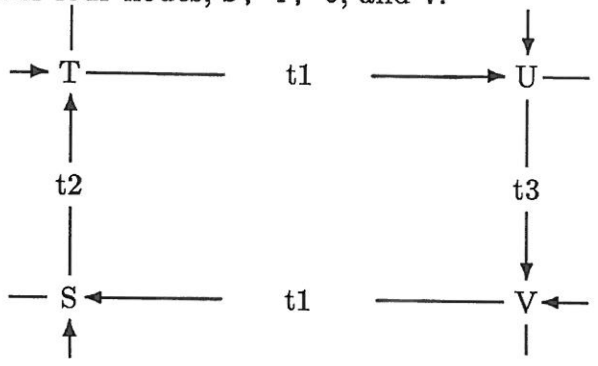
Figure 1: Sketch of problem-mesh

4 A problem-mesh

To illustrate the use of SYNCHRONIZED TRANSITIONS on a nontrivial example, we will describe the VLSI design of a so-called **problem-mesh.** A problem-mesh is a store with a number of external ports. At each port, external processes may change the contents of the store by inserting or removing elements. When removing from a port, an external process may get any element previously inserted; there is no requirement on the order in which the problem-mesh returns elements inserted into it. Elements may not disappear or be duplicated in the problem-mesh. So, the functional behavior of the problem-mesh store is as a multi-set. Furthermore, all external processes should be able to operate on the store simultaneously. This means that with $2 * n$ external ports, n insertions (from different processes) can be done simultaneously. The number of removals which can be done simultaneously depends on the number of elements in the heap and the implementation of the problem-mesh. The algorithm described below attempts to maximize the number of simultaneous removals. There are many alternative ways to realize a problem-mesh, one of which is discussed below. The reasons for using this implementation are not essential for illustrating the use of SYNCHRONIZED TRANSITIONS. A more thorough description and motivation for studying the problem-mesh is given in [4].

The problem-mesh is implemented with active storage units which are capable of exchanging elements. These elements may move around and hence float away from sections of the store with many elements towards sections with few. The choice of an optimal (or at least a provably good one) is in itself an interesting problem. Here, we concentrate on a particular strategy. As indicated by the name of the structure, the storage units are organized in a mesh, see Figure 1. Each node has four neighbors with which it may communicate. We have chosen to make the communication unidirectional with elements floating in the direction of the arrows (alternating directions in every other row/column).

A node may contain 0, 1, or 2 elements, and the rules for communicating elements between nodes attempt to spread out elements at the bottom (closest to the ports) of the mesh; much the same way as balls would spread out, if many are inserted at the same port. Temporarily, elements may pile up, but after a while they will spread out. We will now describe how such a strategy is described using SYNCHRONIZED TRANSITIONS. Consider four nodes, `S`, `T`, `U`, and `V`:



The three transitions, `t1`, `t2`, and `t3`, represent the three kinds of transitions found in the problem-mesh: a horizontal, one going up, and one going down. The state of a node consists of a counter (0, 1, or 2) and the registers containing elements.

```
  node = RECORD
           s: [0..2];
           b: registers;
         END;
```

The three transitions are described below. In this first version, we have concentrated on the conditions for exchanging elements, the actual transfer of these between nodes is just indicated with assignments of the form `U.b:= T.b`. The transfer is discussed in further detail in Section 7.2.1 and in Appendix A.

```
TRANSITION t1(T, U: node);
(* T --t1--> U *)
< T.s>0 AND U.s=0  ->  T.s, U.s, U.b := T.s-1, U.s+1, T.b >

TRANSITION t2(S, T: node);
(* S --t2--> T *)
< S.s=2 AND T.s<2  ->  S.s, T.s, T.b := S.s-1, T.s+1, S.b >

TRANSITION t3(U, V: node);
(* U --t3--> V *)
< U.s>0 AND V.s=0  ->  U.s, V.s, V.b := U.s-1, V.s+1, U.b >
```

Even though this description defines a very specific strategy, it is still rather abstract compared to what it needed for implementing it as a VLSI design. An example of this is the integer arithmetic used, in the finished design this is replaced by manipulations of single bits. Similarly, the synchronization needs refinement. In section 7.2.1, some of the transformations towards a more detailed design are shown.

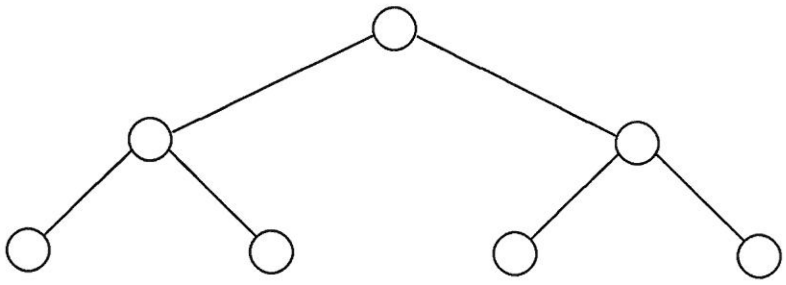


Figure 2: First levels of arbiter tree

5 An arbiter

An arbiter is a frequently used circuit for providing indivisible access to some shared resource e.g. a bus or a peripheral. The arbiter described here is implemented as a tree where all nodes (including the root and the leaves) are identical. The details of a node are described below using SYNCHRONIZED TRANSITIONS.

Each node has three pairs of connections, one for its father and one for each of its sons. A connection pair consist of two signals, `req` and `gr`. Such a pair is used according to the following four phase handshaking protocol.

1. A node raises the request (`req` signal to the father) to indicate that it wants to get the token.
2. When the grant is raised (`gr` signal from the father) the node has the token and it may pass it down the tree. When a leaf node gets the token, it means that it has indivisible access to the resource.
3. The token is handed back by lowering the request.
4. Lowering the grant implies the end of one cycle, i.e. now a new request can be made.

The external connections from a node are shown below:

```
-----reqf---------------grf-----
|                              |
|         arbiter node         |
|                              |
--reql---grl---------reqr---grr--
```

An external process request the resource by setting `reqi` (where *i* is l or r depending on which leaf the proces has been assigned). When `gri` is set (by the arbiter) the proces may go ahead and use the resource. The signals `reqf` and `grf` are connected to `reqi` and `gri` at the next (higher level) of the tree. When both sons of a particular node request the resource, it is first given to the left son, and when this releases the resource it is given to the right son. The `reqf` and `grf` of the root node are connected. This means that a request by the root is immediately granted. The arbiter can be described as follows:

```
    TYPE
     node = RECORD
              reql, reqr, reqf, grl, grr, grf: BOOLEAN;
            END;
   VAR
     tree: ARRAY[1..n] OF node;
     reset: BOOLEAN; (* external connection *)

   TRANSITION requestfather(i:node)
   < i.reql OR i.reqr  -> i.reqf:= TRUE >

   TRANSITION grantleft(i:node)
   < i.grf AND i.reqf AND i.reql AND NOT i.grr -> i.grl:= TRUE >

   TRANSITION grantright(i:node)
   < i.grf AND i.reqf AND i.reqr AND NOT i.grl AND NOT i.reql
                                               -> i.grr:= TRUE >

   TRANSITION doneleft(i: node)
   < i.grl AND NOT i.reql -> i.grl  := FALSE >

   TRANSITION doneright(i: node)
   < i.grr AND NOT i.reqr  -> i.grr  := FALSE >

   TRANSITION done(i: node)
   < i.grf AND NOT i.reql AND NOT i.reqr -> i.reqf  := FALSE >

   TRANSITION init(i: node)
   < reset -> grl, grr, reqf:= FALSE, FALSE, FALSE >

   TRANSITION connect(from, to: BOOLEAN);
   < TRUE -> to:= from >

BEGIN
   connect(tree[1].reqf, tree[1].grf) ||
   $||_{i=1}^{n/2}$ requestfather(i) || grantleft(i) || grantright(i) ||
      doneleft(i) || doneright(i) || done(i) || init(i) ||
      connect(grl, tree[2*i].grf) ||
      connect(tree[2*i].reqf, reql) ||
      connect(grr, tree[2*i+1].grf) ||
      connect(tree[2*i+1].reqf, reqr) ||
   $||_{i=n/2+1}^{n}$ requestfather(i) || grantleft(i) || grantright(i) ||
      doneleft(i) || doneright(i) || done(i) || init(i) ||
   (* make connections with external processes *) ...
END;
```

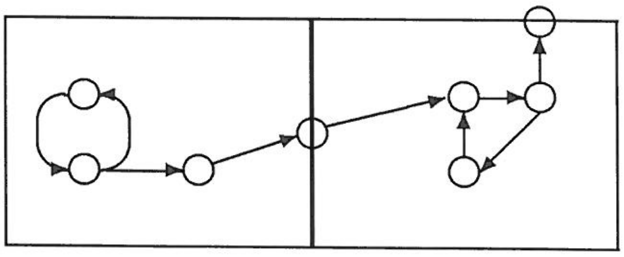


Figure 3: Module diagram

The transition `connect` shows how lower levels of a design can be described using SYNCHRONIZED TRANSITIONS. The transition `connect` is similar to a wire except that it has a sense of direction. A physical wire connecting two points `a` and `b` without any sense of direction could be specified as follows:

```
TRANSITION wire(a,b, temp: BOOLEAN);
< a<>b -> IF a = temp THEN a, temp := b, b ELSE b, temp:= a, a END >
```

6 Modularization

In this section we show how a large design may be described as a hierarchy of (relatively) independent modules. Each such module may consist of many state machines (state descriptions and transitions), so each of the examples given in the previous sections could be an independent module.

A module is an **encapsulation** of a part of the design. Its internal structure is hidden from the remaining parts of the design. Through encapsulation it is possible to reduce the amount of detail which must be considered. Modules can, however, not be completely independent, the different parts of a design do cooperate to perform the overall task of the circuit. Hence, it must be possible for independent modules to communicate. This can be achieved by letting transitions from one module reference the state variables of another. The external state variables used in the transitions of a module is called the **interface** of the module. Functionally, *a module is a set of transitions*. Hence, composition of modules corresponds to adding transitions.

Modules may be illustrated by diagrams such as the one shown in Figure 3, where the circles indicate state variables and the arrows transitions (the arrow leads from the state variables which the transition reads, to the ones it writes, thus the arrows may have multiple heads and tails). The interface of a module are the state variables shown in its bounding box.

A module may be written as shown in the following example:

```
MODULE node(external: signal; reset: BOOLEAN);
                  (* interface description *)
  VAR
    local: signal;
    ...           (* state description *)
  TRANSITION init
  < reset -> local:= external >
  ...             (* transitions       *)
BEGIN
  init ||
   ...            (* instantiation part *)
END node
```

The interface description is similar to the formal parameters of a procedure; so, components of an interface are referred to by their name. The *state description* and *transitions* are written as in the examples shown in previous sections. Transitions are initiated in the instantiation part, exactly as in the examples in the previous sections. Note that they may refer to formal parameters of the module (the interface).

Modules may also instantiate local sub-modules which are not externally visible. This is necessary to hide the internal structure of a module (encapsulation). Instantiation of a module is written in the same way as the instantiation of a transition, by giving its name and actual parameters. Functionally, instantiating a module creates a set of transitions.

```
... node(ext, reset) || ... further instantiations
```

Such a module instantiation creates the state of a module and initiates the internal instantiations of the module, which in turn creates all its transitions and possible further sub-modules.

This paper is not intended to be a reference manual for a full fledged programming language, so the syntax and semantics of the module notation is not described in further detail. Below is shown a new version of the arbiter from Section 5 where each node of the tree is an independent module.

```
TYPE
 pair = RECORD
          req, gr: BOOLEAN;
        END;

 connections = ARRAY[1..n] of pair;
               (* represents the wires connecting external processes
                  with the arbiter *)
```

```
MODULE arbiter(reset: BOOLEAN; ports: connections);
   VAR top: pair;

 TRANSITION wire(from, to: BOOLEAN);
 < TRUE -> to:= from >

 MODULE internal_node(f: pair; reset: BOOLEAN;
                      ports: connections; l, h: [1..n]);
                      (* ports[l..h] are connected to the sub-tree under
                         this internal node *)

   VAR l, r: pair; (* connections to sub-trees *)

   TRANSITION requestfather
   < l.req OR r.req  -> f.req:= TRUE >

   TRANSITION grantright
   < f.gr AND f.req AND r.req AND NOT l.gr AND NOT l.req-> r.gr:= TRUE>
   ...
   TRANSITION init
   < reset -> l.gr, r.gr, f.req := FALSE, FALSE, FALSE >

BEGIN
    requestfather || grantleft || grantright ||
    doneleft || doneright || done || init ||
    IF h-l >= 1
       THEN  internalnode(l, reset, conn, l, (l+h)/2) ||
             internalnode(r, reset, conn, ((l+h)/2)+1, h) ||
       ELSE  wire(conn[l].req, f.req) || wire(f.gr, conn[l].gr);
    END
END internal_node;

BEGIN (* arbiter *)
  internal_node(top, reset, conn, 1, n) ||
  wire(top.req, top.gr)
END (* arbiter *)
```

Passing arrays with varying bounds could be done more elegantly than shown in this example where the bounds are passed explicitly, but such details are outside the scope of this paper.

7 Implementation

The proposed notation is very general which makes efficient implementations difficult (if not impossible). As a high-level description tool, this need not be a problem. Below, a very simple implementation in Modula-2 [2] is shown. Such an implementation allow functional verification, simulation and experiments with the design at a very early stage. By imposing restrictions on the transitions, e.g. how many and which nodes a transition can affect, it is possible to derive simpler and therefore more efficient implementations. To illustrate this, it is shown how the problem-mesh can be used to derive a VLSI design.

7.1 A trivial implementation in Modula-2

This section describes a very simple way of implementing SYNCHRONIZED TRANSITIONS in Modula-2. We have used such an implementation for early experiments with the design. The notation used above is already very close to Modula-2 syntax, the only real difference is a transition. Each transition can be implemented as an independent process. Processes may be created on top of the coroutine mechanism in Modula-2. The module `kernel` used below provides concurrent processes, the details of how this is done are not important for this presentation. Furthermore, it is necessary to provide a module which implements atomic operations, for the indivisible transitions. This is done in the module `atomic` which provides the two operations `left`, corresponding to <, and `right`, corresponding to >. A Modula-2 program for the problem-heap is sketched below:

```
MODULE heap;
FROM kernel IMPORT pause, startprocess;
(* to implement concurrent processes *)
FROM atomic IMPORT left (* < *), right (* > *);
TYPE
  node = RECORD
           s: [0..2];
           b: registers;
         END;
(* T R A N S I T I O N S *)
PROCEDURE t1(to, from: node);
(* to --t1--> from *)
BEGIN
 LOOP left;
   WHILE NOT ( (to.s<2) AND (from.s=2)) DO
    right; pause; left;
   END;
   from.s:= from.s-1; to.s:= to.s+1; assign(from, to);
 right; END;
END t1;
```

```
PROCEDURE t2(to, from: node);
(* from --t2--> to *)
BEGIN   ...  END t2;
 ...
VAR
  mesh: ARRAY[1..n, 1..m] OF node;

BEGIN (* main program *)
  (* set up net  *)
  FOR i:= 1 TO m-1 DO
    FOR j:= 1 TO n-1 DO
      startprocess(t1(mesh[i,j], mesh[i,j+1]),...);
      startprocess(t2....);
      ..
  END; END;
END heap.
```

This implementation leaves a lot of room for optimizations making it more efficient. Despite this, it has been used for simulating nets with hundreds of nodes and transitions. Even on a small personal computer, the running time of these simulations is tolerable. In [3] a systematic way of transforming this design into a more efficient Modula-2 program is shown.

7.2 VLSI implementations

The motivation for SYNCHRONIZED TRANSITIONS is to introduce a notation which allows a smooth transition from the high-level design to a VLSI circuit. In this section, several alternative designs for the problem-mesh are derived to illustrate how a high-level design using SYNCHRONIZED TRANSITIONS can be transformed in a VLSI layout. Before going into the synchronization of several transitions, consider a single transition:

```
< B -> S >
```

Usually, `B` and `S` refer to the same variable(s) (representing the state of one or more nodes). The minimal requirements for an implementation are:

- the value of `B` must be determined before the execution of `S` is started,
- one iteration `B -> S` must be completely finished before a new iteration is started, i.e. `S` must be completed, before a new evaluation of `B` is begun.

The are many well known techniques for doing this, here we will use a traditional two-phase non-overlapping clock [1]. The two non-overlapping clock cycles are called φ_1 and φ_2. Hence a single transition can be implemented as follows:

```
[ phi1 AND B  -> [ phi2 -> S ]]
```

The construct `[ B -> S ]` means wait until `B` is true and then perform `S`. The difference from `< B -> S >` is that the transition is not assumed to be atomic; so, no synchronization is needed. This construct corresponds directly to a piece of hardware doing S whenever B is true.

The next step is to consider implementing a design with many transitions. We will describe two alternative ways of synchronizing the transitions: with and without a common clock.

7.2.1 Using a common clock

When a common clock is used, it is the same φ_1 and φ_2 which appear in all transitions, but this is usually not sufficient to implement the required synchronization. As an example, consider again the Problem-mesh. Any node, *to*, may receive elements from two of its neighbors, and if the node is empty ($to.s = 0$), there are two transitions (t1 and t2) which may take place:

```
TRANSITION t1(from, to: node);
(* from --t1--> to *)
<from.s>0 AND to.s=0 ->  from.s, to.s, to.b := from.s-1, to.s+1, from.b>

TRANSITION t2(from, to: node);
(* from --t2--> to *)
<from.s=2 AND to.s<2 ->  from.s, to.s, to.b := from.s-1, to.s+1, from.b>
```

The definition of a transition requires that it appears atomic, how can this be achieved using the simple two-phase clock described in the previous section? The solution is to look at a node as consisting of two half-nodes, each consisting of one register and a state bit (full/empty).

```
TYPE
  node = RECORD
           s1, s2: BOOLEAN;
           b1, b2: register;
         END;

(* Auxiliary functions *)
PROCEDURE empty(e: node): BOOLEAN;
BEGIN RETURN NOT e.s1 AND NOT e.s2 END empty;

PROCEDURE full(e: node): BOOLEAN; BEGIN RETURN  e.s1 AND e.s2 END full;
```

So, `s1,b1` is one half-node and `s2,b2` is the other. It is possible to write the transitions (in particular the preconditions) so that: *at most one transition at a time affect any half-node.* This means that there will never be a state of the network where a half-node can be changed by two different transitions. The consequence of this is that a very simple form of synchronization is sufficient. It is not necessary to make all transitions into critical sections which exclude each other, since it is guaranteed that the state of a node is not being read in the middle of an update. This can be achieved by the common two-phase clock shown above. To illustrate the approach, consider transition, t1, from the problem-mesh (The full description of this lower level version of the problem-mesh is given in Appendix A).

```
TRANSITION t1(to, from: node);
(* from --t1--> to *)
<from.s2 AND empty(to)-> from.s2, to.s1, to.b1 := FALSE, TRUE, from.b2>
```

Since the precondition `from.s2 AND empty(to)` is disjoint with all other transitions, affecting `from.s2` and `to.s1` (see Appendix A), the synchronization is implemented with a two-phase clock. The transition becomes:

```
TRANSITION t1(to, from: node);
(* from --t1--> to *)
[ phi1 AND from.s2 AND empty(to) ->
            [phi2 -> from.s2, to.s1, to.b1 := FALSE, TRUE, from.b2 ]]
```

From this description the step to an electrical circuit is very small, the precondition `e.s2 AND empty(w)` maps directly into a NAND gate and the assignments to variables map into set and reset signals for a flip/flop. The only statement left is the data transfer `w.b1:= e.b2`, it can either be realized directly (word-parallel design) or it can be refined further into a bit-serial design. In the latter case, a somewhat more complicated circuit is needed. The same circuit may, however, be reused in all the transitions. This design has been taken all the way to a physical layout. The details of the design have been elaborated elsewhere [4].

Two transitions may be performed concurrently and still appear as if they were indivisible, if the two transitions are independent. Two transitions are **dependent** if one writes a variable (state) which the other reads or writes.

> **Concurrent Read Exclusive Write Condition (CREW)**: All dependent transitions must have disjoint preconditions.

Two preconditions, $B1$ and $B2$, are disjoint if they cannot be satisfied simultaneously ($B1 \wedge B2 => false$). So the CREW condition says, that it must not be possible to satisfy the precondition of a transition while another dependent transition is enabled. There are other (weaker) conditions that allow transitions to be performed concurrently and still make them appear indivisible, but the CREW condition is sufficient for the examples discussed here.

As the next refinement, we consider two transitions where the preconditions are not disjoint:

```
    < B1 -> S1 > and < B2 -> S2 >
```

`S1` and `S2` change the state of the same node(s) and the conjunction `B1 AND B2` may be satisfied. Somewhere, a decision must be made about which transition to perform. Let us represent the outcome of this decision by a boolean `b`. If `b` is true, the first transition is performed, otherwise the second. The two transitions must be transformed as follows:

```
< (B1 AND NOT B2) OR (B1 AND B2 AND b)  ->  S1  >

< (B2 AND NOT B1) OR (B1 AND B2 AND NOT b) ->  S2  >
```

Now the conditions are disjoint and the transitions may be implemented as above. Regarding b, there are several alternatives. The simplest is to make a static choice about which transition should be performed, e.g.

```
< B1  ->  S1  >

< (B2 AND NOT B1) ->  S2  >
```

If this is not adequate, we can introduce a bit assigned in the two transitions as follows:

```
< (B1 AND NOT B2) OR (B1 AND B2 AND b)  ->  S1, b:= FALSE >

< (B2 AND NOT B1) OR (B1 AND B2 AND NOT b) ->  S2, b:= TRUE >
```

Finally, b could also be implemented by circuitry producing a random value; then, the choice of which transition to perform would appear non-deterministic.

7.2.2 Implementation without a common clock

Now, assume that a common clock is not used; this means that synchronous clock signals φ_1 and φ_2 are not available. Some mechanism is needed to ensure the internal consistency of a node, i.e. that states are not read and written at the same time. Here, we may still use a (local) two-phase non-overlapping clock. (There are many ways of maintaining state which we will not pursue here.) The key problem is to provide synchronization *between* nodes. In the absence of a common notion of time, this must be provided by some handshaking protocol between the nodes. There are many fundamental problems in this, for a thorough discussion see Seitz's chapter on timing in [1]. Using synchronized transitions does not resolve any of these fundamental problems with synchronizing circuitry in the absence of a common clock.

The following is an example of a handshaking protocol; there are many others. Consider a transition affecting a particular node:

```
          < B -> S >


         --------------
         |    node    |
         --------------
```

This transition must be synchronized with other transitions affecting the same node. This can be achieved with a pair of wires from the node to the logic implementing the transition (one pair for each transition). The two wires are called req and ack. The transition and the node are implemented as follows:

```
(* transition *)
[ [ NOT ack AND B -> req:= TRUE]
  [ ack -> S; req:= FALSE ] ]*

(* node )
[ [req -> ack:= TRUE ]
  [NOT req -> ack:= FALSE ] ]*
```

The sequence of values taken by the two wires `req` and `ack` is `(F, F)`, `(T, F)`, `(T, T)` and `(F, T)`. This sequence is repeated indefinitely and corresponds to the four phases of two non-overlapping clocks. This is not too surprising since the purpose of the two wires is to provide a temporary common notion of time. *As mentioned above, there are many fundamental difficulties in making asynchronous electrical circuitry follow this protocol.*

When several transitions affect a node, it means that several `req` lines go into the node which is then implemented as follows:

```
(* transition1 *)
[ [ NOT ack1 AND B1 -> req1:= TRUE]
  [ ack1 -> S1; req1:= FALSE ] ]*
...

(* node )
[ [req1 -> ack1:= TRUE |
   req2 -> ack2:= TRUE |
...
   reqn -> ackn:= TRUE ]

  [NOT req1 -> ack1:= FALSE |
   NOT req2 -> ack2:= FALSE |
   ...
   NOT reqn -> ackn:= FALSE ]
 ]*
```

As with the clocked implementation, it is necessary with a mechanism for choosing which transition to perform when several preconditions are satisfied. Any of the alternatives discussed above may be used.

The implementation without a common clock appears straightforward. However, with current technology the solution using a common clock appears to be more efficient for implementing the problem-mesh. This will probably change in the not too distant future when smaller circuitry becomes practical.

8 Conclusion

We have presented a high-level notation for describing an abstract model of a VLSI design. The intention with the notation is to enable verification, simulation and analysis at a very early stage of the design process. Implementation is done by systematically transforming the high-level description into a VLSI design.

Acknowledgement

Finn Barrett and Mark Greenstreet at Aarhus University have used SYNCHRONIZED TRANSITIONS for describing several different designs. Marks enthusiastic support of the ideas has been a major source of inspiration. Hans Henrik Løvengreens work with a formal model has lead to several clarifications.

References

[1] C. Mead and L. Conway, **Introduction to VLSI systems**, Addison-Wesley 1978.

[2] **Programming in Modula-2**, N.Wirth, Springer Verlag 1982.

[3] F. Barrett, **Problemhobe og VLSI**, Masters Thesis (in Danish), Computer Science Department, Aarhus University, Denmark, October 1986.

[4] J. Staunstrup, F. Barrett, M. Greenstreet and P. Møller-Nielsen, The Design of a Problem-mesh, **Proceedings from Comp-Euro 87, VLSI and Computers**, IEEE 1987.

[5] L. Lamport and F.B. Schneider, The "Hoare Logic" of CSP, and All That, **ACM Toplas 6, 2**, April 1984.

[6] An Example of Stepwise Refinement of Distributed Programs: Quiescence Detection, M. Chandy and J. Misra, **ACM Toplas 8, 3**, July 1986.

Appendix A

Below, a description is given of one way to implement the problem-mesh so that a a common two-phase clock is sufficient for synchronizing all transitions, i.e. it is not necessary to provide critical regions explicitly. The initialization and connection to the external processes are not included to avoid extraneous details. Similarly, the reset signal has been omitted; it could be done exactly as in the queue example.

```
TYPE
  node = RECORD
           s1, s2: BOOLEAN;
           b1, b2: register;
         END;

(* Auxiliary functions *)
PROCEDURE empty(e: node): BOOLEAN;
BEGIN RETURN NOT e.s1 AND NOT e.s2 END empty;

PROCEDURE full(e: node): BOOLEAN; BEGIN RETURN  e.s1 AND e.s2 END full;

(* The network has two kind of nodes: A-nodes and B-nodes
   connected as shown below. The names on the arrows are
   transition names. The transitions are described below:
                      ...           ...
                       |             |
                      t2            t3
                       |             |
          ...--t1--> A -- t1  --> B -- ...
                    node          node
                       |             |
                      t2            t3
                       |             |
                      ...           ...              *)
TRANSITION t1(to, from: node);
(* from --t1--> to *)
<from.s2 AND empty(to)-> from.s2, to.s1, to.b1 := FALSE, TRUE, from.b2>

TRANSITION t2(to, from: node);
(* from --t2--> to *)
<full(from) AND NOT to.s2 ->
                         to.s2, from.s1, to.b2 := TRUE, FALSE, from.b1>

TRANSITION t3(to, from: node);
(* from --t3--> to *)
<empty(to) AND from.s1-> to.s2,from.s1,to.b2 := TRUE, FALSE, from.s1>
```

In addition to these transitions which all transfer information between nodes, there are "internal" transitions which transfer information between the two registers of a node. The transition `i21` can transfer from `b2` to `b1`, and `i12` the other way. For the latter there is a slight difference between A and B nodes, therefore the two transitions, `i12a` and `i12b`. These transfers are dependent on the state of the neighbors, which are labelled as follows:

$$\begin{tikzcd} & y & \\ x \arrow[r] & m \arrow[u] \arrow[r] & z \\ & w \arrow[u] & \end{tikzcd}$$

```
TRANSITION i12a(m, w, z: node);
< m.s1 AND NOT m.s2 AND empty(z) AND NOT full(w) ->
                                  m.s1, m.s2, m.b2 := FALSE, TRUE, m.b1 >

TRANSITION i12b(m, y, z: node);
< m.s1 AND NOT m.s2 AND empty(z) AND NOT empty(y) ->
                                  m.s1, m.s2, m.b2 := FALSE, TRUE, m.b1 >

TRANSITION i21(m, z: node);
< m.s2 AND NOT m.s1 AND NOT empty(z) ->
                                  m.s2, m.s1, m.b1 := FALSE, TRUE, m.b2 >
```

The preconditions of the internal transitions are written so that internal and external transitions are never in conflict. When an external transition is possible (its preconditions is satisfied), none of the internal are possible and vice versa.