Testing Two-phase Transition Signaling based Self-timed Circuits in a Synthesis Environment

Prabhakar N. Kudva
Dept. of Computer Science, University of Utah

Venkatesh Akella
Dept. of EECS, University of California, Davis

UUCS-93-024

Department of Computer Science
University of Utah
Salt Lake City, UT 84112, USA

September 28, 1993

Abstract

The problem of testing self-timed circuits generated by an automatic synthesis system is studied. Two-phase transition signalling is assumed and the circuits are targetted for an asynchronous macromodule based implementation as in [?, ?, ?, ?]. The partitioning of the circuits into control blocks, function blocks, and predicate (conditional) blocks, originally conceived for synthesis purpose, is found to be very elegant and appropriate for test generation. The problem of data dependent control flow is solved by introducing a new macromodule called SCANSELECT (SELECT with scan). Algorithms for test generation are based on the Petri-net like representation of the physical circuit. The techniques are illustrated on the high-level synthesis system called SHILPA being developed by the authors.

Testing Two-phase Transition Signaling based Self-timed Circuits in a Synthesis Environment

PRABHAKAR KUDVA (*pkudva@cs.utah.edu*)

Dept. of Computer Science
University of Utah
Salt Lake City, Utah 84112

VENKATESH AKELLA (*akella@eecs.ucdavis.edu*)

Dept. of Electrical Engineering and Computer Science
University of California
Davis, CA 95616

Keywords: *testing, self-timed circuits, high-level synthesis, asynchronous systems*

Abstract. The problem of testing self-timed circuits generated by an automatic synthesis system is studied. Two-phase transition signalling is assumed and the circuits are targetted for an asynchronous macromodule based implementation as in [4, 11, 1, 6]. The partitioning of the circuits into control blocks, function blocks, and predicate (conditional) blocks, originally conceived for synthesis purpose, is found to be very elegant and appropriate for test generation. The problem of data dependent control flow is solved by introducing a new macromodule called SCANSELECT (SELECT with scan). Algorithms for test generation are based on the Petri-net like representation of the physical circuit. The techniques are illustrated on the high-level synthesis system called SHILPA being developed by the authors.

1 Introduction

Asynchronous/Self-timed designs are beginning to attract renewed attention as promising means of dealing with the complexity of modern VLSI designs. The advantages of self-timed systems include (i) absence of global clocking and associated problems of reliable clock generation and distribution and loss of valuable *power* in clock drivers. (ii) ability to lend themselves better for incremental modifications, as there are no global control schedules. (iii) capable of exhibiting better average case performance, as they do not have to wait for the next clock "tick" to arrive before a following operation can be triggered [9, 12]. Absence of global clocking and adherence to local communication protocols based on *handshaking* also renders self-timed circuits as *ideal* candidates for language-based (or behavioral) synthesis.

Testing of self-timed circuits has received little attention in recent times as most researchers have been focussing on specification, synthesis, and verification aspects self-timed design. In this paper we investigate a scheme to incorporate testing in a language-based synthesis scheme for self-timed circuits. In the process we rediscover the now-well-understood symbiosis between testing and synthesis. More specifically, we find that a partitioning technique originally devised for synthesis of self-timed circuits from behavioral descriptions actually lends itself very naturally to an elegant testing strategy. Also, we find that one of the basic macro modules, namely, the SELECT module, which is used to implement data-dependent control flow, can be modified slightly to yield a SCANSELECT module which makes the circuit more observable and controllable. (Details of the design of SCANSELECT are presented later in the paper).

The techniques presented in this paper are based on the high-level synthesis system for self-timed circuits called SHILPA [2]. The input to SHILPA are high-level descriptions in a concurrent HDL called hopCP. hopCP is a notation to describe asynchronous hardware behavior as a collection of

concurrent processes communicating through synchronous channels via *handshake* a la CSP and/or through restricted shared variables. The hopCP objects can be conveniently expressed as an annotated Petri net, where the annotations capture the data manipulation aspects of the hardware behavior.

The paper is organized as follows: In Section 2 we will compare our approach with the existing work in the area. In Section 3 we will introduce the high-level synthesis system SHILPA where this work fits in and provide the necessary background to understand the rest of the paper. In Section 4 we will discuss the details of the test generation algorithms and illustrate them on a simple example. In Section 5 we will provide some preliminary results to evaluate the performance and feasibility of our technique and finally we conclude in Section 6 by reviewing the major contributions of the work and outlining the directions of future work.

2 Comparison with Related Work

Hazewindus [7] provides a scheme to test delay-insensitive circuits in the Martin synthesis framework. They use a four-phase level-based signaling and gate-level circuits unlike our approach which uses two-phase transition signaling and macromodule based implementation. Keutzer et. al. [8] and Berel et. al. [3] explore testability of circuits generated from STGs. Keutzer et. al. require a full-scan circuit for all state elements while Bereel et. al. discuss the relationship between semi-modularity and testing. Both these works address testing of the *control* part only. In contrast, our approach handles both control and data parts. Ginosar and David [5] discuss the advantage of dual-rail encoded circuits in making the circuits self-diagnostic. This involves an overhead for implementing the complete data path in dual rail logic while our approach using single-rail datapaths. Finally, in a recent paper Pagey et. al. [10] outline an approach to test Sutherland's micropipelines by a partitioning approach similar to that proposed in our paper but they do not provide any detailed algorithms and unlike ours, it is not based on a high-level synthesis system.

3 Background

In this section, we shall provide the necessary background to develop the algorithms and main concepts of the paper. First we define transition signaling, then we provide a brief overview of the SHILPA system which includes the input language hopCP, the intermediate format HFG (hopCP Flow Graph), the compilation scheme called *action refinement* and the final circuit representation called NHFG (normal-form HFG). We present an example of synthesis in SHILPA and define predicate blocks, function blocks, and control blocks. Finally, we discuss fault-modeling in SHILPA.

3.1 Transition Signaling

In transition signaling also known as two-phase or *event-based* signaling, both the upgoing (rising) and down-going (falling) transition of a signal have *meaning* in the sense that the circuits respond to both the transitions. More specifically, all activity in a system is governed by the *occurence* of a signal transition rather than the the actual value of a signal (1 or 0). Natural advantages of transition signaling include higher performance and lower power consumption (since both the polarities of signal transition perform "useful" work). Asynchronous circuits generated by SHILPA are organized around *two-phase transition signaling* with the data-bundling assumption.[11, 4]. Data-bundling assumption implies that we guarantee that the data arrives at all the receivers before the corresponding control signal.

3.2 Overview of SHILPA

```
MODULE Example

TYPE  byte: vector 8 of bit

SYNCPORT b?,c! : byte

VARIABLE x,y,z : byte

FUNCTION

BEHAVIOR

P [x,y] <= (x>y) -> c!(x+y) -> P [x,y]
        |  (not(x>y)) -> b?x -> c!x -> P [x,y]
END
```

Figure 1: hopCP description of a simple example

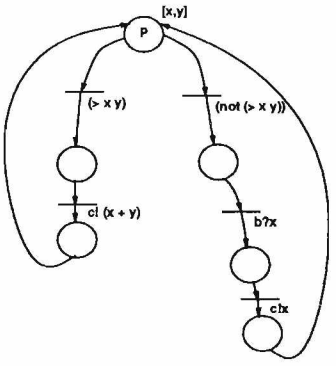

Figure 2: HFG for the specification in Figure 1

Consider the simple hopCP specification shown in Figure 1. In hopCP, hardware is specified as a set of sequential processes communicating over a set of synchronous channels (a la CSP) or via shared variables. In the module example in Figure 1, there is a single process `P`, which communicates with the external environment using synchronous channels `b?,c!` (where `?` denotes *input* channel and `!` denotes an *output* channel). `x` and `y` denote the internal variable (datapath registers for example). `<=` denotes definition of a process, `->` denotes sequencing, and `|` denotes a conditional. Notice that there are no clocks in the specification, only the causal ordering of the various actions is presented. The behavior of the module is specified in the `BEHAVIOR` section and is read as follows: if the value of `x` is greater than `y` then, the value denoted by `(x+y)` is output on channel `c!`, otherwise, a new value is read in from channel `b?` and stored in the register `x`. The same value is output on channel `c!`. After that, the module repeats its behavior.

This behavior is conveniently represented in the form of a annotated Petri net called HFG, where the places denote the *state* of the system, the actions such as `c!(x+y)`, `b?x`, denote the transitions of the Petri net. The variables and the functions which get modified by the execution of the system are captured by the annotations.

Figure 2 shows the underlying HFG for the specification in Figure 1. The HFG is converted into a self-timed circuit using a syntax-directed translation procedure called *action refinement* [1]. Action refinement consists a set of Petri-net transformations to convert the HFG into an RTL description

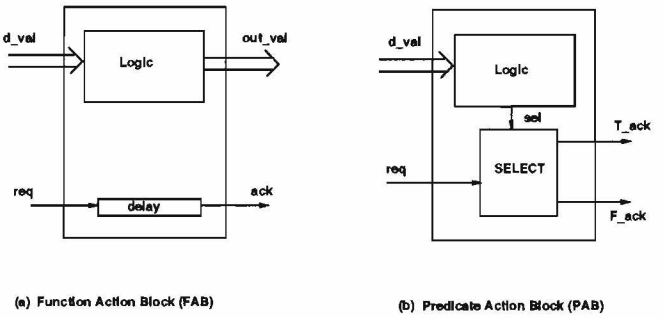

Figure 3: Models for FAB and PAB

denoted by a NHFG (normal-form HFG) and a set of resources.

Every circuit block is modeled as an *action block*, with optional bundled data. An *action block* implements a hopCP action, and presents a *self-timed interface*, consisting of the *initiate* and *completion* signals. If a transition is received on the *initiate* terminal, a transition is produced on the *completion* terminal *after* an unspecified amount of time. If the action block being considered produces *data* output, the completion signal is produced *only after* valid data is produced at its output. There are three types of action blocks in SHILPA. *Control Action Blocks* (CABs) model the control flow, *Function Action Blocks* (FABs) implement functions and *Predicate Action Blocks (PABs)* implement Boolean predicates. The models for PAB and FAB are illustrated in Figure 3.

Figure 3(a) illustrates the architecture of a FAB which is based on the following assumption. The control `req` signal will be asserted only after the arguments to the functions (denoted by `d_val`) are available. The control signal `ack` will be asserted after the result (denoted by `out_val`) is produced. The `delay` denotes the processing delay of the block. Figure 3(a) illustrates the architecture of a PAB which implements a predicate (Boolean function) which has the following protocol. After the arguments (`d_val`) are available, the control signal `req` is asserted and depending on whether the predicate evaluates to true or false, the corresponding control signal (`T_ack` or `F_ack`) is asserted. As shown in the figure a PAB can be thought of as a combination of a logic block and a SELECT element. The logic block produces a transition on the `sel` input of SELECT module, which in turn produces a control signal `T_ack` or `F_ack` at the output of the PAB.

It is important to note that this style of partitioning the whole circuit into PAB, FAB or CAB offers the following advantage. The control and data only interact inside a PAB. The interaction is localized to the `sel` signal of the SELECT modules. By making this signal controllable and observable one can solve the problem of handling data dependent control flow during testing. That is exactly what is done in our approach by modifying a SELECT module to a SCANSELECT (which is described later in this section).

NHFG and Final Circuit

The NHFG and the circuit for the example in Figure 1 are shown in Figure 4 and Figure 5. In a NHFG, `M.a??` denotes wait for a signal transition on the input `a` of module `M` and `F.b!!` denotes a transition produced on the pin `b` of the module F. Note that, the NHFG captures the netlist (or connectivity) of the various circuit elements.

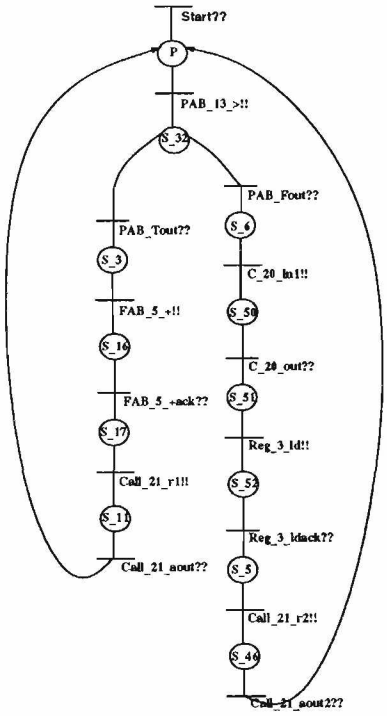

Figure 4: NHFG for the example

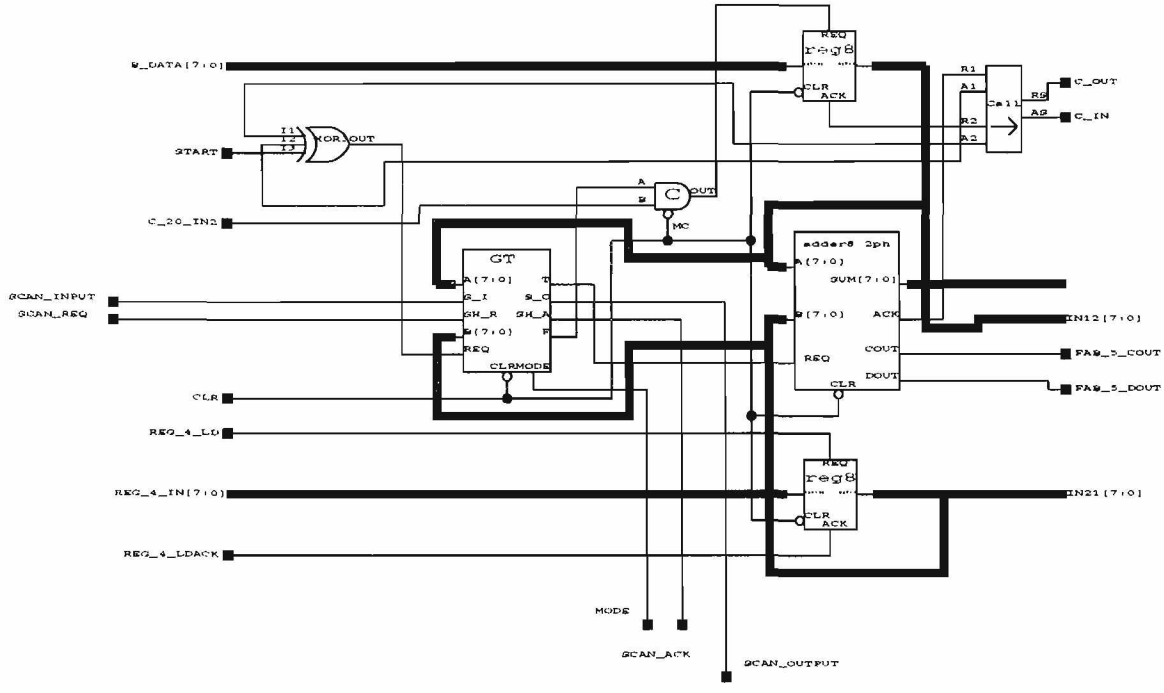

Figure 5: Circuit for the example

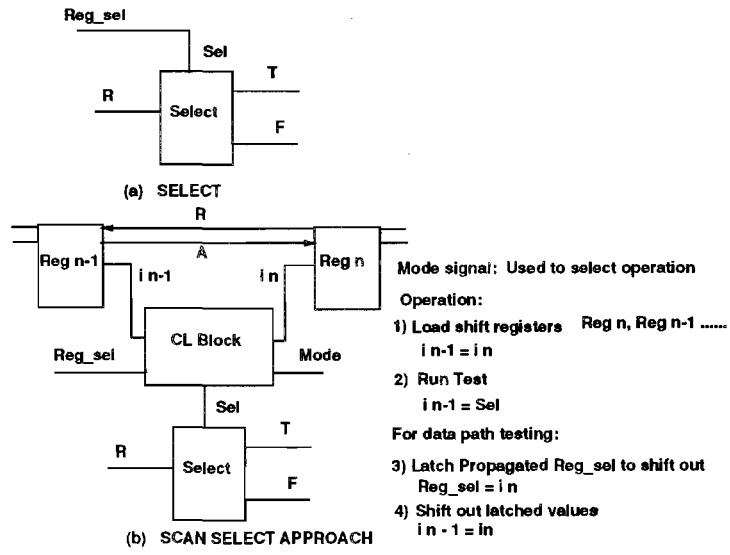

Figure 6: Representation of the SCANSELECT module

3.3 Fault-Modeling in SHILPA

In order to test the circuits generated by SHILPA completely, two types of faults have to be considered: (a) stuck at faults on the various nets and (b) delay faults which result in the violation of the data-bundling assumption. In this paper, we only consider stuck-at faults. To test a node for a stuck-at fault, we try to force a value opposite to what the node is stuck-at and propagate the *effect* of that value to a primary output. This involves an elaborate process for justification and propagation. In case of transition signalling, if a node is stuck-at 1 or 0, it *blocks* any events flowing through that *path*. It is useful to visualize a transition-signaling based self-timed circuit as a set of computing elements connected by *hollow pipes* with *marbles* (denoting events) flowing through them. A stuck-at fault blocks or obstructs the marble. This property can be used to test the circuits more efficiently when compared to standard level-based circuits.

For example, it is easy to see that if one could *successfully* propagate a $0 \rightarrow 1$ transition and a $1 \rightarrow 0$ transition through a node, then the node is devoid of both stuck-at 0 and stuck-at 1 faults. Furthermore, all the nodes in the path of this node, starting from a primary input to the primary output, are also *fault-free*. That means we need not model faults on each and every node in the circuit. This result in a considerable saving in the total number of tests required to test the circuit fully.

However, propagating a $1 \rightarrow 0$ transition or $0 \rightarrow 1$ transition through an arbitrary node, in presence of data dependent control flow, could be tricky, i.e., one may not be able to always find a suitable justification sequence. We solve this problem by modifying the design of the SELECT module in the library of asynchronous modules suggested by Sutherland [11]. This is possible in SHILPA because the only place control flow and data flow interact is within a SELECT module.

3.4 SCANSELECT

Figure 6(a) shows the normal SELECT module described in [11] and used in SHILPA. `Reg_sel` denotes the output of the datapath (logic) block which gets connected to the `sel` input of the SELECT module. The behavior of the SELECT module is as follows. A transition on the request input (denoted by R) after the value on the `sel` input is stable, yields a transition on the `T` or `F` output, depending on whether the value on the `sel` input is a logic "1" or "0". In general, `sel` signal is not fully observable or controllable and lies buried in the PABs. To make the `sel` signal *fully* observable and controllable we make the following modification to evolve a SCANSELECT module which is shown in Figure 6(b).

The SCANSELECT module has *four* modes of operation which are controlled by the `mode`, the *req* and *ack* signals of the shift register FIFO. First the values that are required during testing in registers, $reg_1 \ldots reg_n$, are shifted in by connecting terminals i_{n-1} and i_n. This is achieved by setting the `mode` signal. The values are shifted into the FIFO serially, using a sequence of request acknowledge pairs. After the registers are loaded, the `mode` signal is changed such that the output of reg_{n-1} is connected to the `sel` signal of $SELECT_{n-1}$. This renders the `sel` signal fully controllable and effectively decouples the datapath from the control part. This is sufficient for testing the control part (CAB). While testing the datapath of a circuit one finds that the `Reg_sel` (an output of the data path) value needs to be observed. For this, the `mode` signal is set such that `Reg_sel` is connected to terminal i_n which allows the latching of the value on `Reg_sel` in reg_n. This can then be shifted out by setting the `mode` signal such that i_{n-1} to i_n are connected.

The SCANSELECT needs five additional pins, namely, `mode`, `shift-request`, `shift-ack`,`shiftdata` and `shiftdataout` to achieve the functionality described above. The overhead in terms of logic for each SCANSELECT is 1 C element, 1 transition latch and two multiplexers to implement the control logic (denoted by CL block in the figure).

4 Algorithm for Test Generation

In this section, we present the details of the algorithms to test for stuck-at faults in a self-timed circuit based on its NHFG representation. First, the circuit is partitioned into control part consisting of only the CABs and the data part consisting of FABs and PABs. Then the nodes in each part are modeled separately in the NHFG and the test vectors are generated by traversing the corresponding paths in the NHFG. Initially, we present the top level algorithm which invokes procedures `testCab` and `testDatapath` to test the control and data parts separately. The pseudo-code of the algorithms is presented first and then it is illustrated on the circuit for the example in Figure 1

4.1 Top-Level Algorithm

GLOBAL:

tested-hfg-nodes, choices-made

INPUT:

NHFG, netlist, reslist

OUTPUT:

P /* *set of physical test vectors for the datapath* */

T /* *set of traces* */

C /* *choices-made for the scan-path* */

METHOD:

(N_c, R_c) = extract-control(NHFG, netlist, reslist);

(N_d, R_d) = extract-datapath(NHFG, netlist, reslist);

T_1 = testCab(NHFG,N_c, R_c)

(P, T_2) = testDatapath(NHFG,N_d, R_d)

return $(T_1, (P, T_2), C)$

The input for the top level algorithm is the output of the SHILPA system, namely, the NHFG, the set of resources (hardware modules), and the physical netlist. `tested-hfg-nodes` and `choices-made` are two global variables used by all the routines. The output of the algorithm are (i) physical test-vectors

for all the stuck-at faults in the datapath (denoted by P), (ii) sequence of traces or control sequences (denoted by T) to test the control part of the circuit and to set up conditions to test the datapath with P, and (iii) the choices made at points of data-dependent control (denoted by C) to be used in setting up the scan chain appropriately during testing.

The first step of the algorithm is to partition the circuit into control part and data part (accomplished by appealing to functions `extract-control` and `extract-datapath`). The control parts consists of all the CABs (of SHILPA) and forms a meaningful circuit if the `request` and `acknowledge` pins are connected together. The data part consists of pure logic blocks (PABs and FABs of SHILPA) with hooks to the control part for the corresponding `request` and `acknowledge` signals. This partitioning is merely to simplify the test generation procedure, since, different algorithms will be used for each of part.

4.2 Testing Control Circuitry

The procedure for testing the control part of the circuitry, called *testCab* is shown below.

```
procedure testCab (nhfg,netlist,reslist)
          traces = nil
begin
     For each node in netlist
          hnode = Extract-Node (nhfg)
          if hnode ∈ tested-hfg-nodes skip
          else
               new-trace = Derive-Trace(nhfg,netlist,hnode)
               tested-hfg-nodes = insert(tested-hfg-nodes, new-trace)
               traces = cons(new-trace,traces)
end
     return(traces)
```

The function `extract-node` finds the transition in the NHFG corresponding to a given name from the netlist. If the node has already been tested (by virtue of lying in the path of a different tested node) then we do nothing, otherwise, the function `Derive-Trace` is invoked to generate a test sequence for the node. The rest of the functions are for book-keeping.

```
procedure Derive-Trace (nhfg,netlist,hnode)
     initial-node = get-initial-node(nhfg)
     trace = Visit(initial-node,hnode,nhfg)
     return(trace)

procedure Visit(init-node,hnode,nhfg)
begin
   path1 = find-shortest-path(init-node,hnode)
   path2 = find-shortest-path(hnode, hnode)
     /* here were are computing the sequence of transitions
          from hnode to the occurence of hnode again */
   if (path2 = null) then
```

```
    begin
      path3 = find-shortest-path-to-primary-output(hnode)
       test-path = concatenate(path1,path3)
      result =   concatenate(test-path,test-path)
    end
  else
    begin
      path3 = find-shortest-path-to-primary-output(hnode)
       result = concatenate(path1,path2,path3)
       choice = getChoicenodes (result)
    /* extracts all the choice nodes in the result */
       choices-made = cons(choices-made,choice)
    /* Note that choices-made is a global variable which records all data
    dependent choices to be used later to set up the scan paths */
    end
/* list-of-preconditions is generated during the find-shortest-path routines when
     transitions with more than one input PLACE are encountered.
    Note that NHFG is a Petri net */
    while (list-of-preconditions-to-be-satisfied != nil)
          begin
           new-trace = Visit(init-node, head(list-of-preconditions-to-be-satisfied),nhfg)
           result = append(result, new-trace)
          end
    return(result)
end
```

The algorithm `testCab` works as follows: First a physical node is selected from the netlist and the corresponding transition is marked on the NHFG, if the node has not been tested previously as a consequence of some other test. Let n be the node being tested and t be the corresponding transition in the NHFG. Then a path (sequence of "enabled" transitions in the NHFG) is selected from the initial state (state of the system on global reset) to the state in which t is *enabled* (in the Petri net sense). Call this $path_1$. Another path is selected such that we begin in a state in which t is *enabled* and then t is revisited. These are analogous to the justification sequences in synchronous testing. The significance of $path_2$ is the requirement that the node n being tested be subjected to an $1 \rightarrow 0$ transition and a $0 \rightarrow 1$ transition. If $path_2$ is empty, then $path_1 :: path_3$ is traversed *twice* to get the same effect (where :: denotes concatenation of sequences). $path_3$ is chosen which corresponds to the sequence of transitions from t to a primary output. The final test sequence then is $path_1 :: path_2 :: path_3$. In generating the paths (or sequences) one should remember that NHFGs are not simple graphs but Petri nets, which means that some transitions could have more than one input place (for example, nodes involving C elements). This results in the additional requirement to find all possible paths to enable the given transition. Note, that this is accomplished by recursively invoking procedure visit.

Finally, in the computation of the paths, one can encounter *choice* nodes (which are implemented as part of a PAB in SHILPA). The nodes are recorded in the global variable to derive the possible scan

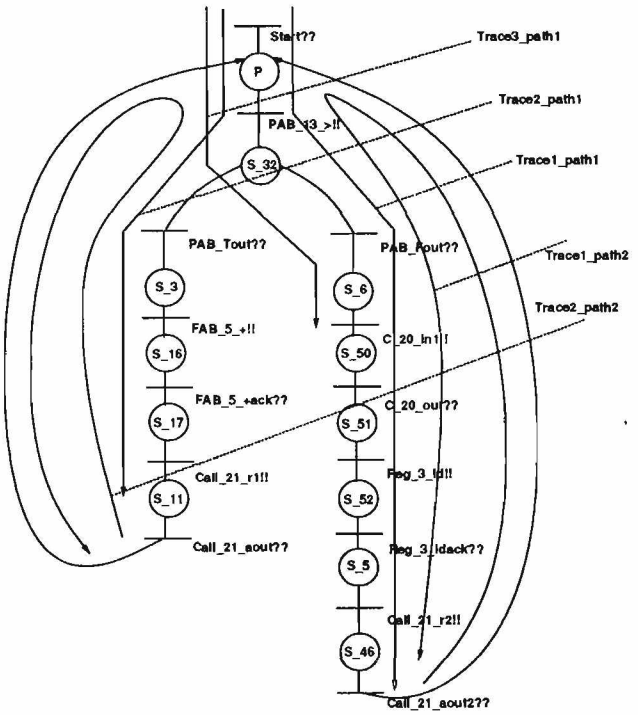

Figure 7: NHFG for the example with paths

vectors.

4.2.1 Illustration of testCab

Consider the testing of the one of the outputs of the CALL module in the circuit shown in Figure 5. First the corresponding transition $CALL_{21}.aout2??$ is selected in the NHFG shown in Figure 7. $path_1$ (sequence of states from the initial state to the state in which the given transition is fired), $path_2$ (the sequence of states from the state in which the given transition is fired to itself), and $path_3$ (the sequence of states from the state in which the given transition is fired to a primary output) are generated by the procedure `Derive-trace`. They are shown as $Trace_1.path_1$, $Trace_1.path_2$, and $Trace_1.path_2$ in Figure 7. For $Trace_1$, we find the following.

$$\begin{aligned} path_1 &= P \rightarrow S_{32} \rightarrow S_6 \rightarrow S_{50} \rightarrow S_{51} \rightarrow S_{52} \rightarrow S_5 \rightarrow S_{46} \\ path_2 &= S\_46 \rightarrow P \rightarrow S_{32} \rightarrow S_6 \rightarrow S_{50} \rightarrow S_{51} \rightarrow S_{52} \rightarrow S_5 \rightarrow S_{46} \\ path_3 &= empty \end{aligned}$$

We find that during the course of this trace, nodes corresponding to transitions

$$PAB_{13}.Fout??, C_{20}.in1!!, C_{20}.out??, Reg_3.ld!!, Reg_3.ldack??, Call_{21}.r2!!$$

also get tested. Next, let us test the other output of the CALL module. The corresponding transition is $Call_{21}.aout??$ and $Trace_2$ is generated in a similar fashion (as shown in Figure 7). This tests nodes corresponding to $PAB_{13}.Tout??, FAB_{5}.{+}!!, FAB_{5}.{+}ack??, Call_{21}.r1!!$. The nodes that remain to be tested are $start??$ and $PAB_{13}.\rangle$. No path traversing these nodes twice can be derived. So, instead we derive a path that traverses these nodes once (going up to the primary output) and execute it twice.

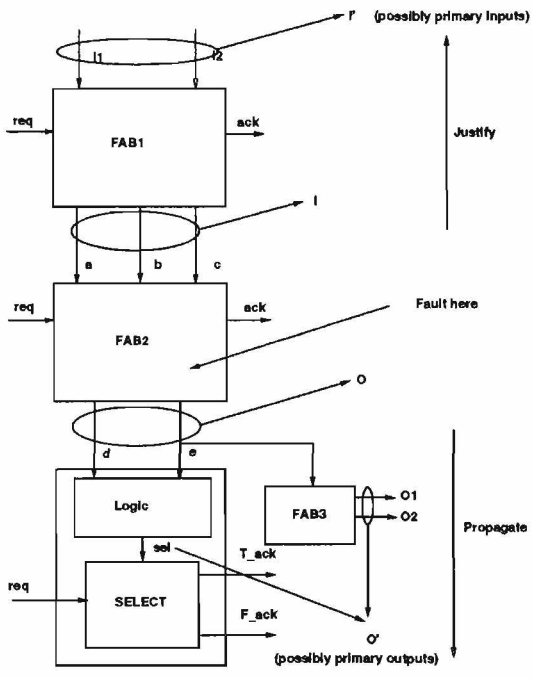

Figure 8: Illustration of data path

This corresponds to $Trace_3$ in the Figure 7. The *final test sequence* is ($Trace_1$, reset, $Trace_2$, reset, $Trace_3$). This way we have tested the *complete* control part of the circuit.

We notice that in generating $Trace_1$ and $Trace_2$, we have gone through the state S_{32} which is a data dependent choice, controlled by the input of the corresponding SELECT module. In order to subject the system through these traces, one has to set the input of the SELECT module appropriately (and independent of the data values) and this is achieved by remembering the values of the SELECT input for a given trace and setting up the corresponding scan path from the external world during testing. This is the reason why a SCANSELECT (discussed in Section 3) in our synthesis instead of a simple SELECT module.

4.3 Testing Data Part

Figure 8 depicts the organization of the datapath in our synthesis scheme. The data path can be viewed to consist of PABs and FABs with control signals acting as the interface to the CABs (in the control part). The only point at which, the data has effect on control is at the SELECT element inside PABs. The top level algorithm for testing the data part is given below.

```
procedure testDatapath(nhfg,netlist,reslist)
begin
    datapath = GetDatapathFromNetlist(netlist,reslist)
     for each FAB/PAB F in datapath
     begin
         (I, O) = AdaptedDalgorithm(F,Fault)
         J  = set of FAB/PAB affected in justifying I
         P  = set of FAB/PAB affected in propagating O
```

```
            control-points =  FindAffectedControlPoints(F)
            markedHfg = MarkHfg(nhfg, control-point)
            Primary-input = justify-data(J,I,markedHfg)
            Primary-output = propagate-data(P,O,markedHfg)
        listOfTraces =  findCover(markedHfg)
        /* Find a path(s) covering all the marked nodes in the nhfg
              starting from the initial node */
           choices-made =  getChoicenodes (listOfTraces)
        /* extracts all the choice nodes from the listOfTraces */
        test = (list Fault, ListofTraces, choices-made,Primary-input, Primary-output)
        test-list = (cons test test-list)
        end
    return test-list;
    end
```

We have the option of modeling the whole circuit as a single (giant) combinational block, by setting up the control signals appropriately. (Sutherland points this out in discussing micropipelines [11]). One could then use standard combinational test generation algorithms to detect the stuck-at faults. However, we find that it is more *efficient* if the standard test generation algorithms are modifed to only test for the FAB's and PAB's that are *affected* by the fault. Unlike a synchronous circuit, in a self-timed circuit there is no global clock signal and hence most parts of the circuits are not *activated* in a given operation. In fact, that's where the many of the advantages of an asynchronous circuit come from. We want to use this advantage in testing too. Hence we *unlike* [10] modify the standard combinational test generation procedure to *process* only those FABs and PABs that are *active* for a given stimulus. This is done by the procedure `AdaptedDalgorithm`.

We will use datapath representation in Figure 8 and the example in Figure 9 to illustrate the algorithm. The top level algorithm takes as inputs, the netlist, the resource list and the NHFG of the circuit. The output is a list of *Fault, Choices-made, ListofTraces, Primary-input, Primary-output* The outputs of the algorithm are used as follows. For each *Fault*, we first set up the scan chain as identified by *Choices-made* thus decoupling the data and the control. We then apply the input vector on the input data pins *Primary-input*. The control sequences in the *ListofTraces* are applied. This moves the data from the primary inputs, through the fault to the primary outputs. The Primary-outputs are then observed. In addition to the primary outputs, the outputs of the scan chain are observed to look for particular *sel* signals of Select modules (data path outputs).

The algorithm works as follows. We begin by first identifying the fault in a given FAB/PAB, and then applying the D algorithm on the datapath of the FAB (F in example). The result represented, by (I, O) represent the input needed and the outputs respectively to test sensitize and test the fault. This is shown in Figure 8 The inputs I are either primary inputs or the outputs of some other FABs (set J). In which case, the wires values in I need to be justified through this set of FABs. This needs to be recursively performed until primary inputs are encountered and the values at these primary outputs are noted. This recursive part is implemented by the function *justify-data* (given later in the paper). It recursively justifies backwards through the FAB's generating new sets of *affected* FABs (J') and justifying the outputs at these FAB's (I') to their respective inputs. Propagation is similarly done by the function *propagate-data* till primary outputs are encountered. It is important to note here that a `sel` input to a select element encountered during propagation is considered a *primary output* due to

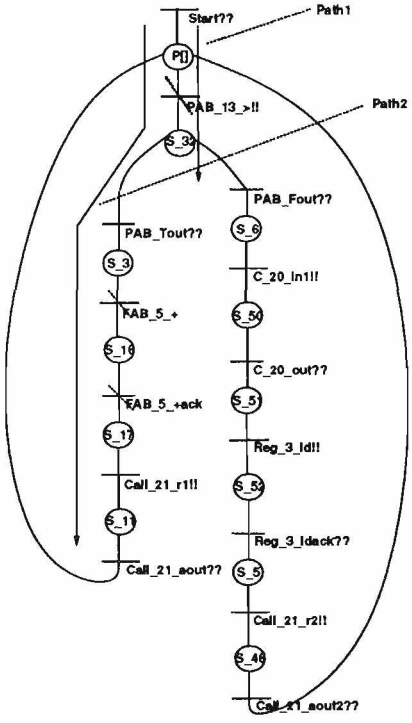

Figure 9: Illustration of marking HFG

the existence of the scan chain. During both justification and propagation, we have to note the control points that need to be activated to move the data from input to the output in each FAB. These control points are marked on the NHFG. The markings on the NHFG indicate a sequence of control actions that need to be performed in order to move the data from the primary inputs, through the fault to the primary outputs. At this stage, we use graph based algorithms to obtain traces that cover all the marked points on the NHFG. While obtaining the paths, we also find that we traverse choice nodes, that need to be decided independent of the data values (since we are testing the data path). We note the choices that need to be made for each path. In the example in Figure 9, we find that the two traces to cover all the points are path1 and path2. Path1 has *choices_made* as (S_32, True) and path2 has *choices_made* as (S_32, False).

procedure justify-data(J,I,markedHfg)
for each FAB/PAB f **in** J
begin
I' = AdaptedDalgorithm(f,I)
control-points = FindAffectedControlPoints(f)
markedHfg = markHfg(markedHfg,control-points)
J' = set of FAB/PAB *affected* in justifying I'
if null J' return I'
else justify-data(J',I',markedHfg)
end
procedure propagate-data(P,O,markedHfg)

Table 1: Table of experimental results

Circuit Name	Num gates	Control test paths	data path test vectors/paths	Percent Faults testable
GTADD8	233	3	338/3	100
PMULT8	377	4	602/6	100
QR42	64	2	0/0	100
fact8	379	4	626/6	100
SALU8	348	6	640/6	100

```
for each FAB/PAB f in P
begin
  O' = AdaptedDalgorithm(f,O)
  control-points = FindAffectedControlPoints(f)
  markedHfg = markHfg(markedHfg,control-points)
  P' = set of FAB/PAB affected in propagating O'
  if null P' return
  else propagate-data(P',O',markedHfg)
end
```

5 Experimental Results

Although we do find that for each Select module in the circuit, we need two additional multiplexers, a C element and a transition latch in order to use our technique, the number of Select modules in a circuit are very small compared to the size of the circuit. Therefore the overall circuit overhead as a percentage of the size of the whole circuit is very small. The pin overhead if we were to assign a different pin for each extra signal to the chip would be 5 pins. Since some of the signals will not be used at the same time, we feel that the number of pins can be reduced. The number of gates for each circuit is given in Table 1. In the same table, we find that the control sequences to test the control part is very small for even large circuits. This is a very important saving in our method. Since the paths cover all the control nodes,we are able to use very few sequences before we are sure that the whole control path has been tested. The number of gates shown in the table correspond to the total number of two input AND/OR gates. We also note that for purely control circuits, such as the qr42, the tests are extremely simple involving only a few control sequences.

6 Conclusions

There are two major contributions to this work. The first is a set of a algorithms to completely test self-timed circuits based on two-phase transition signaling and macromodules and the second is the idea of a scannable SELECT mdoule which makes the circuits *fully* controllable and observable. Our algorithms are based on a Petri-net like model of the physical circuit and are general in the sense that they are compatible with other asynchronous behavioral synthesis approaches like Brunvand [4], Ebergen [6] and any circuit which yields an NHFG. Our approach also reveals an intersting relationship between testability and self-timed synthesis via partitioning the hardware into control, function and

predicate action blocks.

Our future work in this area will include studying the relationship between data-bundling constraint and path-delay fault testing and investigating techniques for on-line testing of self-timed circuits produced by the SHILPA system.

References

1. AKELLA, V. *An Integrated Framework for the Automatic Synthesis of Efficient Self-timed Circuits from Behavioral Specifications.* PhD thesis, Department of Computer Science, University of Utah, 1992.

2. AKELLA, V., AND GOPALAKRISHNAN, G. SHILPA: A High-Level Synthesis System for Self-Timed Circuits. In *In International Conference on Computer-aided Design, ICCAD 92* (Nov. 1992), pp. 587–591.

3. BEEREL, P. A., AND MENG, T. H.-Y. Semi-modularity and self-diagnostic asynchronous control circuits. In *Advanced Research in VLSI, Proceedings of the 1991 University of California/Santa Cruz Conference* (1991), The MIT Press.

4. BRUNVAND, E. *Translating Concurrent Communicating Programs into Asynchronous Circuits.* PhD thesis, Carnegie Mellon University, Nov. 1991.

5. DAVID, I., GINOSAR, R., AND YOELI, M. Self-timed is self-diagnostic, 1990.

6. EBERGEN, J. C. *Translating Programs into Delay Insensitive Circuits.* Center for Mathematics and Computer Science, Amsterdam, 1989.

7. HAZEWINDUS, P. J. *Testing Delay-Insensitive Circuits.* PhD thesis, California Institute of Technology, 1992.

8. KEUTZER, K., LAVAGNO, L., AND SANGIOVANNI-VINCENTELLI, A. Synthesis for testability techniques for asynchronous circuits. In *International Conference on Computer-Aided Design* (Nov. 1991), pp. 326–329.

9. MEAD, C. A., AND CONWAY, L. *An Introduction to VLSI Systems.* Addison Wesley, 1980. *Chapter 7, entitled "System Timing".*

10. PAGEY, S., VENKATESH, G., AND SHERLEKAR, G. Issues in fault modeling and testing of micropipelines. In *In First Asian Test Symposium, Hiroshima, Japan* (Nov. 1992), pp. 107–111.

11. SUTHERLAND, I. E. Micropipelines. *Communications of the ACM* (June 1989). *The 1988 ACM Turing Award Lecture.*

12. WILLIAMS, T. E., AND HOROWITZ, M. A zero-overhead self-timed 160ns 54bit cmos divider. *IEEE Journal of Solid State Circuits 26*, 11 (Nov. 1991), 1651–1661.