INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps. Each original is also photographed in one exposure and is included in reduced form at the back of the book.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

UMI

A Bell & Howell Information Company
300 North Zeeb Road, Ann Arbor MI 48106-1346 USA
313/761-4700 800/521-0600

Testing of Delay-Insensitive Circuits Using Protocol Extraction Strategies

A Thesis

in

The Department

of

Computer Science

Presented in Partial Fulfillment of the Requirements
for the Degree of Doctor of Philosophy at
Concordia University
Montreal, Quebec, Canada

July 1996

Abstract

Testing of Delay-Insensitive Circuits Using Protocol Extraction Strategies

Ping N. Lam, Ph.D.
Concordia University, 1996

Delay-insensitive (DI) circuits are asynchronous circuits whose functional correctness is independent of the delays in their components and the interconnecting wires. The environment and circuit module follows a specified usage protocol, sometimes called handshake signals. It has been believed that purely DI circuits are easy to test because any fault will prevent the generation of an acknowledge signal. We show that this is not always true under a general set of components. A Petri net fault model is used to reveal the possibility of critical races/hazards during test. A design for test method using observation points is shown which guarantees 100% fault coverage, and structural theorems are proved which can reduce/eliminate these observation points.

No automated techniques exist so far for reducing the test length of these circuits, in particular, two-phase transition signaling circuits which contain implicit state encoding. Since there can be very few combinational components, classic scan design would not apply. A theory for control point insertion is presented for test length reduction. The major difficulty is in deriving a safe hazard-free test; the specified usage protocol does not directly apply because of the alterations made. Algorithms based on partial order protocol extraction and partial states are presented which produce provably correct test behaviors. Finally, an impossibility result is proved for fault tolerance in the most common class of DI circuit.

Acknowledgments

I would like to thank Dr. H.F. Li, my thesis supervisor, for all his help, patience, and guidance. I have learned a lot from him over the years. His clarity of thought has broken many deadlock situations; without him, I might still be walking around in circles.

Although not the most social person, I would like to thank some of the people I have known at Concordia, including Dr. D.K. Probst, Dr. R. Jayakumar, Dr. S.C. Leung, T. Fu, K.J. Venkatesh, Dr. X. Xia, A. Horvath, and J. Chan.

This research was funded in part by the exorbitantly high taxes paid by Canadian tax payers, in the form of an NSERC scholarship and FCAR scholarship. Hopefully, I will be able to pay some of those taxes back. Thanks also go to the companies which funded a CRIM bursary.

I would also like to thank my parents, brother, and sister for their endless support. Hey, you guys can finally stop asking me when I'll be finished!

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Asynchronous Circuits

Asynchronous circuits have previously been considered difficult to design and test because of the problems in handling races and hazards and the difficulty of guaranteeing correct gate/wire delays for large circuits in the presence of fabrication and temperature variations which can affect circuit delays. A return of interest to asynchronous circuits has been generated recently due to the developments of synthesis/design methods which guarantee circuits free from races and hazards and whose operation is correct independent of gate and/or wire delays. The earlier work on asynchronous circuits is known as the Huffman style of circuits. The model used is a feedback-delay model in which the sequential circuit is modeled as an acyclic combinational circuit with a set of feedback wires and a delay associated with each of these wires. Because of the numerous possibilities of races and hazards, the single input fundamental mode of operation is enforced: only one input to the circuit is allowed to change at a time and the circuit must be

completely stable before an input can change. Under the realistic assumption of gate delays, redundant gates must be introduced to eliminate hazards and a proper state assignment must be selected to avoid races. These circuits have not been popular because of the many constraints; system composition can change the individual modules' environment such that the single input fundamental mode is no longer guaranteed.

The more recent work can be characterized by the use of the input-output mode of operation [13] in which the environmental use of the circuit is explicitly specified and the environment does not have to wait for the circuit to completely stabilize before making an input change; it is only required that some previous set of inputs be acknowledged by the output. Self-timed circuits [75] is an all encompassing term used to describe these circuits, where a circuit consists of modules which communicate via completion signals and whose correct operation is independent of module or wire delays.

The types of circuits can be classified according to the delay model used. The unbounded (finite) component delay with unbounded wire delay model is known as delay-insensitive (DI), the unbounded component delay with zero wire delay is known as speed-independent (SI), and the bounded component delay with bounded wire delay is sometimes called timed asynchronous circuit. DI circuits with isochronic forks [53] are sometimes called quasi-delay-insensitive, where an isochronic fork is a wire branch in which the differences in branch delays are less than any one component delay. If the delays of the isochronic fork is moved and considered as part of the output delay of the corresponding component, then this is equivalent to an SI circuit. However, quasi-DI usually refers to circuits in which only a small number of forks must be isochronic and not all. DI $\subseteq$ Quasi-DI $\subseteq$ SI $\subseteq$ Timed, where a circuit contained in one class can be contained in a larger one by assuming a stronger delay requirement. A common relative wire delay assumption

is in the bundled data convention in which signals from the data wires must be received and be stable before a request signal is received. Another is the complex gate delay model of [17] where the circuit is SI if the combinational block implementation is modeled as an instantaneous boolean function evaluator followed by a delay element.

Two popular signaling conventions are used. The four phase signaling in which request followed by acknowledge lines are raised high and then lowered (return to zero signaling) and the two phase signaling, also called transition signaling, where events are indicated by transitions, regardless of whether they are rising or falling transitions (non-return to zero signaling).

To justify the reason current research is being done in this area, and hence the necessity of asynchronous testing, the advantages/disadvantages of asynchronous designs are compared to synchronous designs:

Speed: Asynchronous circuits have been quoted to be potentially faster because they can operate at average case speed instead of the worst case speed of synchronous circuits where the clock cycle time must be as slow as the slowest path in the circuit (combinational circuit delay between flip-flops/latches, plus setup and hold times, plus error margin). Examples of average case time being faster than worst case time are when computation time is dependent on data inputs such as in ripple-carry adders or when different paths are taken by the control as in [41]. The error margin of the clock cycle time is affected by fabrication, temperature, and voltage variations and the amount of clock skew. As a system grows larger and wire delays increase relative to gate delays, the potential for clock skew increases and the error margin becomes a larger percentage of the total cycle time. However, there are several reasons why many current asynchronous designs have not been observed to be faster. Self-timed circuits incur an area overhead and

hence a speed overhead for completion detection in handshake operations. To handle worst case delay paths which are much longer than the average delay path in synchronous designs, latches can sometimes be inserted to divide that path (pipelining) or latches can be relocated (retiming) to improve the cycle time. H-tree clock distribution structures have been effectively used to reduce clock skew, although not eliminate it. In theory, it takes logarithmic time for a signal to traverse from source to destination of a clock tree, however, multi-level insertion of clock buffers allows for the use of wave pipelining of the clock signal so that more than one wave of clock signals can exist in the tree and consecutive clock signals can reach the circuit in a constant time. Asynchronous designs will have an advantage when there is high variability in delays. Popular fast synchronous designs such as conditional-sum adders have low variability of delays and hence do not translate well into fast asynchronous designs, while several types of hybrid adders do translate well as demonstrated in [25]. The speed advantage of asynchronous design should be seen when new circuit structures are explored which have large variations in delay and low average case performance.

In addition, since asynchronous circuits may consume less power, they can be operated at a higher voltage and speed given the same power consumption. The current trend of synchronous circuits has been to reduce supply voltages to keep heat dissipation at manageable levels. A reduction in voltage gives a quadratic decrease in power consumption, but circuit delays increase rapidly as the supply voltage approaches the threshold voltages of the devices [15]. The voltage in DI designs are easily adjustable during operation without affecting correctness so that a temperature sensor may be used to dynamically adjust the voltage to maintain maximum voltage and speed without overheating. If there can be significant variability in power consumption, the counterpart for synchronous design is to maintain high voltage and dynamically adjust the clock frequency to prevent

overheating.

Area: If 2-rail signaling is used, twice as many wires are required in the data path, as well as a C-element tree to detect completion. Such delay-insensitive designs have a disadvantage in area. Timed asynchronous circuits usually do not have an area disadvantage, such as in bundled data designs, because the data path is identical to that of synchronous circuits. For common pipeline operations, the area overhead is an inserted delay element between latches and one C-element for synchronization. Area overhead increases only if complex control is required. Synchronous circuits in comparison require area for the clock wires and clock buffers.

Power Consumption: It has been clearly demonstrated that asynchronous circuits can require less power than synchronous circuits, such as in [6,7]. This is because many systems operate in "burst mode" where computation occurs in short bursts between idle time. For example, typing in a word processor results in the processor being idle in between keystrokes the majority of the time. Even the input data rate of sampled speech is slow compared to normal processor cycle times [7]. For technologies such as CMOS, almost no power is consumed when a circuit is idle, giving an automatic low-power mode of operation. This compares to synchronous systems which continually toggle the clock lines and consume a large amount of power from the clock buffers. Special circuits can be used to disable the clock during idle times, but is unlikely to have as fast a response time and only applies to selected sections of a circuit. Since self-timed circuits can have an area overhead, more signal transitions may occur for the same computation. Synchronous circuits can have a power advantage if every part of the circuit is running continuously, but this is not usually the case.

Reliability: Synchronous systems suffer from the metastability problem which may occur during arbitration or the sampling of asynchronous input data by a clock. The resulting output can take an unbounded length of time to resolve and can lead to failure in synchronous systems. The problem can be reduced but not eliminated by adding a synchronizer to reduce the probability of metastability occurring to a negligible amount. Asynchronous systems do not have this problem because they can wait for unbounded periods of time before continuing execution. Self-timed systems are more robust to fabrication/temperature variations and power supply voltage changes because of insensitivity to delays. Asynchronous circuits may be more susceptible to noise because it continuously accepts input transitions. Noise occurring at the combinational circuits between clock cycles do not affect synchronous circuits, as long as the input value has stabilized to the correct value before the next clock transition. Synchronous systems can have a larger problem with electromigration where current is synchronized to be consumed at the same time when the clock makes a transition, compared to asynchronous systems where power consumption is averaged out in time. This can be solved by using thicker power lines.

Composability: Asynchronous systems allow components to be easily replaced and result in an immediate speed increase. The replacement of components with faster ones in synchronous systems does not increase the speed unless the global clock is changed which cannot reliably be done without a global analysis of the new critical paths.

Delays affecting design time: Design time is largely affected by the type of asynchronous circuit used. In terms of design time to ensure correctness, circuits which have greater delay insensitivity require less design time; automated routers can be used without regard to the length/delay of wires. Bounded delay circuits require custom design to enforce relative delay requirements. In synchronous

systems, a proper clock distribution is required to minimize skew and the minimum clock cycle time needs to be calculated. Clock distribution becomes more difficult as systems grow larger and wire delays increase due to scaling. In terms of performance, the delays of synchronous systems are easier to optimize because the system is clearly partitioned into combinational and sequential components, and the worst case path identified. Bottlenecks in asynchronous systems are more difficult to identify; the longest delay path may not be important when one wants to optimize the average case speed.

Yield: Asynchronous designs can have a higher yield. A delay fault in synchronous designs can force the entire system to operate at a speed below the specified requirement because of worst case operation, while a delay fault in asynchronous designs may appear in a less commonly used part of the system and not significantly decrease the average case speed.

Overall, asynchronous designs can have an advantage in power consumption over synchronous designs, have advantages in composability and yield, have both advantages and disadvantages in reliability, have some disadvantage in area depending on the circuit type, and are comparable in speed so far but should improve as new circuit structures optimized for average case performance are explored and designs become as advanced as that seen in synchronous designs.

1.2 Previous Work on Testing

Synchronous circuit testing is a well known area. By far, the most popular fault model used is the single stuck-at fault (saf) model which models a wire node as either stuck-at-0 (sa0), fixed to logic 0, or stuck-at-1 (sa1), fixed to logic 1. It has been shown in practice to accurately reflect real faults such as opens and shorts. Other fault models are possible, such as multiple saf, bridge faults, or faults specific

to a type of circuit such as for PLA's or RAM memory. Work has centered around testing combinational circuits, with the D-algorithm and variations of it being the most popular. Sequential circuit testing is more difficult since its behavior depends on its past history of inputs; a sequence of test vectors is required to detect one saf compared to a single test vector required for detecting one saf in a combinational circuit. In addition, the test requires that the initial state is known to obtain a predictable output; however, a saf can change the state such that the intended initial state cannot be reached. Because of the difficulties, heuristics are usually used to obtain a high fault coverage, or design for testability techniques such as scan design or built-in-self-test is used.

1.3 Testing of Classical Asynchronous Sequential Circuits

Like synchronous circuits, asynchronous sequential circuits are just as difficult to test, if not harder, because of the increased possibility of a saf causing a race or hazard. Asynchronous circuits also may contain more redundancies which are required for masking hazards. By definition, a saf on a redundant node is not detectable (is fault tolerant), but we would still want to check whether a saf exists on a redundant node if it was originally introduced to remove a possible glitch. In this case, the only possibility is to introduce a control point to make that saf detectable (some of the newer DI strategies do not require redundant gates).

1.3.1 Without Design for Testability

The work by Cheng et. al. [16] appears to be a fair indication of the current status of testing asynchronous (as well as synchronous) sequential circuits without design for testability. They use a heuristic (randomizing local search algorithm) for generating the set of test vectors. Starting from an initial random vector, subsequent vectors are obtained by checking vectors which differ by a unit Hamming distance. The vector selected is the one with least cost, based on a cost function which

determines how far the vector is from detecting a fault. The cost is obtained by a simulation of each vector under faulty and fault-free conditions. Since a saf can cause a race or hazard, a simulator which is capable of handling gate delays and races is used. The unit Hamming distance is equivalent to operating under single input fundamental mode and reduces the likelihood of a race or hazard during the test. If the test generation is not complete, redundant faults cannot be distinguished from those not detected due to incomplete search. Although the procedure is a heuristic, the repeated use of a race simulator is still time consuming.

The earliest work on asynchronous circuit testing considered the problem as an identification problem of sequential machines, finding a test sequence which uniquely identifies a fault-free machine and then using this sequence as the fault detecting sequence. Kohavi [40] is one example of this method. The test is composed of two parts: the first part distinguishes between the fault-free n-state machine and all other machines with k<<n states, and the second part distinguishes the fault-free n-state machine from all the transformed n-state machines due to some fault. But generally, obtaining these distinguishing sequences was inefficient using state tables and was tractable only for small circuits.

A common method for handling cyclic sequential circuits is to cut the feedback loops and transform the circuit into an acyclic iterative combinational circuit, such as in Putzolu and Roth [65]. The problem then becomes one of detecting multiple saf's in a combinational circuit and algorithms such as the D-algorithm can be used. But besides requiring much backtracking due to increased depth of the combinational circuit, races and hazards which can appear in the cyclic circuit may no longer appear in the transformed acyclic one.

1.3.2 With Design for Testability

Scan Design:

The asynchronous circuit implementations of [32,44,82] use flip-flops and combinational circuits. The flip-flops are converted so that the popular scan test design can be used, ie. under test mode the flip-flops are connected in the form of a shift register; values are shifted in, one clock cycle is executed, and the result is shifted out. Testing is thus done synchronously and combinational tests such as the D-algorithm can be used. Besides the usual drawbacks of scan design (area/speed overhead in the altered flip-flops and time to shift test data in and out), the asynchronous functions of the flip-flops and logic feeding the asynchronous inputs are not tested. Clock lines must be added, compared to synchronous scan where the clock lines can be reused. If the circuit contains hazards or races, then it will not be detected since the test is done synchronously, eg. a glitch coming from the combinational circuit may not be detected because the clock samples the output only after certain time periods and the glitch can stabilize to the correct value before the sample occurs. Under asynchronous mode, the glitch can cause a fault; hazards must therefore be taken care of by adding redundancies, control points, and use of delay-fault testing (the micropipeline [79] design methodology does not have this problem; even though it is asynchronous, the latches are configured to sample data in a similar manner to synchronous designs). As well, the additional logic adds a delay to the flip-flops which can affect the functional correctness if this delay is not considered when switching from synchronous test mode to asynchronous operation mode. Some asynchronous design methodologies have high ratio of storage elements to combinational gates which can make scan design impractical.

Sakashita et. al [70] uses a slightly different scan strategy in which the existing

circuit blocks are not modified. For each input to a circuit block, one special shift register latch is inserted, with all latches chained together. The latches can be configured to output four types of waveforms which are sent into the circuit blocks, allowing for tests which include delay fault testing. The disadvantage is that the area overhead is high if large circuit blocks are not used, but the number of test vectors required would increase for larger circuit blocks.

Susskind [78] achieves scan design by augmenting the state table during the design process. Li and Hollaar [48] and Spaanenburg et. al. [77] use a similar method for one-hot asynchronous circuits (one flip-flop per state) in which the state/flow table is modified during synthesis. Other circuit specific design for testability methods is for asynchronous counters [14,80].

1.4 Recent Work on Testing Self-Timed Circuits

Self-timed circuits appear to be more easily testable than classical asynchronous sequential circuits. Due to the use of request/acknowledge protocols, a saf on a wire should surface at the environment interface of the circuit as a missing acknowledge signal. Moreover, a restricted environmental use can allow a test sequence to be directly derived from the circuit's usage specification.

Testability of STG Circuits:

Beerel and Meng [3,4] address the testability of SI circuits. Two saf models are used, the output saf (OSAF) and input saf (ISAF) model, Fig. 1.1. The OSAF model considers faults only at the output of components, ie. forks are not considered as components so saf's at the output of forks are not covered. The ISAF model is the more general one and considers faults at the inputs of components. Since the intention is to model saf's at every node in the circuit, as is usually the case for synchronous circuit testing, the fork is considered a component in the ISAF model to cover the input of the fork. The specification model used is STG (Signal

Fig. 1.1: OSAF and ISAF models.

Transition Graph) which is an interpreted Petri net, ie. Petri net transitions are interpreted as positive and negative signal transitions of the circuit.

The model used for proofs in [4] is ESG (expanded state graph) which is a finite automata in which each state is labeled with a bit vector representing the value of every gate output. The claim made is that deterministic hazard-free SI circuits are self-diagnostic, ie. any saf in a self-diagnostic circuit halts its operation. The proof is based on showing that (i) all cycles of states in the valid ESG and invalid ESG must contain all signal transitions, and (ii) any saf causes no cycle of signal transitions to be possible in the ESG, thereby causing the circuit to halt. However, as stated in [3], the result applies for the OSAF model, since a saf at the output of a fork may not halt the circuit as shown in [52]. In addition, the result applies to a particular set of single output components, since we will show in this thesis that a saf not on a fork output does not always halt a DI circuit and hence SI circuit as well since DI is a subset of SI. [3] also shows that SI circuits are self-checking for a subclass of saf under the ISAF model, for saf's at the output of forks which are not isochronic. Their proof results apply to a subclass of SI circuits which satisfy the ***single up/down transition restriction*** and the ***construction assumption***: a strongly connected circuit composed of only AND/OR/C-elements with possible inverters attached to gate inputs. A CAD tool is described which tests whether a circuit is SI and so determines the circuit's testability under the OUPUT SAF model.

The testability of asynchronous timed control circuits is addressed by Beerel and Meng in [5]. The single input fundamental mode is used in testing so that races/ hazards are minimized. The main result of the paper is a sufficiency condition for single saf testability (ie. whether all saf's are testable) and an automated testability checker. The sufficiency condition is given without proof and it is stated that the condition must be strengthened.

Work at Caltech:

Martin and Hazewindus [31,52] address the testability of DI and SI circuits. They demonstrate that some control points are necessarily required for testing some saf's in SI circuits. The specification model said to be used in [52] is a set of partial orders of transitions on the I/O ports of the circuit, although the notation used is a linear handshaking expansion. The modeling of the circuit is in the form of a set of production rules. A production rule has the form $G \rightarrow S$, where G is a boolean guard expression and S is a set of actions to be performed. A production rule fires when its guard becomes true. For example, an AND gate with inputs a, b and output x is modeled by two production rules $\{ a \wedge b \rightarrow x\uparrow, \neg a \vee \neg b \rightarrow x\downarrow \}$ where the arrow denotes a positive or negative signal transition. Under fault-free conditions, all production rules are non-interfering (guards of rules are mutually exclusive) and stable (if a guard evaluates to true, it remains true until the rule fires). The existence of a saf modifies some production rules so that these properties are no longer true. It is shown that most saf's can be easily detected and that the test can be derived from the usage specification. Saf's which are undetectable require a control input.

According to [31], an undetectable fault can exist when testing a production rule R which may fire prematurely due to a saf; a fault is declared untestable if there exists an unstable production rule going from (i) the initial state to R, and (ii)

from R to the final state where the premature firing is observed at the output. An unstable production rule implies a race condition. This appears to be too strong a condition because a saf can still be detected even when there is a race. To obtain the test sequence, the use (handshaking expansion) is executed once and the result is then fed through a simulator to check fault coverage. Some heuristic is then used to try to cover the remaining faults. Since a saf may cause a race or hazard, a race simulator is required if one wants to determine exactly whether the saf will be detected by the test, and such simulators can be computationally expensive [74]. Faults which cannot be covered will require the insertion of control points; a control point here means the introduction of a channel, the creation of a disconnection on a wire and making it a primary input/primary output. Normal operation will connect this primary output to the primary input. For such control points, if a shift register is used to input values to the control points, then it is possible for the circuit to pass a test but not function under normal operation if a fault prevents the switching from test mode to normal mode.

Since there is no redundancy checker (this may be difficult for such sequential circuits; a redundancy check for combinational circuits is NP-complete [26]) undetectable faults can be confused with redundancies. For example, the untestable fault in the two connected D-elements of [52] appears to be actually a redundancy (a saf on the node does not affect the usage behavior). Control points can then be unnecessarily inserted.

Design for test at the programming level:

Work done at Philips Research Laboratories [69] is based on the compilation of a CSP-like programming language into a quasi-DI circuit. The test circuitry is designed and inserted at the programming language level. The scan and partial scan design is written as part of the program and scan channels are explicitly

inserted. The test is fully asynchronous and there is no synchronous test mode involved in the scan. The method is different from other strategies where the circuit is designed first and then design for test is added afterwards. The advantage of such a method is that knowledge of the system architecture and algorithm is available and the programmer can decide which parts require additional aid in testing. However, the method requires hand design of the test portion of the program. Automatic optimization of the test length and minimization of test area is not available. What is normally done by computer in most test strategies must be performed by the user. Test design for complex control circuits may be difficult without a detailed simulation, and evaluation of the quality of test improvement can be difficult if done by hand. The method uses the OSAF model and assumes that a saf will always lead to a deadlock or timeout at the external output. Observability is then ignored and test design is based on improving controllability of the circuit. Since multiple output components can sometimes be used, it is not clear that the result of Beerel and Meng [3] also apply here where a saf always causes a circuit to halt for a certain type of single output components under the OSAF model.

Delay-fault testing:

Timed asynchronous circuits can produce more area efficient circuits when the user is allowed to enforce certain bounds in the delay of gates/wires through the insertion of delay elements, changes in the layout, or optimization of gate speed. But the testing of such circuits is more difficult compared to DI designs and the yield can be lower because delay faults must also be tested after manufacture. An increase in delay at some location does not affect the functional correctness of DI designs but does in a timed circuit. Even isochronic forks in SI designs may require delay-fault testing if the fork is large; this was demonstrated by the existence of such a fault in an asynchronous microprocessor [54] where there was a malfunction for certain delays in the memory interface. Any design methodology

which requires the addition of redundant gates to eliminate hazards must be a timed circuit and as a result requires delay-fault testing. Since the redundant gate is used to mask a glitch caused by a particular signal transition path, the delay requirement is that the gate must be faster than the delay on that path, ie. the redundant gate cannot have an unbounded delay. Since a saf at the input of a redundant gate by definition is not detectable, some design for test strategy such as insertion of control point is required to obtain 100% fault coverage, in addition to the delay-fault testing.

Keutzer et. al. [37] describes a delay-fault testing method for timed asynchronous circuits which is similar to that done for synchronous designs. It uses the path delay model ([36] also extends the result to the less accurate but more efficient gate delay model) which checks that the bounds for all delay paths are met by sending a signal transition through every possible path in the combinational circuit. Full scan design is used, and the scan elements (eg. RS flip-flop, C-element) must be altered to store two bits instead of the single bit used in normal scan designs. Two bit vectors are shifted into the elements, the first to initialize the value at the input of the combinational gates and the second to create a signal transition. Redundant gates require additional control points and their associated latches if their inputs are shifted in. Path delay-fault testing is usually expensive since there is potentially an exponential number of paths to test, however, their target circuits are not too large and usually are two-level. Since the test mode is synchronous, the asynchronous parts of the storage elements will not be tested in the scan test.

Lavagno et. al. [42] presents a delay-fault testing strategy in which the combinational logic of the original design is not altered, ie. additional control points do not need to be inserted at the redundant gates. It is based on the theorem that any two-level circuit implementing an unate function is automatically prime and irredundant if it is free from Single Cube Containment. The circuit is

transformed into an unate one through variable phase splitting, ie. each input variable and its complement, implemented as a fork with inverter, is converted into two separate variables which are then tested individually so that the originally complemented variable does not always have to contain the complemented value. The area overhead is that special scan flip-flops are required that can drive both phases of each signal separately (outputs Q and $\overline{Q}$ are separately controllable), and the primary inputs containing complements require such a special scan flip-flop inserted as well (which is transparent under normal operation). The method may give lower overhead than conventional control point insertion which increases the number of inputs with the corresponding gate size increase and additional flip-flop to store the value of the control point.

Micropipelines:

Since micropipelines [79] has become a popular design structure, several papers exist on testing them [29,38,58,59]. Khoche and Brunvand [38] describes a scan path method which tests both the combinational and control logic. The main difference between this method and conventional scan design is that it eliminates the global scan clock and uses normally existing lines for clocking. The C-elements are modified such that a clocking chain is formed through all the C-elements with the acknowledge out line being the clock input. The latches are also modified to include serial in and out lines which are chained together so test vectors can be shifted in. Under scan mode, they become master-slave scan latches. Under normal operation when the pipeline is empty, data placed in the first stage of the latches would immediately propagate to the last stage. When testing, the state of the pipeline is set to appear full so that test vectors only pass through a single stage of combinational logic. Test speed is increased by clocking the latches in the reverse direction from the data flow which bypasses all the delay elements.

Delay-fault testing of micropipelines requires the checking of all paths in the combinational logic between two stages to ensure that the combinational delay is always less than the bundling delay (largely controlled by the delay element). The bundling delay between each stage can be different. Delay-fault testing can be performed in the same way as [36,37], but this would require the introduction of a global clock and the overhead of a second bit storage in the latches. Instead, [38] uses a method which has been used in synchronous design where the first vector is shifted through the scan path and the second vector is derived from the combinational block in the previous stage, ie. given the pipeline $v : B : v1 : T$, where T is the combinational block to test, B is the combinational block preceding T, and v, $v1$ are vectors that have been shifted in, then the second vector will be $v2 = B(v)$. The combinational blocks must be able to produce all possible combinations of outputs in this method and is ensured during synthesis.

Petlin et. al. [59] shows an alternate method using built in self test which is very similar to that used in synchronous designs. An asynchronous pseudorandom pattern generator is used for generating test vectors and an asynchronous signature analyzer (register) is used for storing compressed results. Like synchronous designs, its advantage is high speed on chip testing without the serial vector shift time, at the cost of higher area overhead without the guarantee of 100% fault coverage.

Internal fault coverage of components:

C-elements are commonly used DI components in asynchronous design. Since the size of C-elements are larger than implementations of classical gates, a question arises as to how many internal faults at the transistor level are covered by exercising only the external inputs. The coverage of internal faults is highly dependent on the particular implementation of the C-element. Brzozowski and Raahemifar [12] makes such a study on several known C-element implementations.

They show that depending on the circuit implementation, from 14.7% up to 40% of single saf's do not cause the circuit to halt (is not self checking). They show that a test of length seven is required to detect all detectable single faults. For some implementations, it is possible for a saf to cause oscillation or destroy the speed-independence of the circuit. The tests are serialized (only one input changes at a time). It remains to be seen how such tests can be implemented for C-elements which are internal to the circuit, since normal operation will generate concurrent inputs to C-elements which are not directly serializable. It should be possible to design C-elements or other components which maximizes their internal fault coverage given some external test.

Fault tolerance:

Fault tolerance is an area related to testing in which a test does not need to perform 100% fault coverage to guarantee correct functionality. Whereas the redundant logic introduced to eliminate hazards needs to be tested by introducing control points (otherwise a saf in the redundant logic may allow the hazard to appear during operation), faults in the redundancies of fault tolerant circuits do not always need to be tested. Although every redundant node can be tested by insertion of control/observation points, the area and speed overhead makes this infeasible. A functional test is generally required at the circuit level, or else individual modules can be tested separately at a higher level.

Fault tolerance can be introduced in timed asynchronous circuits using largely the same methods as in synchronous designs. At the higher system level, methods such as triple modular redundancy can easily be implemented by taking three identical modules, synchronizing their outputs, and comparing the results, with a timer required to generate a timeout if one of the modules does not generate an acknowledge due to a saf. Some speed is sacrificed to implement such fault

tolerance since we must wait for the worst case delay of the three modules. Timeouts are always required when results need to be synchronized and compared. An alternative is to take the first result out of the three modules which is received (OR result rather than AND). This requires that the modules be completely self-checking with excitory faults not generating any outputs. Since the other two modules have not generated outputs yet and are not stable, the timing requirement is that these two modules generate their acknowledge before the next input arrives. Taking the OR result only requires an arbiter to select the first result and is faster since it does not need the overhead of synchronization and comparison of results. However, the faulty module must then be switched out since continued input to the faulty module may produce unknown outputs. Other alternatives which exist in current fault tolerant design are to use dynamic redundancy or hybrid redundancy in which active and spare modules are used.

Self-timed designs incorporating fault tolerance has appeared in [2,24]. Fifield and Stapper [24] implements a fault tolerant memory chip using an error-correcting-code design and DCVS logic which can generate completion signals. Allen et. al. [2] describes an adaptive router and algorithm which tolerates faults in links and nodes. However, their implementation did not yet include the fault detection circuitry. Rennels and Kim [68] describes the use of a timeout circuit to detect acknowledges which are not generated and is used in concurrent error detection, detection of faults during execution. They use checker circuits instead of C-elements for the acknowledge tree.

There have not been any results on the existence/non-existence of fault tolerance in completely DI or SI circuits. Is a timer or some timing assumption always required?

1.5 Contribution of Thesis

The class of circuits we concentrate on is transition signaling DI circuits with no restrictions on the set of components used, only that they be atomic, ie. we can only observe the external DI interface of the components; their internal implementation cannot be observed and may use delay compensation to realize the external DI behavior. Gate level testing is used: saf's are considered only on wires between components and not internally at the transistor level. Even though the DI components are usually larger than classical AND/OR gates, we do not consider internal faults to limit the complexity of testing large scale circuits (not all faults internal to NAND/NOR gates may be detected by a combinational test either). The more general ISAF model is used.

Testability of DI and SI circuits:

A major difference between the results in this thesis and previous results is that we consider the testability of general DI circuits which include multiple output components. Several of the previous proof results have been restricted to a certain set of components, in particular single output components, or the OSAF model. Our results are independent of the set of components used; we show that some important well known theorems in asynchronous testing are not true when they are extended to include multiple output components. The theorem that SI circuits are self-checking under the OSAF model [3] is shown to be false under a general set of components. Many sources including [31,33,52] have stated that DI circuits are self-checking under the ISAF model. The reasoning is that every signal transition in a DI circuit must be acknowledged by the receiver of the transition, including all outputs of a fork (only a single output needs to be acknowledged in a fork of an SI circuit). This is called the acknowledgement property in [31,52]. Any saf then would halt the circuit. This implies that all DI circuits are very easy to test. However, we show that

this theorem is false for the general case with multiple output components. Several counterexamples to these two important theorems are shown in the thesis, eg. Figs. 3.3, 3.4, and 4.10. Whereas [31,52] showed that design for test is necessary for some SI circuits to achieve 100% fault coverage under the ISAF model, we show that it can also be necessary for purely DI circuits. This is because the acknowledgement property of all transitions on every node being acknowledged is true only for the fault-free circuit. In the faulty circuit, a saf can cause a critical race condition and cause this property to be false.

It can be justified that the set of DI components considered for testing should include multiple output components because it has been shown that the set of realizable DI behaviors using only classical single output gates (with fork included) is very small [13,45,47,53,73]. Leung and Li [45] give a classification of the properties of DI components and the behavior realized by such networks. In particular, it is shown that classical gates (including the C-element) lack properties which exist in some multiple output components and that to obtain more general realizable behaviors, such multiple output components are required. A set of components is shown to implement any determinate finite state DI system. Synthesis of very general DI behaviors appear in [23,46,47].

Modeling:

A Petri net fault model is introduced in the thesis which simply and clearly models saf's and their behavior in transition signaling circuits. It is at a higher level than using state graphs to specify faults, thereby simplifying proofs and analysis. Unlike previous methods of modeling faults for SI/DI circuits such as [31], races and hazards are clearly specified. The entire system is consistently specified using the same Petri net model, including the system specification, circuit network, faulty network, system environment, and test environment, and allows their composition.

This has advantages compared to the popular STG specifications which only models the fault-free specification; the faulty behavior, circuit network, and test requires a different model. In particular, there exists a simple procedure to unfold the Petri net into a partial order behavior which makes analysis and algorithms less complex. Since the size of partial orders do not suffer from the state explosion problem due to concurrency compared to state graphs, algorithms can exploit the fact that not all states must be checked.

Fault tolerance:

We answer an open question: there does not exist fault tolerance in general DI circuits which contain multiple-active nodes, nodes which are exercised two or more times during normal operation. We also show that there can exist fault tolerance for DI circuits with inactive and single-active nodes.

Structural Theorems:

We explore more precisely when a saf is detectable and not detectable when there is no design for test. [31,52] demonstrated cases for SI circuits, in particular at some outputs of isochronic forks. Previous results require race simulation of the test on the circuit in order to determine whether the saf is undetectable for that test. We show a method for DI circuits which can avoid race simulation by using structural theorems instead. It has a major advantage in that structural checks are simple, many requiring time linear to the size of the circuit, compared to race simulation which can be an NP-complete problem for most race models [74].

The at-least-2 test is used which exercises every node twice in the fault-free circuit. A major proof result is given: the at-least-2 test is sufficient to detect all saf's in any DI network consisting of a set of CRF (Critical Race Free) components. This implies that circuits containing these components will always be self-checking. Because of the possibility of race conditions, the proof requires a much more

complex argument than the acknowledgement property. Inclusion of an additional component such as a demultiplexer may cause a critical race condition to occur and lead to an undetectable saf. Various structural theorems are explored, and its limitations are revealed through counterexamples.

Sensitization:

In previous general methods of testing SI/DI circuits [3,4,31,52], the test is obtained by running the normal usage protocol until every node is exercised twice. Without design for test, there was no method to improve the fault coverage if there were undetected faults (undetectable using the given at-least-2 test but not undetectable because of some uncontrollable race or a redundancy). We introduce a strategy which can improve the fault coverage by extracting tests which do not have to directly follow the usage protocol. The method is based on the sensitization of a certain path from the saf or the spurious token/transition caused by the saf to some external output. Path sensitization has not been used in testing protocol based asynchronous circuits, unlike synchronous circuit testing which is mostly based on sensitization such as the D-algorithm. The difficulty is that arbitrary injection of inputs would violate the input protocol and is very likely to result in races and hazards. The trick we use is to extract the test behavior from the original usage protocol, ie. the partial order behavior. Extraction from the structure of the circuit is nearly impossible because of the explosion of behaviors as a result of being a sequential rather than a combinational circuit. Segments of the partial order can be extracted and combined to form a test and since the segments follow the original protocol, races/hazards do not occur. However, since the segments are executed out of order, a method must be used to prevent hazards from occurring as the segments are executed. The concept used is the partial state on the Petri net/ partial order. As partial order test segments are executed, only part of the state relevant to the test needs to be checked. The rest of the global state may contain

hazards, as long as it does not reach the partial state under consideration for the test.

Design for Test: Control Point Insertion:

The most commonly used design for test strategy, scan path design, is mainly used to test combinational gates within a sequential circuit; this cannot be efficiently used in DI transition signaling circuits as well as other asynchronous design methodologies [54] because of the low number of combinational gates used. For such circuits, no automatic design for test strategy exists so far. Martin and Hazewindus [31,52] used control/observation point insertion to make saf's testable but not to decrease the test length. Roncken [69] can decrease the test length through the hand design of the test program/circuit, but this can be very difficult for complex control circuits. Control point insertion is a method which can decrease the test length, but the determination of exactly where to insert it to optimize the test length and what the new input protocol will be has been an open problem. This design for test problem is much more difficult than scan design which simply alters the storage element and performs a combinational test using existing algorithms. Since the inserted control points change the circuit, the original usage protocol cannot directly be used. Some form of intelligence is required to identify when inserting a control point will be useful for reducing the length of the test. The most obvious location for insertion would be at the node which is least used after being determined by simulation. However, this does not exercise the node immediately preceding the control point which is also least used.

The strategy proposed can be seen as the compaction of an at-least-2 test which cannot be shortened further without control points. Gaps are first identified in the partial order which are not necessary. Unnecessary gaps are ones containing more than two events of the same node and are not required to satisfy the at-least-

2 property. The remaining partial order segments can then be executed, possibly out of order, to complete the test. The requirement is that the partial states between segments have to match. Unlike the strategy of first identifying control point locations and then trying to find a hazard free protocol involving those points, we insert control points to aid in the matching of partial states. The result is not obvious and avoids the extremely difficult problem of extracting a hazard free behavior from an altered circuit.

Internal fault coverage of components:

Our test strategies do not consider internal faults and assume component implementations which have good internal coverage given an external at-least-2 test. It is possible to serialize inputs to C-elements by inserting pass gates at the inputs, disabling one input at a time. Assuming that this problem can be solved, our method of control point insertion would be orthogonal to solving this problem; that is, nodes which are difficult to exercise are still difficult to exercise if inputs to C-elements are serialized. However, an actual test of C-elements may not be as problematic as [12] implies, requiring a test length of seven (an at-least-2 test has length of four when serialized). For deterministic control circuits, the serializations received by C-elements under test are likely to be the same as the serializations under normal execution because the circuit layout and delay paths are fixed. Even though a C-element can cause a fault under a particular serialization, this faulty serialization may not be reachable under the given delays of the layout.

1.6 Contents of Thesis

Chapter 2 introduces the Petri net specification and fault model and how the net representation of systems can be easily composed. An example set of transition signaling components which are commonly used is given, based on the components used by [23,47]. The possible types of test modes are classified. A

timed Petri net model is given, since a test is not DI and must have fixed time intervals between test vectors. A method of calculating the exact minimum time interval required is presented based on unfolding of the timed Petri net.

Chapter 3 states the correctness condition for a test. The problems with testing circuits with inactive and single-active nodes are illustrated. An impossibility proof is given for fault tolerance in DI circuits with multiple-active nodes. The pomset model and the at-least-2 test is introduced. Counterexamples to the sufficiency of the at-least-2 test to detect every single saf is given, for circuits containing multiple output components. An optimal breadth first search algorithm is presented for finding the minimum length at-least-k test for both deterministic and non-deterministic (input choice) circuits.

Chapter 4 describes how observation points can be inserted to achieve 100% fault coverage. Problems at initialization including infinite cycle (oscillation) is discussed. The structural theorems are presented. Detailed and rigorous proofs are given using the trace model and the idea of representation traces. A behavioral theorem is given for covering cases not covered by the structural theorems.

Chapter 5 defines the sensitization problem. The notion of partial states and sensitization region of a test is introduced. A polynomial time search algorithm is presented for efficiently finding the sensitization test. The correctness of the sensitization test is shown by proving that the test is 1-safe (hazard free) and will correctly reveal the fault at some external output. An example of the search procedure on the partial order is given.

Chapter 6 considers design for test and the insertion of control points to reduce the length of a test. A strategy for solving the problem is presented involving three steps: gap detection, gap matching, and tour finding. The problems are proved to be NP-complete, and efficient polynomial time heuristics are given to

solve the problems. The steps are illustrated for a large example circuit.

Chapter 7 gives some additional examples of the algorithms run on DI circuits.

Chapter 8 is the conclusion and discusses possible future research.

Chapter 2

Petri Net Specification and Fault Model

A Petri net specification and fault model is introduced for modeling both the faulty and fault-free circuits. The model can be consistently used to model the entire system and allows for composition of the basic components, faulty and fault-free circuit networks, environment net, module net, and test net. Stuck-at-faults, races, and hazards are simply and clearly specified. The pomset partial order model is used to describe the semantic behavior of a fault-free Petri net. A timed Petri net is shown for specifying and deriving a test schedule, since a real test will not be insensitive to all delays.

2.1 Petri Net Model

A Petri net (P/T-net) is a tuple N = $<P, T, F, M_o, \Sigma, L>$ where $P=\{p_1, p_2, \ldots, p_m\}$ is a finite set of places, $T=\{t_1, t_2, \ldots, t_n\}$ is a finite set of transitions, $F \subseteq (P\times T) \cup (T\times P)$ is a set of arcs, and $M_o : P\rightarrow\{0,1,2, \ldots\}$ is the initial marking. Σ is the finite set of actions which can be either input, output, internal, or null actions. $L: P\rightarrow\Sigma$ is

a labeling function which labels each place with an action. The Petri net follows the usual firing rule: a transition is enabled when a token is contained in each of its input places and a transition fires by removing a token from each of its input places and adding a token to each of its output places. A place $p_i \in P$ of a net with initial marking M_0 is ***k-safe*** for $k \geq 1$ if for all reachable markings, $M(p_i) \leq k$. A Petri net is *k*-safe if each place in that net is *k*-safe.

A high-level Petri net will be used to model the system, while a low-level Petri net will be use to describe internal behavior of components.

2.2 Set of Basic Components

Some of the most common DI transition signaling components are shown in Fig. 2.1. The set {wire, inverter, C-element, XOR, toggle, demultiplexer, arbiter} is used by Leung in [47]. The demultiplexer is also called a Select component. Ebergen [23] uses the set {wire, inverter, C-element, XOR, toggle, RCEL, arbiter} where the RCEL is used instead of the demultiplexer. It has been shown by both [47] and [23] that the class of delay insensitive behaviors that can be implemented with these components is very general, where [23] describes the translation from a regular expression language and [47] describes the translation from a partial order specification. Similar components are also used by [11,79], sometimes with the addition of isochronic forks. [23] also uses an NCEL component, but is not included here because it requires isochronic forks and is SI. However, any isochronic fork can be converted into an equivalent non-isochronic fork DI circuit at the expense of extra area and speed overhead: Ebergen in [23] shows a conversion using a ring of RCEL's, while the "choice recorders" in [47] can also be used to eliminate some isochronic forks.

Each component in Fig. 2.1 is specified by a set of high-level Petri net transitions. These high-level transitions can be directly translated into a low-level

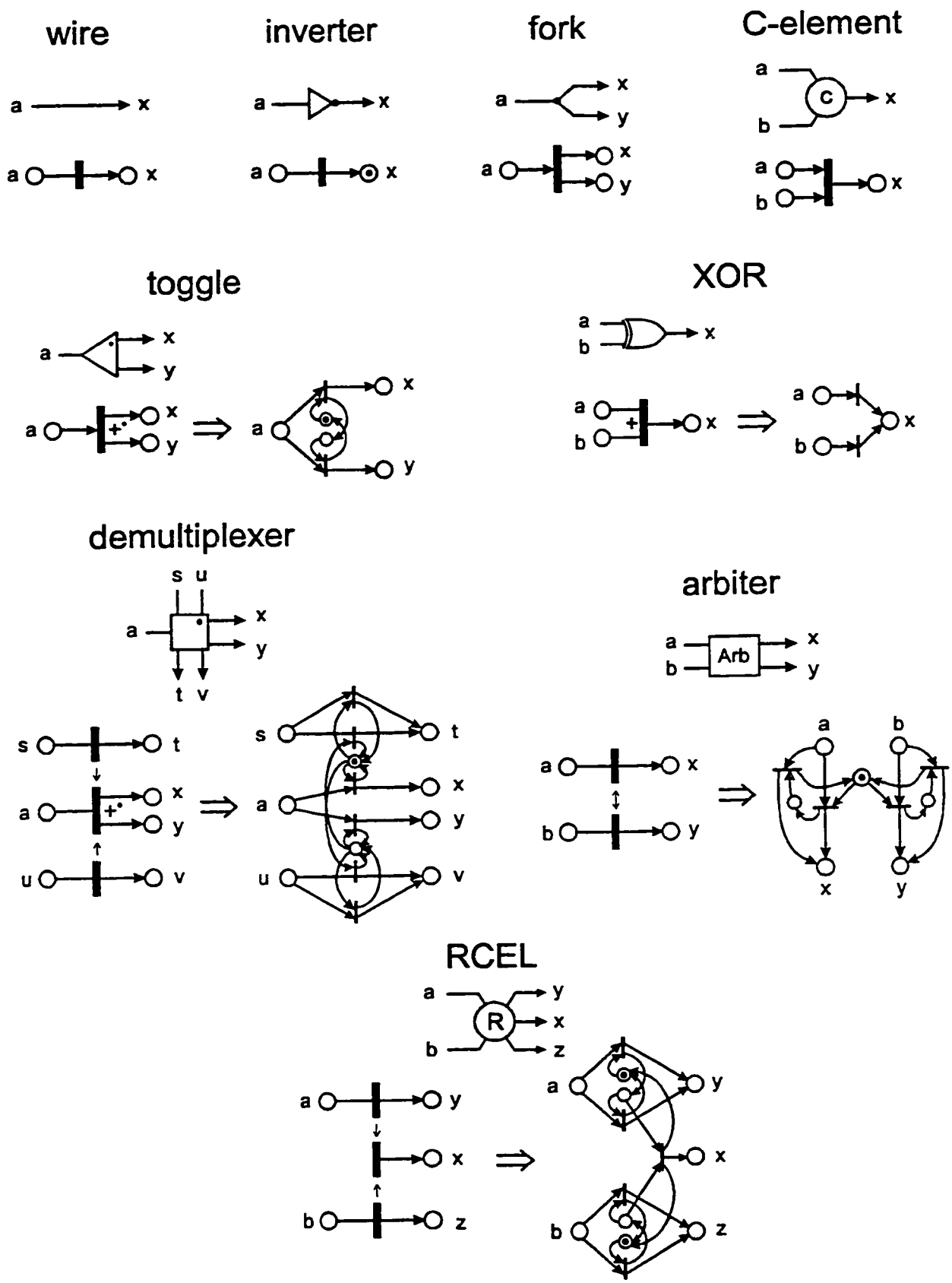


Fig. 2.1: Set of DI basic components.

Petri net (a P/T-net) as shown. We use a high-level net representation to reduce the size of the graph and to be able to formulate structural theorems later on in this thesis, where a test can be proved to detect all faults containing certain sets of high-level transitions and its corresponding I/O ports. The low-level net representation is contained in the most general class of P/T-net because the arbiter, demultiplexer, and RCEL contain symmetric confusion [57]. Analysis of this class of net is usually more complex than for smaller classes such as free-choice nets or marked graphs, but the results of this thesis is based on more efficient structural analysis of the high-level net and avoid complex behavioral analysis.

The placement of a token at a net place corresponds to the firing of a signal transition. Since 2-phase transition signaling circuits are used, there is no distinction made between a positive and negative signal transition. The wire, inverter, fork, and C-element have the same high-level and low-level net representations. The '+' sign in the toggle, XOR, and demultiplexer indicate mutual exclusion of inputs/ outputs when used correctly. The dot in the toggle and demultiplexer indicates initial state. For example, the arrival of tokens at the a input of the toggle produces tokens at x and y alternately, starting at x. For the XOR, the arrival of a token at a or b produces a token at x. The fork used here is DI and does not make the isochronic fork delay assumption [53]. The demultiplexer contains a set of control lines $\{s, t, u, v\}$ and a set of data lines $\{a, x, y\}$. There are two relevant states in the demultiplexer. The control lines set/reset the state, and a token at the a input will be directed to either the x or y output depending on the current state. The small arrow between the high-level transitions indicate that there is an internal effect in the low-level Petri net when the control transition fires. Independently, the control transition behaves like a wire. For the arbiter, we assume that metastability [39] will not occur in the component. This can be guaranteed if the test guarantees that concurrent tokens will not be sent to the arbiter; the test would always serialize the inputs in

some order.

These Petri net specifications describe component behavior under *both* fault-free and faulty conditions. For example, under fault-free conditions, the environment will follow a delay-insensitive protocol and send only a single input token to the XOR, and wait for an output token to be generated before sending in a second input. Under faulty conditions, tokens may be sent concurrently to both inputs (this is input safety violation under normal operation); two tokens will then be deposited at the output place if this were to occur.

The components used are considered to be atomic, ie. the granularity of delay-insensitivity is at the component level; each component operates correctly and has a DI interface. In the actual implementation, there may be hazards internal to the component, but proper delay compensation can be performed internally such that the component's external behavior is DI.

Definition:

A *stable state* is a state in which no internal or output signal transition can occur when the input of a circuit does not change. A component is called a *terminal component* if there exists some component stable state in which an input transition to the component does not produce any output transitions. A component is called a *non-terminal component* if an input transition to the component must always cause an output transition. These terms are defined similarly at the Petri net level. □

The set {wire, inverter, fork, toggle, XOR, demultiplexer, RCEL} are non-terminal components, and the set {C-element, arbiter} are terminal components. The C-element is a terminal component because when both of its inputs are of the same value (contains no tokens at its input places), a token sent to one of its inputs will not result in an output. The RCEL behaves similar to the C-element at its x output, but it is a non-terminal component because an output token is always generated at

y or z when an input is received, regardless of its internal state. The arbiter is a terminal component because under certain internal states, ie. when a request-acknowledge has occurred for one input and a release signal has not yet been sent, a token sent to the alternate input will not result in an output.

Inverters are the only source of internal tokens in the high-level Petri net of a fault-free circuit. At initialization, the output place of the inverter net contains a token. Stable state is reached when these internal tokens have traversed through non-terminal components and stop at the input of terminal components or appear at the environment.

2.3 Module-Environment System Specification

The specification of the system *Ns* is divided into two parts, the module Ns^M representing the behavior of the circuit, and the environment Ns^E representing the expected user behavior of the circuit. There are different circuit nets *Nc* which can be used to implement the behavior of Ns^M. Both Ns^M and Ns^E are *open nets*, ie. places labeled as input actions do not have input transitions and places labeled as output actions do not have output transitions. The output alphabet of Ns^E is equal to the input alphabet of Ns^M and vice versa. Together, the composition $Ns^M \bullet Ns^E$ form a closed system. Composition is performed by co-location, the collapsing of all places with the same action label into a single place, while preserving all transitions and arcs. The circuit network *Nc* is also formed by the co-location of component I/O places. Under fault-free conditions, the net of a DI circuit and its environmental use must form a 1-safe Petri net.

2.4 Petri Net Fault Model

At the circuit level, the fault model used is the single stuck-at-fault (saf) model, where each node in the circuit can have a stuck-at-zero (sa0) or a stuck-at-one (sa1) fault. At the Petri net level, under faulty conditions, the net may no longer be 1-safe or even *k*-safe for bounded *k*. The net fault model is augmented with the additional Petri net rule that when two tokens collide at the same place, the tokens may either cancel each other out or behave normally as two separate tokens. This corresponds to expected circuit behavior where either two signal transitions may cancel each other out when they are traversing on the same wire, or the two transitions do not interfere with each other. This is called transmission interference by Udding in [81].

There are two possible types of saf's: inhibitory and excitory. Suppose all wires are low in the fault-free circuit at initialization. A sa0 shown in Fig. 2.2(a) will appear as an inhibitory fault, corresponding to a disconnect preceding place *s* in the Petri net. The XOR transition behaves as a sink transition. Repeated firings of the XOR transition can never produce an output token at place *s*. A sa1 shown in Fig. 2.2(b) will appear as an excitory fault, corresponding to a disconnect preceding

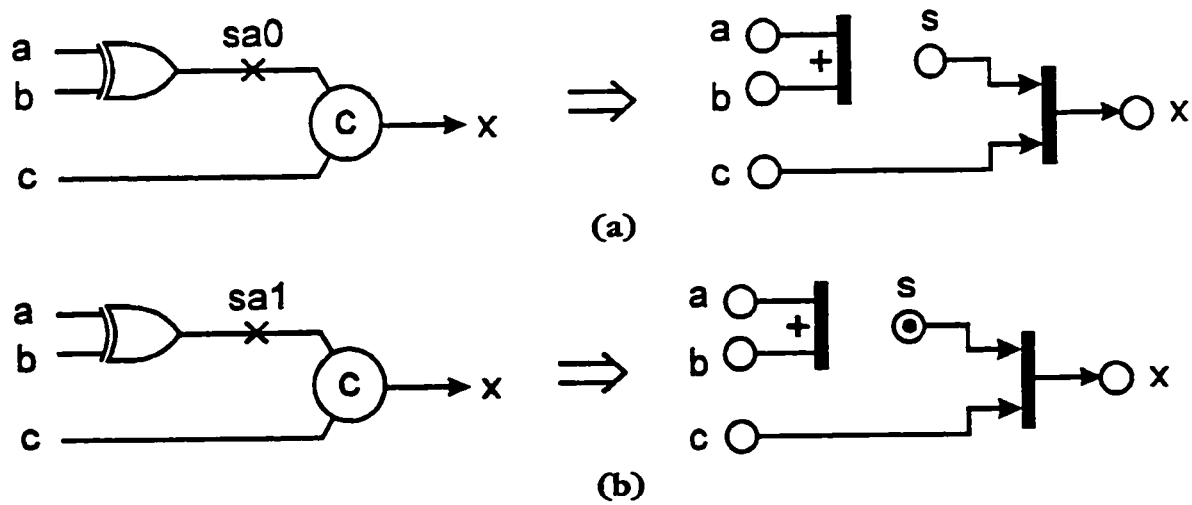


Fig. 2.2: (a) Inhibitory fault, (b) excitory fault.

s with a spurious token deposited at *s*. The sa1 can generate only a single spurious token at the input of the C-element; once the spurious token is consumed, no further tokens can be produced at *s* since no signal transitions can pass through the saf. If the initial state of the wire is high, then a sa0 appears as an excitory fault and a sa1 appears as an inhibitory fault.

The following details what occur at circuit initialization:

(1) Global reset line is enabled.
- Internal state of components are initialized.
- Inputs of state components are held low.

(2) Global reset line is disabled.
- Spurious token and inverter tokens are allowed to traverse through the circuit until they stop at terminal components.

A reset input exists for each state component, {toggle, demultiplexer, C-element, arbiter, RCEL} and a global reset line is connected to these inputs. On enabling the global reset line (eg. on positive transition of reset line), the inputs of these components are forced low, regardless of the current input value, in effect disabling the inputs. When the global reset line is disabled (eg. on negative transition of reset line), the components are allowed to function normally. Since all state holding component inputs are disabled, the spurious token appears at the location of the saf at initialization. When the global reset line is disabled, the spurious token will traverse through non-terminals and stop at a terminal component or arrive at an interface output. The input of combinational components (XOR) do not have to be disabled since their behavior is the same regardless of when the spurious token passes through the component.

Holding the inputs of state components low can be implemented by inserting an AND gate or equivalent transistor level circuit at the input of the state component and connecting the second input of the AND gate to the global reset

line. Fig. 2.3 illustrates this for the C-element. State components need to be reset because their internal state is not known at initialization. A C-element with one inverted input can initialize into two possible states where the next input token may or may not generate an output. In the case of the toggle and demultiplexer, forcing all their inputs low is not sufficient to reset their internal states, so additional internal reset circuitry is required for these components. When transistor level reset circuitry is used, it will be dependent on the particular component implementation and technology. [31] shows that two transistors are required for resetting a CMOS C-element with one inverted input and only a single transistor for non-inverted inputs.

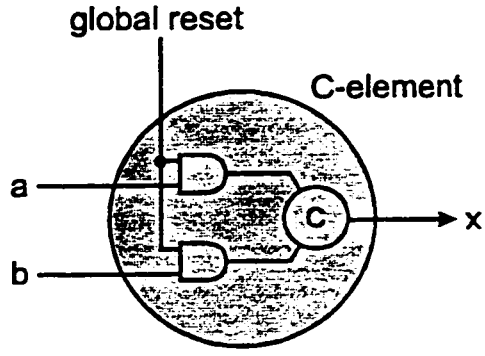


Fig. 2.3: Insertion of global reset for state components.

It is possible to eliminate reset lines at some components internal to the network if we can guarantee that the inputs of these components are always low as a direct result of the environment inputs being low, such as for some C-elements with non-inverted inputs. But under faulty conditions, this cannot be guaranteed because of the possibility of excitory faults; the component can be in an unknown internal state if a reset is not used to disable the effects of the excitory faults. The component reset can then only be eliminated if the test guarantees detection of saf's independent of the initial component state, or if the particular implementation of the component guarantees that its initial state will be the same with and without

a reset even in the presence of excitory faults. We will not explore the elimination of component resets further and will assume that all state components have a reset.

It is also possible for the global reset to force the outputs of components low instead of their inputs, however, under the fault model we will use, this is valid only if the internal implementation of the component is guaranteed to have the same behavior when it is reset with a mix of high and low inputs vs. the component is reset with only low inputs and the high input transitions are sent in afterwards.

Fig. 2.4 illustrates a specification $Ns^E \bullet Ns^M$, a circuit implementation C' of Ns^M

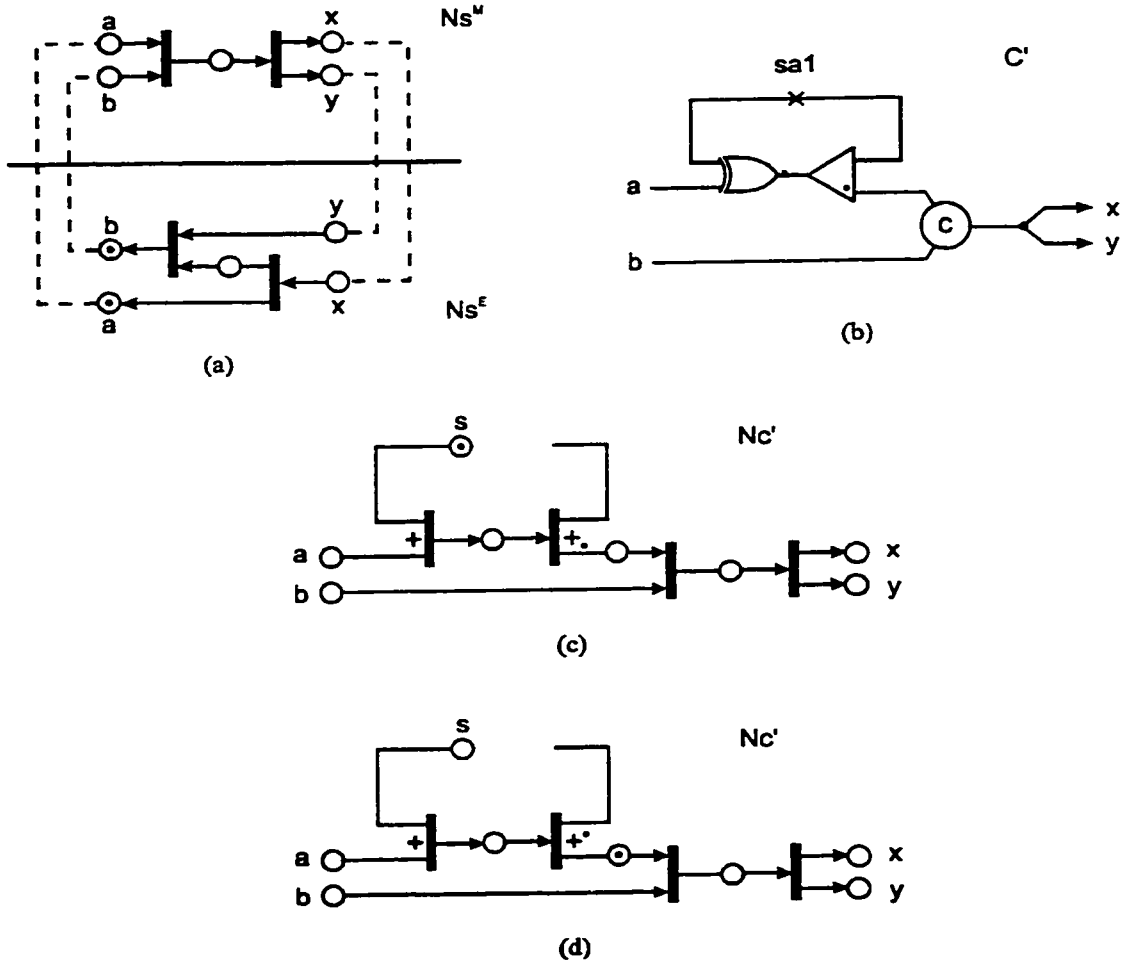


Fig. 2.4: Example (a) Specification *Ns*, (b) circuit implementation *C'* with sa1 fault, (c) net *Nc'* of *C'* at state immediately after global reset, (d) net *Nc'* at stable state after global reset.

with a sa1 fault, and the faulty net *Nc'* of *C'* (single apostrophes are used to denote faulty circuit/net, while no apostrophes indicate fault-free circuit/net). At initialization when the global reset is enabled, a spurious token appears at place *s* because of the sa1 fault (Fig. 2.4(c)). When the global reset is disabled, the spurious token traverses through the XOR and toggle, switches the toggle state, and stops at the terminal C-element. The resulting net in Fig. 2.4(d) shows the stable state after global reset. Note that using the reset strategy described, normal asynchronous operation will not require the waiting for a stable state before sending inputs. Waiting for a stable state in theory will not be DI since internal tokens can take a finite unbounded time to traverse through the circuit, which means the environment cannot know exactly when the circuit will be stable.

2.5 Pomset Behavior Model

In concurrency models, a distinction is made between a generator (finite) and the behavior (possibly infinite) that it generates. Behaviors are sometimes called runs, traces, or processes. Petri net generators can use occurrence nets [9,55,66,83], pomsets [10,20,27,60-64], or trace structures [22,23,67,81] to describe their behavior. Other generators may be STG's [18,56], behavior machines [46,47], regular expressions, or recursive programs. This thesis will use pomsets to describe the behavior of fault-free Petri nets because of the explicit modeling of concurrency allowing for more efficient algorithms and the clearly defined operations available for the pomset model.

Definition:

A ***pomset*** (partially ordered multiset) [10,20,27,60-64] is defined as the isomorphism class of a labelled partial order. A labelled partial order is a 4-tuple (V, Σ, Γ, μ) where V is the set of ***events*** (vertices), Σ is a finite set of ***actions***, $\Gamma \subset V \times V$ is a partial order (set of arcs) expressing necessary temporal precedences among V, and $\mu: V \rightarrow \Sigma$ is a labeling function mapping events to actions.

Let u, v, and w be events and Γ a partial order. Γ is *transitively closed*, denoted Γ^+, if $(u,v) \in \Gamma \wedge (v,w) \in \Gamma \rightarrow (u,w) \in \Gamma$. Γ is *transitively reduced*, denoted Γ^-, if $(u,v) \in \Gamma \wedge (v,w) \in \Gamma \rightarrow (u,w) \notin \Gamma$. Event v is a *successor* of u if $(u,v) \in \Gamma^+$ and v is an *immediate successor* of u if $(u,v) \in \Gamma^-$. Event v is a *predecessor* of (*precedes*) u if $(v,u) \in \Gamma^+$ and v is an *immediate predecessor* of (*immediately precedes*) u if $(v,u) \in \Gamma^-$. A pomset p is a *prefix* of pomset q, denoted $p \leq q$, if p contains a subset of events from q such that for any event u in p, every predecessor of u is also in p. Pomset p is a *proper prefix* of q, denoted $p < q$, if $p \leq q$ and $p \neq q$.

The *projection* of pomset $P = [V, \Sigma, \Gamma, \mu]$ onto a set of actions A, denoted $P\lceil_A$, gives pomset $P' = [V', \Sigma', \Gamma', \mu']$ where $V' = \{v \mid v \in V \wedge \mu(v) \in A\}$, $\Sigma' \in A$, $\Gamma' = (\Gamma^+ \cap (V' \times V'))^-$, and μ' is μ restricted to V' and Σ'.

A *cut* of a pomset P divides the events of P into two disjoint subsets U and V. A cut is *inconsistent* if $\exists\, u_1,u_2 \in U,\ v_1,v_2 \in V$: ($v_1$ is a successor of $u_1 \wedge v_1$ is a predecessor of u_2) $\vee$ (u_1 is a successor of $v_1 \wedge u_1$ is a predecessor of v_2). Otherwise, the cut is *consistent*. □

The partial order Γ of a pomset can be defined as either transitively closed or transitively reduced, but its graphical representation is usually transitively reduced. A multiset is sometimes called a bag in set theory where there can be more than one instance of an element in a set, eg. $\{a,a,b\}$. When modeling circuits, an action is usually interpreted as a labeled wire node (I/O port of a component), and an event is interpreted as the occurrence of a signal transition on that node. Fig. 2.5 shows a pomset P where the occurrence number of actions are labeled with superscripts (this convention will be used in the remainder of this thesis). q is a consistent cut. r is an inconsistent cut because it partitions the events of P into events U on the "left" side of r and events V on the "right" side of r such that event y^2 is both a successor and predecessor of events in U, and b^3 is both a successor and predecessor of events in V. Or equivalently, there exists an arc from the "right" side of the cut to the "left". Every prefix must correspond to a consistent cut of the given pomset.

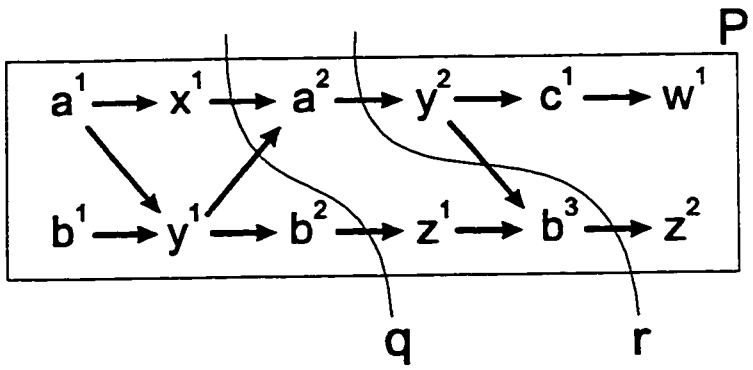
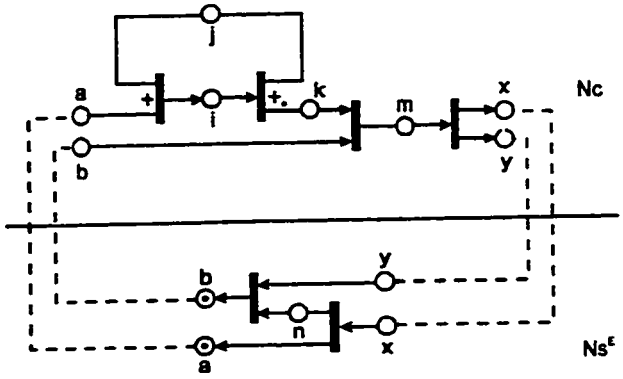
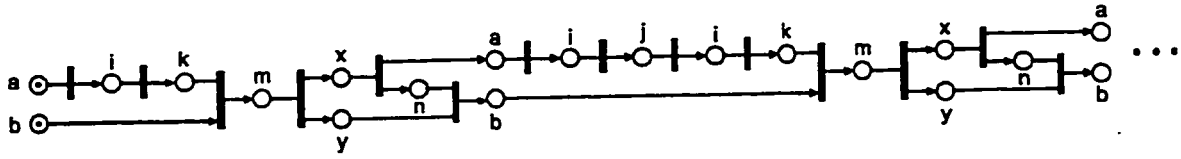


Fig. 2.5: Consistent and inconsistent cuts.

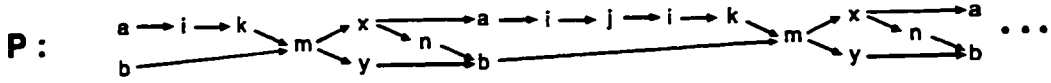
The pomset behavior of a Petri net can be easily obtained by unfolding the Petri net into an occurrence net [9,55,66,83] which does not contain choice places (places containing more than one outgoing arc). This is called a configuration in [55], while some definitions of occurrence net such as [9,66] already exclude choice places. Fig. 2.6 illustrates the net unfolding of the fault-free circuit in Fig. 2.4 composed with its specified environment. The occurrence net directly corresponds to a pomset where pomset events correspond to labeled net places and pomset arcs correspond to net transitions. The existence of choice places in the net means that the net will unfold into a set of pomsets. A choice place in the circuit net is called *output choice* and implies that the circuit contains an arbiter or some possible metastable behavior. A choice place in the environment net is called *input choice* and allows the environment to choose from a possible set of inputs to send into the circuit. A deterministic control circuit contains no input choice nor output choice. Fig. 2.6(d) shows the projection of the pomset onto the I/O ports of the circuit and is the interface behavior that is observable by the environment.



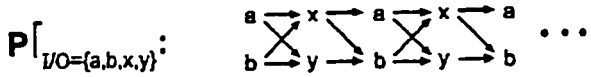
(a)



(b)



(c)



(d)

Fig. 2.6: Unfolding of a Petri net into an occurrence net/pomset (a) Net $Nc \bullet Ns^E$, (b) unfolded occurrence net, (c) equivalent pomset, (d) projection onto I/O ports.

2.6 Test Schedule

2.6.1 Types of Test Mode

There are different types of tests which affect both the implementation of the test and the sufficiency condition of the test:

1) asynchronous
2) semi-synchronous
3) synchronous
   (a) maximally concurrent
   (b) partially concurrent
   (c) single input fundamental mode

An asynchronous test is a test which does not wait for a stable state; after sending a set of inputs to the circuit, the tester waits for only the required outputs to appear and can immediately send in the next allowable set of inputs. A synchronous test is a test which waits for a stable state after each set of inputs. An iteration in a synchronous test is a single step of sending inputs, waiting for stable state, and checking output. A synchronous test is maximally concurrent if the maximum number of allowed inputs are sent during each iteration. This is equivalent to as soon as possible scheduling. The time interval of each iteration does not need to be the same, as long as the time interval is long enough to ensure that the circuit is stable. Single input fundamental mode minimizes the possibility of race/hazard. Partially concurrent synchronous test falls between (a) and (c): a subset of the allowed inputs at each iteration is chosen, allowing scheduling of the input test. In a semi-synchronous test, the environment tester synchronizes sets of inputs from an asynchronous test to occur at the same time so that for example, selected race conditions can be eliminated. It is different from purely asynchronous test since it does not immediately send an input as soon as an output is received. Semi-synchronous is different from synchronous partially concurrent because it

does not have to wait for the entire circuit to settle to a stable state as required in synchronous test. This type of test can be used for example when we want a synchronous test but do not require feedback signals to stabilize, which would then result in a reduction in the iteration interval time. Generally, asynchronous test allows maximum concurrency and potentially fastest test time. However, it also allows the most hazard/race conditions under saf and is the least robust test, while single input fundamental mode is the most robust in that it potentially detects the most saf's since the least number of hazards/races occur.

A self-diagnostic circuit [4] implies an asynchronous test realized by the diagnostic subcircuit since the circuit is run under normal asynchronous operating conditions and no input delay constraints are assumed. [4] assumes asynchronous test mode while [5,31,52] assume synchronous test mode.

Existing test equipment can be used to implement synchronous test. Since each iteration of the test can have variable time, depending on how long the circuit takes to stabilize, the iteration interval can consist of varying numbers of clock cycle times (sampling rate) of the test equipment. It is only required that the test equipment be fast enough so that an iteration interval can be approximated by an integer number of these clock cycles. The implementation of an asynchronous tester would be more difficult because it would require the dynamic change of input vectors, depending on when an output occurs. Current synchronous testers are usually static in that the entire test schedule (set of test vectors) is fixed in advance before testing starts. This is because checking the output value and then changing the input vector value requires a comparator and the checking of a programmable store to determine what the new input vector will be, which would introduce a large delay in the tester and reduce the test speed. A true asynchronous tester might take the form of a dedicated asynchronous DI/SI circuit, or possibly a programmable asynchronous circuit to enable more than one circuit to be tested.

Determining the exact time of each iteration in a synchronous test is not difficult under a fault-free circuit; it can be determined by checking the worst case signal paths during each iteration. However, the existence of a fault can drastically change the circuit such that the worst case path becomes longer than in the fault-free circuit. To determine the minimum iteration time (the least upper bound delay), one must check the worst case signal paths under every possible saf location. Such a check is quite expensive because the possibility of races/hazards can potentially make the check require exponential time.

This check can be avoided if a test which has been proved to be sufficient to detect all saf's under asynchronous conditions is used, that is, we determine the iteration times under fault-free conditions (requiring polynomial time which will be shown) and run the synchronous test; the circuit does not need to stabilize since the test is sufficient under asynchronous conditions. Such a test would be semi-synchronous. Since an asynchronous test is a superset of semi-synchronous and synchronous, ie. when certain delay conditions are enforced by the environment, an asynchronous test will cover a synchronous one.

2.6.2 Timed Petri Net Test Schedule

Strictly speaking, every component and wire can have an unbounded delay in a fault-free DI system, so we must also wait for an unbounded amount of time for an output to appear during normal operation. In order to be certain that there are no faults in the system during a test, we must wait for that finite unbounded time as well, which can be arbitrarily and unrealistically large. However, real world circuit delays are bounded and can be predicted, which means that in testing, the environment does not have to be strictly DI. We will concentrate on synchronous test schedules since existing test equipment are synchronous.

A real-time Petri net model will be used to determine the exact iteration times.

The model associates a timing window [*min*, *max*] with each transition (similar to [61,63]). If a net does not contain any shared places (places which have more than one output transition), then a transition, once it is enabled, will not fire until the delay *min* has elapsed and must be fired before the delay *max* has elapsed. If the net contains shared places, then given transitions t_1 and t_2 with timing windows [min_1, max_1] and [min_2, max_2] which are enabled by a shared place, t_1 (t_2) will not fire until min_1 (min_2) has elapsed and either t_1 or t_2 must fire before max_1 and max_2 respectively has elapsed. For untimed DI specifications, each transition is interpreted as having a timing window of [0, ∞].

The timing windows for each component is determined by the physical implementation. The windows for the net *Nc* can be derived given (i) the windows for each component, and (ii) the windows for each interconnecting wire (dependent on the estimated length of the wire). Alternatively, timing information for *Nc* can be extracted directly through simulation of the final physical layout of the circuit. Since positive and negative signal transitions for the same node may have differing delays because of the implementation of the components, the Petri net model used can be easily converted to each place having either a positive or negative action label.

There are different types of protocol violations:

Definition:

A module has *input safety violation* if the environment sends a token to an input port which is not allowed by the specification. A module has *output safety violation* if it generates a token at an output port which is not allowed by its specification (a spurious token). A module has *output progress violation* if it does not generate a token at an output port as required by the specification. □

In terms of the Petri net specification *Ns*, input safety violation can occur if the environment net Ns^E is replaced, and output safety and output progress

violations can occur if the module net Ns^M is replaced. The tester *Nt* is a special case environment and is assumed to be fault-free, sending only allowed inputs, so input safety violation does not need to be checked. The faulty circuit *Nc'* replaces Ns^M, and may not follow the module protocol correctly, so output safety violation and/or output progress violation must be checked during a test. The requirements for a correct test will be defined in terms of unfolded behaviors in the next chapter.

The test schedule *Nt* has the general form of Fig. 2.7. It contains a series of iteration transitions, I_0, I_1, ..., I_n. When an iteration I_i ($0 \le i \le n$) fires, it sends a set of input tokens to the circuit *Nc'* under test. If *Nc'* is not faulty on this iteration, it sends the required outputs back to *Nt*, and the next iteration I_{i+1} is allowed to fire. If on the other hand *Nc'* is faulty, it may generate one or more spurious tokens and cause transition OSV_i (output safety violation) to fire and report a fault. If *Nc'* does not generate all the required output tokens within a certain time limit (timeout), transition OPV_i (output progress violation) will fire.

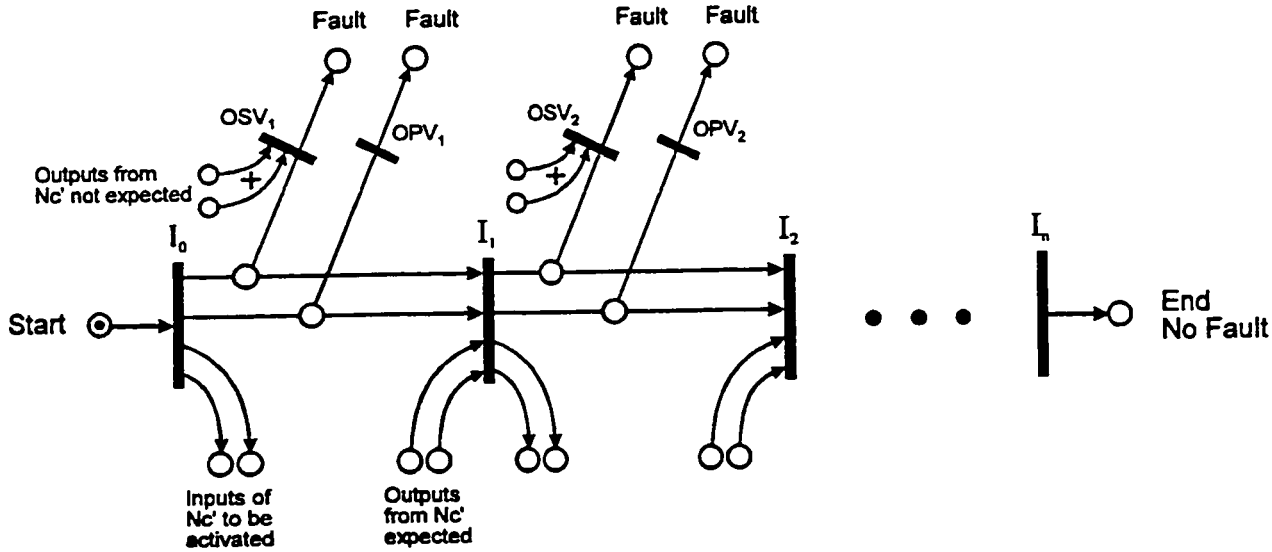


Fig. 2.7: General form of test schedule *Nt*.

The times of the transitions are:

$$I_i = [t_i, t_i], \quad t_0 = 0$$

$OSV_i = [0, t_i - \varepsilon]$

$OPV_i = [t_i + \varepsilon, t_i + \varepsilon]$

Transition I_i fires exactly after time t_i elapses if all its input tokens are available. For synchronous tests, t_i is calculated beforehand and fixed; for asynchronous tests, t_i may be variable, depending on when all the required outputs from *Nc'* has been received. OSV_i has a window ***min*** time of zero and is allowed to be fired immediately upon reception of a spurious output from *Nc'*. It is enabled up till a finitely small time ε before the next iteration transition fires. If I_i does not fire, OPV_i will fire after a finitely small time has elapsed.

Fig. 2.8 shows an example test schedule for a circuit. Starting at the initial marking of *Nt*, I_0 fires and sends a token into the a input of *Nc*. Fault-free operation calls for no outputs to be generated by *Nc*. If an output *x* or *y* is generated during

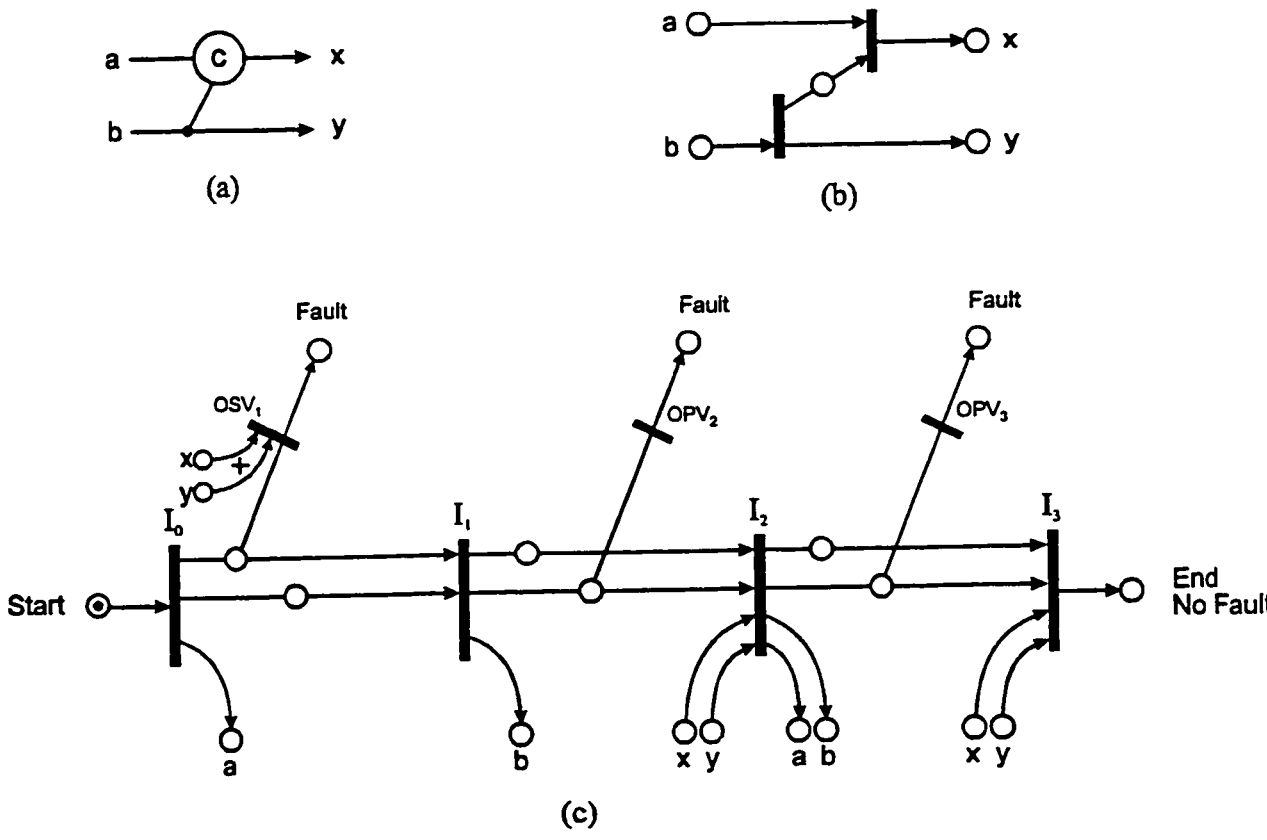


Fig. 2.8: Example test schedule (a) Circuit *C*, (b) Petri net *Nc* of *C*, (c) possible test schedule *Nt* for *Nc*.

window time $[0, t_1 - \varepsilon]$, then output safety violation occurs and the net terminates, reporting a fault. Otherwise, I_1 fires and *Nt* waits for *x* and *y* to be concurrently produced by *Nc*. If both aren't produced as expected after time $t_2 + \varepsilon$, output progress violation occurs. This continues until **End_No_Fault** is reported. As shown, it is possible to either serialize concurrent inputs during the test (*a* followed by *b*) or send all concurrent inputs maximally (*a* and *b* concurrently).

To calculate the time interval t_i of each iteration, we can unfold the composed net *Nt•Nc* into a pomset. Fig. 2.9 shows an unfolding of a timed Petri net, where events of the pomset correspond to labeled places and the arcs with [*min*,*max*] delays correspond to the delays of the timed net transitions. Example transition

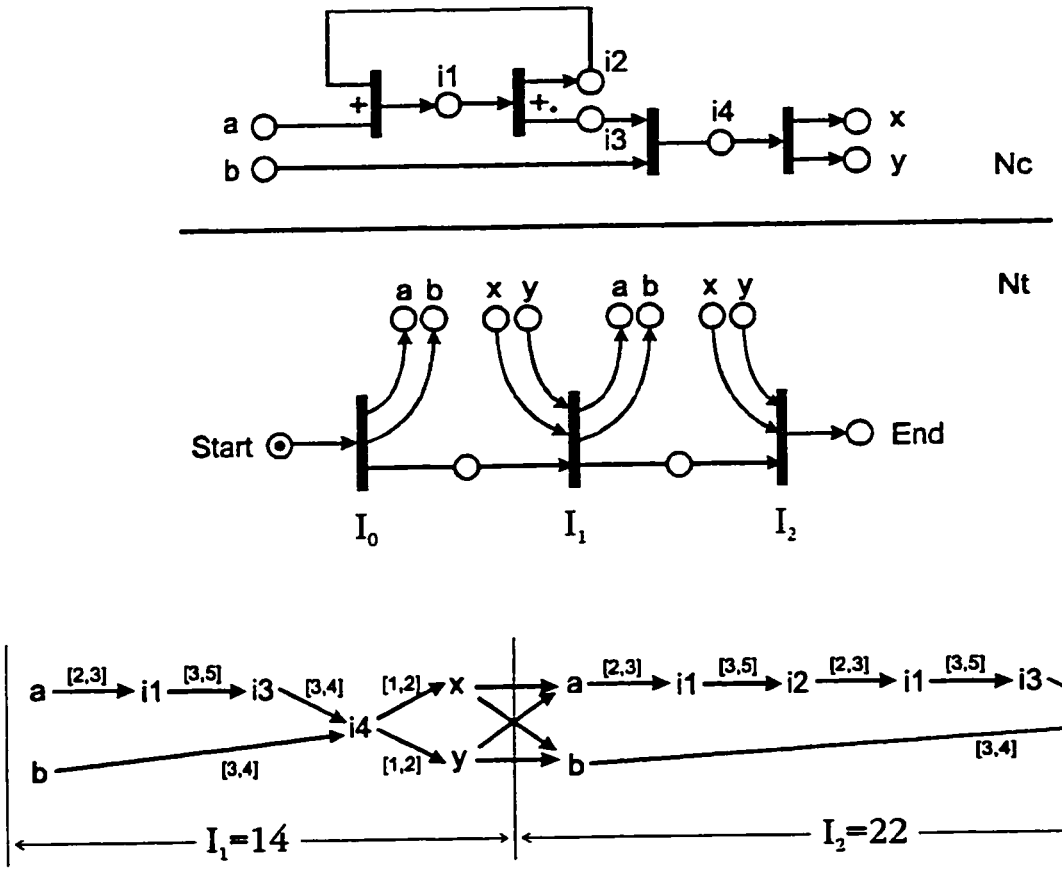


Fig. 2.9: Unfolded timed pomset of net *Nt•Nc*.

time delays of components used are:

XOR = [2, 3]

toggle = [3, 5]

C-element = [3, 4]

fork = [1, 2].

The time interval is equal to the sum of the *max* delays of the worst case path within an iteration. In this case, I_0=[0, 0], I_1=[14, 14], and I_2=[22, 22]. There is potentially an exponential number of paths to check to find the worst case delay path, but the path length is restricted to a single iteration which is usually short.

Chapter 3

Correctness Condition for a Test and the At-least-2 Test

This chapter shows exactly what is required for a correct test. Simply identifying the existence of a saf will not always be sufficient. The correctness condition is dependent on the specified use of the circuit. It is proved that fault-tolerant DI circuits do not exist under the most common use. The at-least-2 test, the test usually implied by other DI and SI asynchronous test groups, is described. Through counterexamples, we show that it is not always sufficient for detecting every single saf. An optimal algorithm is given for deriving an at-least-2 test for circuits with non-deterministic behavior.

3.1 Equivalence

The notion of equivalence is needed to determine that a particular test can always identify whether a circuit containing a possible saf will operate correctly given a particular usage specification. Two generators are said to be equivalent if their behaviors are equivalent; there can be different definitions of generator

equivalence, depending on what semantic behavior is used for the generators and whether the comparison of events in the behaviors are restricted to a certain set, such as observable I/O port events. Behavior can be represented using branching time or linear time semantics and partial order or interleaved semantics. For a survey of different equivalences, see [28]. One possibility is pomset trace equivalence (linear time, partial order semantics) where two Petri nets are equivalent if and only if their set of pomsets generated (unfolded pomsets projected onto I/O ports) are isomorphic. But net unfoldings are currently defined for only 1-safe nets. Under the possibility of token collisions in the Petri net fault model we use, the unfolding is not clear; non-1-safe nets can produce a phenomenon called autoconcurrency (concurrent events with the same action/port label) and there can be a confusion between true causality and time causality when there is a race condition. Instead, we use the trace model [67,81] (linear time, interleaving semantics) which represents behavior as linear sequences of events and is able to specify the behavior caused by token collision/cancellation.

The trace model used is similar to the pomset model except that concurrent events are treated as nondeterminism and sequential composition, eg. $a \| b = ab + ba$. A single pomset with concurrent events can directly be converted into a set of traces with concurrent events interleaved. A trace does not distinguish between necessary causality and temporal precedence. Token cancellation in a trace is then represented as pairs of events being deleted (a shortening of the trace). A formal definition of the trace model will be given in Chapter 4. We will retain the use of pomsets to describe fault-free net behavior because it clearly specifies concurrency, is higher-level, and results in more efficient algorithms, while the trace model will be used only when faulty behavior is involved.

Two Petri nets N_1 and N_2 are said to be ***trace equivalent***, denoted $N_1 \approx_{tr} N_2$, if and only if the unfolded trace sets of N_1 and N_2 projected onto I/O ports (interface

ports) are identical.

3.2 Correctness Condition of a Test

Given the specification *Ns*, its circuit implementation *C*, and a possibly faulty circuit *C'* to be tested containing a possible saf, the requirement is to obtain a test *Nt* which determines if there exists a saf in *C'* which can affect its behavior *according to the use* as specified by Ns^E. A correct test *Nt* for the most general case must satisfy the following condition:

$$\forall i\ (0 \le i \le 2m)\ [\ (Ns^E \bullet Nc'_i \approx_{tr} Ns^E \bullet Nc) \leftrightarrow (Nt \bullet Nc'_i \approx_{tr} Nt \bullet Nc)] \qquad (3.1)$$

under all possible delay conditions, where *m* is the total number of nodes in *C*, Nc'_i is a possible instance of circuit net *Nc'* in which $i = 1$ to *m* represents a sa0 at each of the *m* nodes, $i = m+1$ to $2m$ represents a sa1 at each of the same corresponding *m* nodes, and $i = 0$ represents no saf. Under the composed systems, Ns^E possibly generates infinite behavior while *Nt* necessarily generates finite behavior. If the behavior resulting from the environmental use of a fault-free net *Nc* is different from the use of a net *Nc'* with a saf, then the test *Nt* must also reveal this difference. *Nt* does not necessarily have to follow the usage specification Ns^E, ie. the trace set of $Nt \bullet Nc$ does not have to be contained in the trace set of $Ns^E \bullet Nc$. This can occur when the test follows a different protocol from the given use; the test may even contain token collision/cancellation when run on a fault-free circuit.

The usage specification Ns^E is required in the condition because it is possible to have a saf in *Nc'* with the saf not affecting the behavior according to the usage specification; we call this ***usage fault tolerance*** and is defined as:

$$\exists i\ (1 \le i \le 2m)\ [\ (Ns^E \bullet Nc'_i \approx_{tr} Ns^E \bullet Ns^M)] \qquad (3.2)$$

This is different from ***general fault tolerance*** in which a saf cannot be detected by *any* test *Nt*, whether *Nt* follows the specified usage Ns^E or not, under any delay

condition:

$$\exists i\ (1 \le i \le 2m)\ [\ (Nt \bullet Nc'_i \approx_{tr} Nt \bullet Nc)] \qquad (3.3)$$

A node which is ***redundant*** is a node which exhibits general fault tolerance, and a node which is ***usage redundant*** exhibits usage fault tolerance. Synchronous design usually refers to general fault tolerance. Usage fault tolerance is relevant in protocol based circuits. A saf in the circuit does not necessarily mean that the circuit is incorrect; a saf on a usage redundant node does not affect the behavior when only the specified use Ns^E is used. That same saf may still be detectable if the test does not follow the usage protocol, which means that the saf is not on a generally redundant node. Usage fault tolerance is a superset of general fault tolerance in that it allows more saf's to occur without affecting the protocol.

Definition:

An ***inactive node*** is a node which cannot be exercised (by any environment), a ***single-active node*** is a node which can be exercised only once, and a ***multiple-active node*** is a node which can be exercised two or more times. A ***usage inactive node*** is a node which is not exercised by the specified usage, a ***usage single-active node*** is a node which is exercised only once by the specified usage, and a ***usage multiple-active node*** is a node which is exercised two or more times by the specified usage. □

Fault tolerance can occur if a saf exists on an inactive node (eg. on a grounded programmable input), a usage inactive node, or a usage single-active node. Fig. 3.1 illustrates these cases. The grounded input to the XOR is an inactive node; the circuit is both usage fault tolerant and generally fault tolerant to the sa0 on this node because it can never be detected. According to the specification, node ***n*** is not exercised so it is a usage inactive node; the circuit is usage fault tolerant to a sa0 on this node because it will not affect the use of the circuit. The circuit is not generally fault tolerant to this sa0 because there exists a test not following the

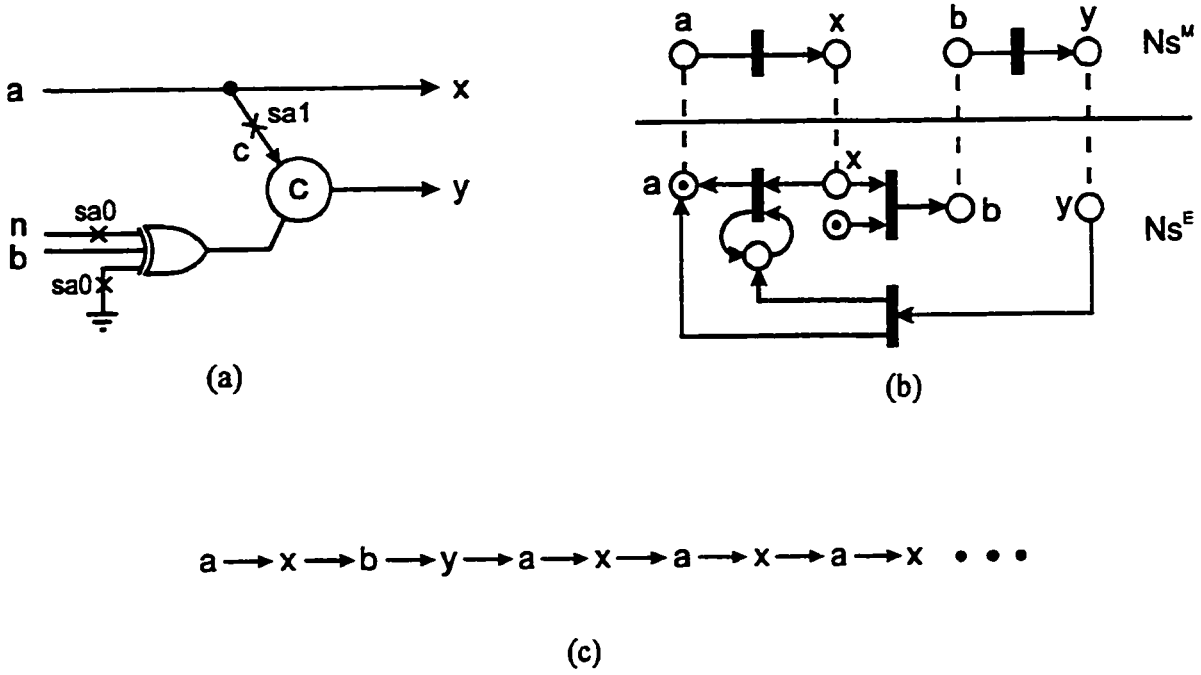


Fig. 3.1: Existence of fault tolerance for class of circuits with inactive and single-active nodes, (a) circuit *C'*, (b) specification *Ns*, (c) pomset behavior for both faulty and fault-free circuit nets.

protocol which can detect it. Node *c* is a usage single-active node since it is exercised only once by the specification (we do not count pairs of tokens canceling as an exercise of the node). A sa1 on node *c* does not affect expected external behavior so the circuit is usage fault tolerant to it, but not generally fault tolerant to it. (This circuit is one of the rare cases where a DI circuit does not need to be 1-safe. It is possible to have token cancellation and transmission interference [81] at node *c* under normal operation of the fault-free circuit, with the circuit still being delay insensitive. The circuit still functions according to the specification regardless of delays. This is because the node is used only once and any subsequent token cancellation effects at that node do not propagate further. DI circuit nets with only multiple-active nodes will always be 1-safe.)

There can also exist circuits which are impossible to test if inactive nodes are present. Fig. 3.2 shows such a circuit where a sa1 causes an initial token at the

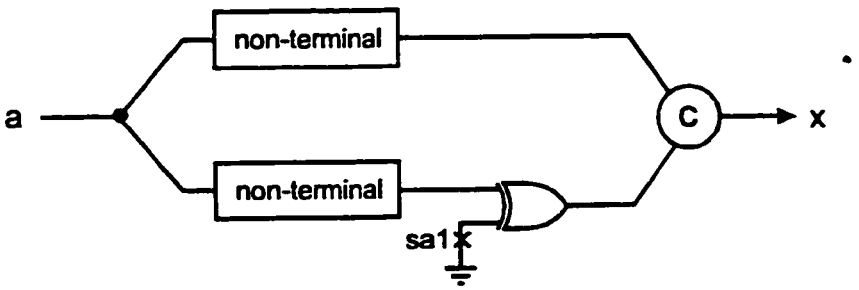


Fig. 3.2: Impossible to test circuit where an inactive node exists.

input of the C-element. A token sent into input *a* causes a race of two tokens through the non-terminals to the C-element. The outcome of the race determines whether an output *x* is produced; if the token on the upper branch wins the race, an output is produced, but if the token on the lower branch wins the race, it will be cancelled by the sa1 token and no output is produced. Suppose the internal token path delays are not directly controllable. Such a circuit passing one test does not guarantee fault-freeness since actual operation may cause different resolutions of the token race due to changes in temperature, different rise and fall times of I/O transitions, etc.

An impossible to test circuit is not fault tolerant because some delay conditions can lead to faulty operation. Such circuits can only be tested using design for test: altering the original circuit by inserting control/observation points such that the saf can be revealed.

Fault tolerant circuits can be extremely difficult to test. There are two possibilities: (i) fault tolerance to improve yield due to fabrication defects (static fault tolerance), and (ii) tolerance to faults during normal operation (dynamic fault tolerance). Dynamic fault tolerance usually requires the test to detect *every* saf, even on the redundant nodes, so that dynamic faults are allowed to occur on redundant nodes without affecting normal operation. Design for test is necessary

for this case because by definition, faults on redundant nodes are not detectable. Static fault tolerance does not require every saf to be detected; it only requires that the circuit is functionally correct and allows saf's to occur on redundant nodes. A functional test can be used to test such circuits, but this is very expensive because of the exponential explosion of possibilities due to choice of inputs and number of states potentially exponential to the size of the circuit. Separating redundant and non-redundant nodes first and then testing the non-redundant nodes for saf's can be just as difficult because identification of redundant nodes is likely to be NP-complete for general asynchronous circuits. Identification of redundant nodes for general combinational circuits has been proved to be NP-complete by Fujiwara [26].

This thesis considers testing for functional correctness as defined by condition 3.1, and so implies the consideration of static fault tolerance and the requirement of a functional test (simply testing for *any* saf means that some circuits would unnecessarily be discarded when the saf occurs on redundant or usage redundant nodes). However, we can restrict the class of circuit tested to the most common type of circuit containing only usage multiple-active nodes (multiple-active nodes not used by the protocol will not be important). This is a common assumption made by other asynchronous test groups.

Theorem 3.1: (non-existence of DI fault tolerance)

There does not exist fault tolerance for DI circuits with only usage multiple-active nodes.

Proof:

Since usage fault tolerance is a superset of general fault tolerance, we only need to prove impossibility of usage fault tolerance.

Suppose there exists usage fault tolerance to a saf at location *s* of the circuit.

→ Every possible execution (trace) under the specified use does not reveal the saf as output safety or output progress violation.

→ We prove that there does exist a trace which reveals violation, hence contradicting the above condition.

Case inhibitory fault:

(The only difference between the faulty and fault-free nets is the disconnect preceding place *s*.)

Fire identical transitions in both the faulty and fault-free nets until the first token is produced at *s* in the fault-free net and no token is produced at *s* in the faulty net. At this point the trace of both nets are identical except for the missing *s* in the faulty trace. The faulty net/trace delays indefinitely due to the missing token at *s* and must appear as output progress violation.

Case excitory fault:

(The difference between the faulty and fault-free nets is the disconnect preceding place *s* and a spurious token deposited at *s*.)

Phase 1: Chose execution in which spurious token is held at place *s* and execute both nets in the same manner as for the inhibitory fault above. At this point, the net marking is identical for both nets, with the spurious token masquerading as the normal token produced at *s*.

Phase 2: Continue execution until the fault-free net produces a token at *s* and the faulty net does not because of the disconnect. The faulty net/ trace delays indefinitely and must appear as output progress violation. □

This is an important theorem which greatly simplifies DI testing. The proof is not valid for SI circuits because the above executions are not guaranteed to produce output progress violation due to the fact that an SI system does not always have to wait indefinitely for a token to be produced. This can occur when a saf appears at one branch of a fork. Since the fork can be isochronic, the system needs only to wait for an acknowledge from one branch of the fork and not all branches as in DI circuits. If the saf appears on such a branch, the system does not wait

indefinitely as in DI circuits, but can continue operation, producing faulty behavior such that safety violation will mask the progress violation. It appears that SI fault tolerance can exist, such as the sa0 in the two D-element circuit of [52] since the interface behavior is the same for the faulty and fault-free circuit under all delay conditions; even though the output request can be produced temporally prematurely due to the sa0 compared to the fault-free circuit, it does not affect the interface protocol.

Non-existence of DI fault tolerance implies that the detection of *any* saf in the circuit will indicate a faulty circuit. The usage specification Ns^E in condition 3.1 is therefore not required for circuits containing only usage multiple-active nodes, and the correctness condition for a test is simplified to:

$$\forall i\ (1 \le i \le 2m)\ [\ \neg(Nt \bullet Nc'_i \approx_{tr} Nt \bullet Nc)\ ] \qquad (3.4)$$

It is sufficient to obtain a test Nt independent of Ns^E which *distinguishes* Nc' from Nc for any saf, where distinguishes means the exercise of Nt on Nc' produces output safety or output progress violation with respect to $Nt \bullet Nc$.

This thesis distinguishes saf's which are (i) usage fault tolerant, (ii) generally fault tolerant, and (iii) impossible to detect. Only (iii) requires design for test. Hazewindus [31] considered these three cases together as *undetectable faults* and inserted control/observation points to detect them, which meant that cases (i) and (ii) unnecessarily had control/observation points inserted. This is because redundancy detection can be a difficult problem for SI circuits, even if every node is usage multiple-active.

3.3 At-least-k Test

In traditional testing of combinational circuits, the possible existence of every sa0 and sa1 fault at every node in the circuit is usually checked and a set of test

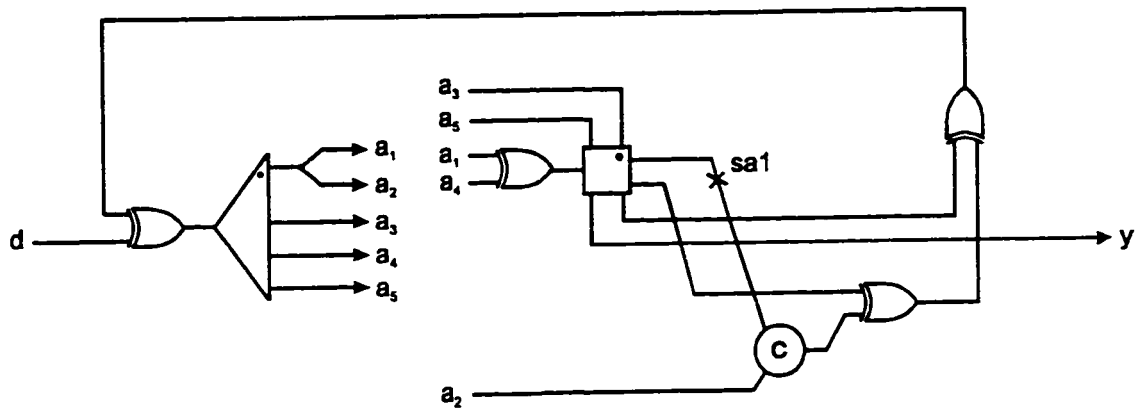
vectors is obtain which covers the faults. In asynchronous circuit testing, this would correspond to the brute force method of enumerating every saf along with its corresponding faulty circuit net Nc_i' and obtaining a separate test Nt_i for every saf. Some optimization can be done for minimizing the overlap of Nt_i where one Nt_i is guaranteed to detect more than one saf, equivalent to a test vector detecting more than one saf. Such a strategy is much more difficult for asynchronous sequential circuits because it is very difficult to find a test which is hazard and race free. Random generation of inputs is very likely to create hazards and the checking of fault coverage under hazardous/race conditions is equivalent to race simulation and potentially requires exponential time. The method of scan design to partition the combinational components and test them separately does not apply here since 2-phase transition signaling control circuits have a low number of combinational components. However, due to the nature of DI circuits, tests which satisfy simple conditions can significantly reduce the complexity of a test.

Definition:

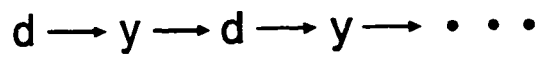
Let *Nt* be a test schedule, *Nc* be the net for a fault-free circuit *C*, and the pomset action set be the set of input, output and internal nodes of *C*. *Nt* satisfies the ***at-least-k*** condition if the pomset unfolding of *Nt•Nc* contains a prefix *p* such that (i) *p* contains at least ***k*** occurrences of each action in *C*, and (ii) the last events of *p* are output events. □

An at-least-k test guarantees that a set of tokens has traversed every node in the circuit ***k*** times (controllability) and that this set has propagated to the environment output (observability). The hypothesis is that at-least-2 is a sufficient condition for detecting a saf in a circuit since intuitively, a saf will prevent a node from being exercised twice and cause output progress violation. Since every transition in a DI circuit must be acknowledged, a saf inside the circuit will very likely prevent an acknowledge from reaching the output. But this is true only for a

certain class of circuits since the at-least-2 test guarantees exercise of the fault-free circuit and not the faulty one. The existence of a saf can change the state of the circuit such that input tokens are redirected around the saf and prevent detection of output progress violation.



(a)

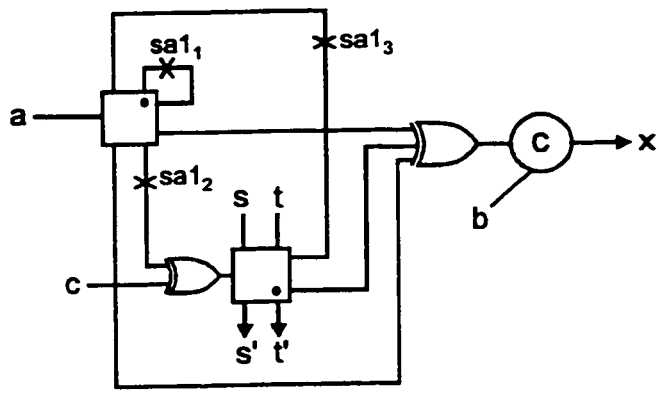


(b)

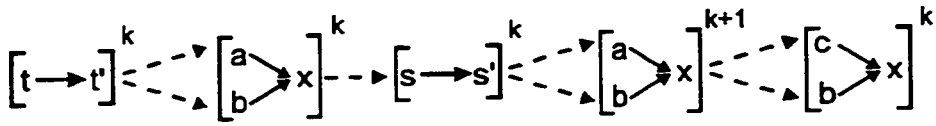
Fig. 3.3: C1: counterexample to sufficiency of at-least-2 test detecting saf for asynchronous test, (a) circuit, (b) fault-free behavior.

Fig. 3.3 shows such a case where an at-least-2 test is insufficient to detect a saf. A test *Nt* which exercises every node twice in the fault-free circuit will be (*d*; *y*; *d*; *y*). The faulty circuit can also pass this test depending on certain delays (since the sa1 causes operation to be no longer DI). The initial input token sent into *d* will pass through the toggle and be forked into tokens a_1 and a_2. Suppose the path of a_1 is much longer than that of a_2 (path of a_1 may contain additional components). Token a_2 enters the C-element input which fires prematurely due to the sa1. Suppose this token traverses through the feedback path, creates a_3 and sets the demultiplexer before a_1 enters the demultiplexer. a_1 is now redirected to the lower output around the sa1. Execution continues and two *y* outputs can be produced.

Under asynchronous test conditions, this circuit is impossible to test without additional control points; an at-least-k test with unbounded k is not sufficient to detect this fault because token a_3 can always win the race and redirect token a_1 around the saf. Under synchronous test conditions, at-least-3 will be sufficient to detect the saf because waiting for stable state allows a token to hit the saf.



(a)



(b)

Fig. 3.4: C2: counterexample to sufficiency of at-least-k test detecting saf for both asynchronous and synchronous test (at-least-k for k>1 not sufficient), (a) circuit, (b) fault-free behavior.

Fig 3.4 shows a second counterexample circuit in which at-least-k for unbounded k is not sufficient to detect a saf, even when synchronous test is used, for any one of the three single saf's: $sa1_1$, $sa1_2$, or $sa1_3$. The k superscript in the bracketed pomset segments denotes k repetitions of that pomset segment. Unlike the circuit of Fig. 3.3, the critical race at the demultiplexer appears at initialization when $sa1_1$ exists in the circuit, ie. the $sa1_1$ falsely sets the demultiplexer at initialization so that future tokens sent into the a input will be redirected around the

saf. The existence of either $sa1_2$ or $sa1_3$ makes the test insufficient by depositing a spurious token at the input of the C-element and prevents the proper setting of a demultiplexer during execution. It is clear that design for testability is required for such circuits.

In general, any race of two tokens into a demultiplexer (input safety violation) will cause non-deterministic behavior. Ebergen uses the RCEL in place of the demultiplexer, and the symmetric confusion appearing in the low-level Petri net also likely causes the same critical race problem. Since the toggle is the only other state component which can control the path of tokens and which cannot have non-deterministic behavior, the hypothesis is that the at-least-2 test is sufficient for demultiplexer and RCEL-free circuits, which will be proved in the next chapter.

3.4 Derivation of At-Least-K Test

The derivation of an at-least-k test for deterministic circuits (circuits in which no choice can be made in the module or environment) is simple. The net $Ns^E \bullet Nc$ can be unfolded into a single pomset. Only the projection of the pomset onto the I/O ports need to be recorded. The pomset can simply be advanced (net transitions fired) until the at-least-k condition is satisfied. The resulting pomset is the test schedule Nt.

For nondeterministic circuits (circuits in which choice can appear in the module or environment), the unfolding can lead to many different pomsets, depending on which choices are made. Pomsets of different lengths may satisfy the at-least-k condition.

Definition:

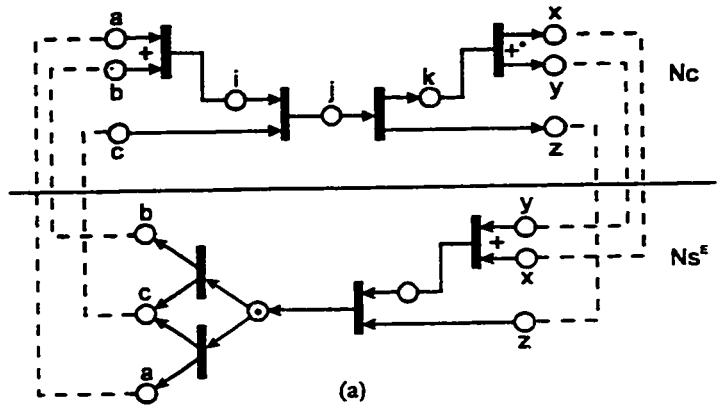
The *postset* of a Petri net place p is the set of output transitions of p, $\{t \mid (p, t) \in F\}$. The *preset* of a Petri net transition t is the set of input places of t, $\{p \mid (p, t) \in F\}$. A *choice place* is a Petri net place whose postset contains more than one

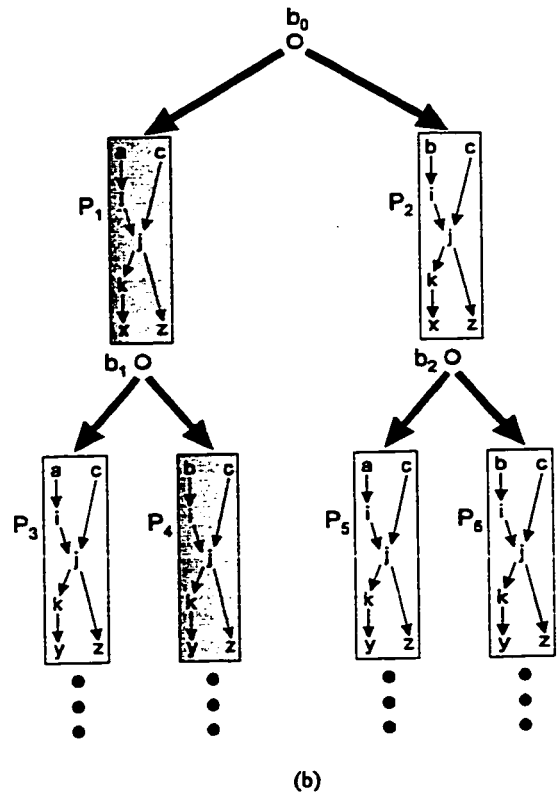
transition. A ***choice transition*** is a transition whose preset contains a choice place. A ***branch state*** is a Petri net marking in which no transition other than a choice transition may fire. A ***pomset tree*** is a 4-tuple $T=<B, P, b_0, \delta>$ where B is the set of branch states, P is a set of finite pomsets with pomset events mapped onto Petri net places, b_0 is the initial branch state, and $\delta: B \times P \rightarrow B$ is a transition function. Graphically, a pomset tree has the form of a tree in which b_0 is the root node, tree nodes are branch states, and arcs are pomsets, with the outdegree of each node being variable. □

For the derivation of an at-least-k test for non-deterministic circuits, a pomset tree can be used (the pomset tree is the same as the pomtree in [64] but defined differently here). A pomset tree explicitly records when choice is made, enabling an efficient search through a system's state space. It is analogous to a Petri net reachability tree, but instead of having state transitions between states (net markings), a pomset tree has pomset transitions between branch states. Its major advantage is the elimination of unmanageable state space explosion due to concurrency.

Given a Petri net, the corresponding unfolded pomset tree is obtained as follows. The initial net marking M_0 corresponds to the initial branch state b_0. The net is executed (unfolded) until a branch state is reached (until no more transitions can fire without making a choice). Record branch state $b \in B$. The number of pomsets branching out from b is equal to the total number of unique sets of choice transitions which can concurrently fire, where the firing of each set of transitions deletes every token sitting on a choice place at branch state b. Each pomset branching from b is similarly unfolded.

Fig. 3.5 shows the composition of a net representation of a circuit and its environmental use, $Nc \bullet Ns^E$, and the corresponding unfolded pomset tree. For the partial pomset tree shown, $B=\{b_0, b_1, b_2\}$, $P=\{P_1, P_2, ..., P_6\}$, $\delta(b_0, P_1)=b_1$, $\delta(b_0, P_2)=b_2$.





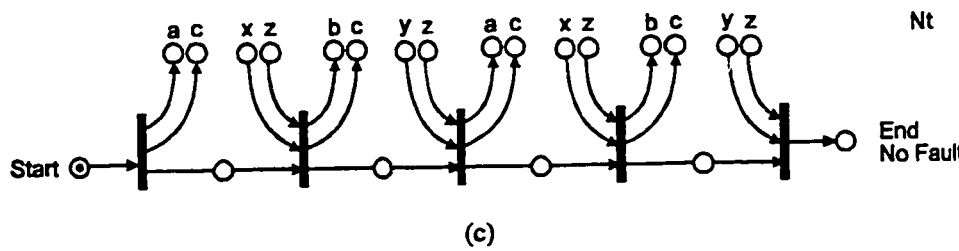


Fig. 3.5: Pomset tree example (a) System $Nc \bullet Ns^E$, (b) corresponding unfolded pomset tree, (c) at-least-2 test derived from pomset tree.

A test is obtained by following a single path (sequence of branch states) through the tree until all external and internal nodes are exercised k times and all internal nodes have caused an external node firing. The sequence of finite pomsets traversed will be the at-least-k test. The shaded pomsets indicate the path taken which leads to the at-least-2 test Nt of Fig. 3.5(c).

The following is an optimal algorithm for deriving the shortest length at-least-k test. Let Q be a set of linear paths through a pomset tree represented by 4-tuples $<A, R, P, D>$ where $A=[1..m]$ is an integer vector of all m actions (nodes) of the fault-free circuit to keep track of the number of times each node $A[i]$ has been exercised, R is a sequence of branch states in a path, P is a pomset obtained from traversal through a single path in the pomset tree, and D is the total delay incurred by P, representing the time requirement of the test. Let $p,q,r \in Q$ and the period in $p.X$ denote tuple element X of p.

Algorithm 3.1: (derive at-least-k test)

extend_path(q)
  if $q.P$ has not fully reached branch state last($q.R$) **then**
    advance $q.P$ up to an output event or a branch state;
    increment $q.A$ according to actions exercised;
    calculate delay d of advance;
    $q.D := q.D + d$;
    return(q);
  else
    foreach outbranch $b \in B$ from last($q.R$) **do**
      r:=copy(q); $r.R$:= append($r.R$, b); $Q := Q \cup r$;
    endfor
    $Q := Q - q$;
    return(*nil*);
  endif

end.

Derive_Test()

$\forall i$: $q.A[i] := 0$; $q.R := b_0$; $q.P :=$ Λ (empty pomset); $q.D := 0$; $Q := q$;

Repeat

select a $q \in Q$ such that $q.D$ is minimal;

$q :=$ extend_path(q);

Until $q \neq nil$ and at-least-k($q.A$);

{check if q is really optimal solution}

Repeat

select $p \in Q$ such that $p.D < q.D$ and ¬at-least-k($p.A$);

extend_path(p);

Until $\forall p \in Q$: [at-least-k($p.A$) or $p.D \geq q.D$]

solution:= $q \in Q$ such that $q.D$ is minimal and at-least-k($q.A$);

end.

where at-least-k(A) is the predicate $\forall i$: $A[i] \geq k$, and last(R) denotes the last branch state of sequence R. □

This is a breadth first search algorithm which extends the path with the minimal delay incrementally until a path q which satisfies the at-least-k criteria is found. To ensure that this is the optimal solution, all the other paths are extended up to the delay of path q, since the final extension of q may be large. The pomsets are advanced incrementally up to output events because exercised internal events must be observed at an output. The advancement does not jump from one branch state to the next because a single pomset segment in the pomset tree can contain several iterations of I/O events without making a choice, eg. in a deterministic control circuit. The advancement of a prefix/pomset up to an output event will be defined clearly in Chapter 5. Other non-optimal depth first search algorithms are possible which improves on run time and space, but for testing circuits in

production runs, an optimal test is more desirable. Since we will be interested in at-least-2 tests, the height of the tree will be small. Chapter 6 will show an alternate method in which a control point insertion algorithm will automatically optimize the test length (such as when zero control points are selected), so an optimal initial at-least-k test will not always be required.

Chapter 4

Structural Theorems

As seen from the previous chapter, the at-least-2 test is not sufficient to detect every single saf in some circuits. In particular, components which contain the confusion structure in their low-level nets can be a source of problem because of their ability to redirect tokens around a saf and make the fault not observable. We first show how the at-least-2 test can be made to have 100% fault coverage through design for testability, the insertion of observation points, and then show when these observation points are not necessary through a set of structural/behavior theorems. We prove that a large class of circuits do not require any observation points. The proof is non-trivial because of the need to consider race behavior.

4.1 Observation Points to Achieve 100% Fault Coverage

The detection of inhibitory faults is simple. It can be shown that an at-least-1 test achieves 100% fault coverage of inhibitory faults in DI circuits without the use of observation points. For all theorems in this chapter, it is assumed that every node in the circuit is multiple-active under a given usage.

Theorem 4.1:

At-least-1 test is sufficient to detect every single (and multiple) inhibitory saf in a DI circuit.

Proof:

The only difference between the faulty and fault-free net is a disconnect preceding the saf place s in the faulty net. The unfolded behavior of the faulty net must be contained in the fault-free behavior because the net marking of both nets are identical (every transition firable in the faulty net is also firable in the fault-free net) up until the firing of the transition preceding s which does not deposit a token in s of the faulty net. This is an infinite delay in the faulty net, resulting in every event caused by s^1 not being produced, including some interface output event which results in output progress violation. □

A net with an inhibitory fault is one case where the faulty net can be unfolded into a pomset because there is no possibility of race and autoconcurrency. Because the circuit is DI, the indefinite delay is included in the fault-free behavior and no other faulty behavior can arise. The indefinite delay of an event results in the faulty pomset behavior being a proper prefix of the fault-free behavior.

To detect the remaining excitory faults, in the extreme case, an observation point can be inserted at the input of every state component such that the existence of a spurious token due to an excitory fault will be detected at initialization. The observation point is a fork at the input of the components and all observation points are combined through a large OR gate (insert inverter if the observation input is normally high) as shown in Fig. 4.1. At circuit initialization, the inputs of all state components are disabled upon enabling of the global reset. The observation output is then checked for existence of spurious tokens, and then the global reset line is disabled. Any spurious token must be detected since they will traverse through non-terminal non-state components and stop at the input of a state

component, or else be observed at the interface output. It is assumed that the observation point is placed physically as close to the input of the component as possible and that no excitory fault exists between the observation point and the component input. For example, an excitory fault at location *b* of Fig. 4.1 would not be detected. Since that wire would be very short in practice, one can consider it as part of the component and assume that there are no saf's inside the component under the present fault model. In theory, this may be considered as using the OSAF model, but is different in that the fork branch is not allowed to have arbitrary length. This assumes that the excitory fault only propagates in the forward direction of normal components; if the OR gate is considered as a normal component, then *b* would be a fork to the inputs of both the C-element and OR gate, and the excitory fault would always be observed.

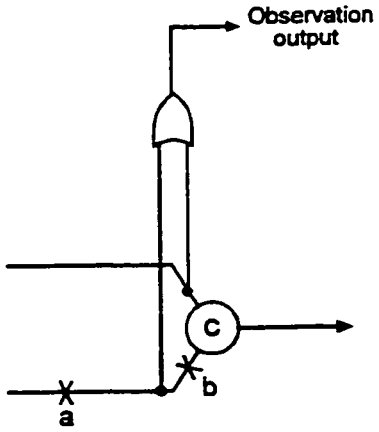


Fig. 4.1: Observation points at input of state components.

If a saf occurs in the OR gate network, the observation output may incorrectly report no spurious tokens. However, under the single saf model, there would then be no saf in the rest of the circuit, and the circuit would operate normally.

If no spurious tokens are observed, then running an at-least-2 test will detect the remaining possibility of inhibitory faults. Observing every input may be expensive, so we would like to relax this requirement. One hypothesis is that

observing only the inputs of terminal components (C-elements, arbiters) is sufficient to detect all excitory faults. It would then be possible to enable and disable the global reset, wait for the circuit to stabilize, and then observe the terminal component inputs. Any spurious token will have traversed through the non-terminals and stop at some terminal component. This is true in most cases, however, two cases exist in which a spurious token would not be observed. (The paper of [49] inadvertently overlooked these cases.) The spurious token can either cancel itself out or traverse in an infinite cycle of non-terminals so that observing only terminal component inputs would not detect the excitory fault. Fig. 4.2 shows the possibility of self-cancellation where the spurious token caused by the excitory fault forks into two tokens and then merges into the XOR. The two tokens existing

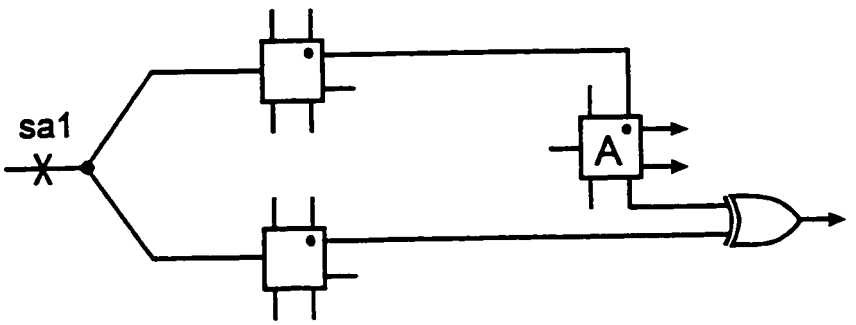


Fig. 4.2: Cancellation of spurious token in a fork-merge.

at the output of the XOR would then cancel each other. This behavior would not normally occur in a DI circuit, but this circuit structure can exist if the normal behavior first sets a demultiplexer on only one branch of the fork before sending a token through the fork so that only one token ever merges at the XOR at a time. The circuit should also contain a non-terminal state component on a branch which potentially can have its internal state switched by the spurious token, as exemplified by demultiplexer *A* in the figure; otherwise the token would cancel itself without changing the low-level net marking and the fault would reduce to an inhibitory fault. To make the spurious token observable, a single observation point

can be inserted in any one of the non-terminals in the branch before cancellation occurs.

Fig. 4.3 shows the possibility of the spurious token traversing in an infinite cycle. Infinite cycles normally do not exist in DI circuits (one exception is the infinite cycle in Ebergen's token-ring interface circuit [23], pp. 35), but such a structure can exist if one of the demultiplexers are set first before a data token is sent in. No infinite cycle exists in the fault-free circuit behavior, but under faulty conditions the infinite cycle may occur.

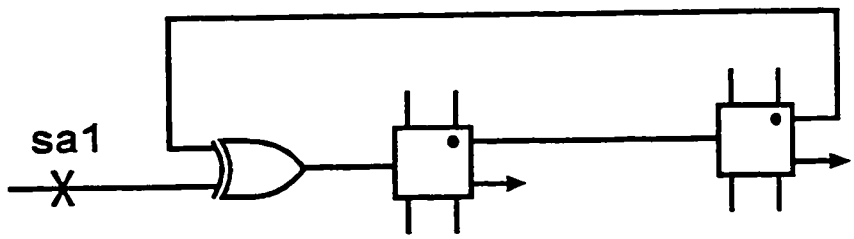


Fig. 4.3: Infinite cycle of spurious token.

These two cases can be detected by performing a structural check of the circuit. The infinite cycle behavior can also be simply detected by checking the current drawn at initialization for technologies such as CMOS where almost no current is consumed when there does not exist a signal transition. The delay paths of the fork-merge can also be checked to verify the impossibility of a token cancellation. If such a structure exists, then a single observation point can be inserted in either the fork-merge or the cycle. The inserted point is sufficient, but may not be necessary because the existence of the structure might not mean there will be a cancellation/infinite cycle at the behavioral level. Behavioral simulation can be used to check the exact condition at initialization, but it may be more expensive than simply inserting one observation point.

The following clarifies what is required if a structural check is used.

Definition:

Let the ***structural graph*** of a high-level Petri net be a directed graph $G = \langle V, E \rangle$ in which graph vertices V correspond to net places and for every net transition with input places $\{a,b,\ldots\}$ and output places $\{x,y,\ldots\}$, there are directed edges $E = \{a,b,\ldots\} \times \{x,y,\ldots\}$. A ***path*** from initial vertex a to terminal vertex b in a directed graph G is a sequence of one or more edges (x_0,x_1), (x_1,x_2), (x_2,x_3), ..., (x_{n-1}, x_n) in G where $x_0=a$ and $x_n=b$. A ***reconvergent branch*** is a set of two or more paths $\{p_1, p_2, \ldots, p_n\}$ where the initial vertex of every path is a, the terminal vertex of every path is b, where $a \neq b$, and the set of edges of each path are disjoint (set of edges of $p_1 \cap p_2 \cap \ldots, \cap p_n$ is empty). A ***directed cycle*** is a path that begins and ends with the same vertex. A ***simple cycle*** is a directed cycle which does not contain the same vertex more than once, except for the initial vertex. A ***complex cycle*** is a directed cycle which contains one or more vertices which occur more than once, except for the initial vertex. A ***control vertex*** is a vertex corresponding to the input place of a component in which a token sent to that input can cause the internal state (marking) of that component to change. □

The graph is structural since it includes all possible paths that a token can take from input place to output place of a transition, regardless of the net marking. An example structural graph is shown in Fig. 4.4. A toggle component will appear

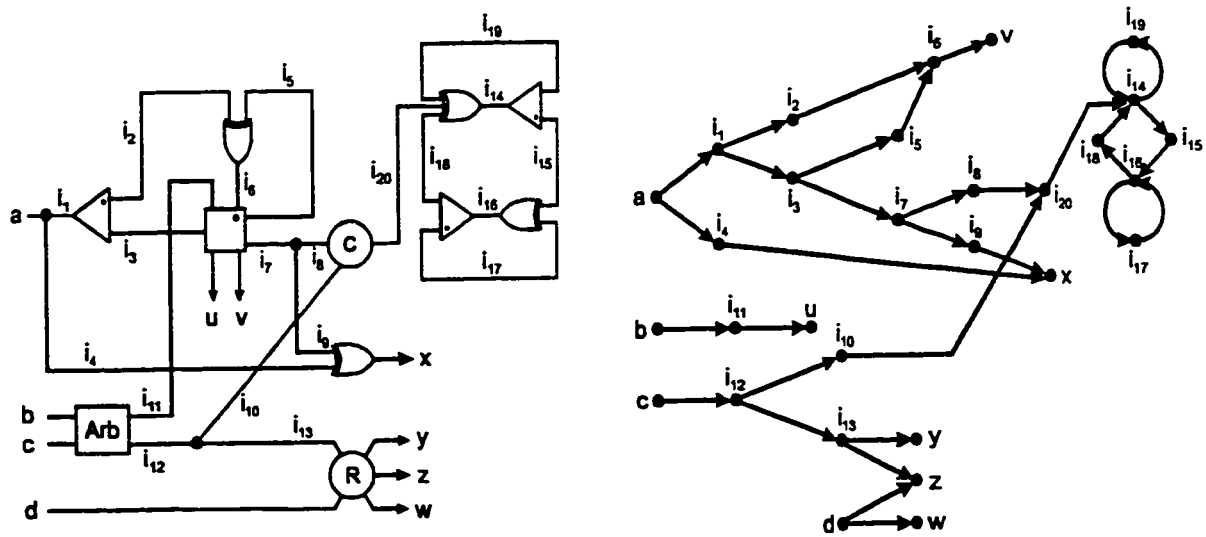


Fig. 4.4: A circuit and its corresponding structural graph.

as a fork to cover the possibility of an input token being sent to either one of its outputs. Even though the circuit has a feedback at i_5, i_5 is not contained in a cycle in the structural graph because tokens on the data and control paths of a demultiplexer do not cross. Reconvergent branches exist from i_1 to i_6 and from a to w. A complex cycle exists from i_{14} to i_{19} and which also appears as three simple cycles. Existence of reconvergent branch or directed cycle implies possibility of token cancellation or infinite cycle in the behavior of the circuit. Additional structural conditions can be used to determine more precisely when these two behaviors may occur.

The following conditions must hold if the ***token cancellation structure*** exists:

(1) existence of reconvergent branch in structural graph
(2) initial vertex of branch must be a fork vertex, terminal vertex of branch must be a XOR vertex.
(3) all paths in reconvergent branch contain non-terminals only
(4) existence of a control vertex within reconvergent branch.

Condition (2) eliminates reconvergent branch i_1 to i_6 from consideration since the initial vertex is a toggle. Condition (3) is required since the existence of any terminal component on any path will stop the spurious token and allow its detection. Condition (4) is included since token cancellation without changing component internal states is equivalent to existence of inhibitory fault. Additional structural conditions can be included to strengthen the requirement and reduce the need for an observation point. For example, if forks exist within the reconvergent branch structure, then the outputs of all forks which do not lead to the terminal vertex XOR will be producing additional spurious tokens, and these spurious tokens can only be eliminated if they can reach another token cancellation structure, or an even number of these spurious tokens reaches another XOR. This would eliminate reconvergent branch a to w because there is a fork at i_7 which

would produce a spurious token at i_8 and be detected by an observation point at the C-element. The size of the initial vertex fork does not necessarily have to have an even number of outputs since it is possible for an odd number of spurious tokens to reach another cancellation structure or there may be another fork within the structure to create an additional token such that the terminal vertex XOR receives an even number of tokens.

The following conditions must hold if the ***infinite cycle structure*** exists:

(1) existence of directed cycle in structural graph
(2) there must exist at least one XOR vertex within the cycle to allow the injection of a spurious token
(3) the cycle contains non-terminals only
(4) if a toggle input vertex exists within the directed cycle, then the directed cycle must form a complex cycle which includes both toggle output vertices.

Condition (2) is required because the spurious token must be injected into the cycle; an excitory fault creating a spurious token from inside the cycle will obviously not result in infinite cycle behavior since the disconnect will terminate the cycle. A control vertex is not required in the cycle. In condition (4), a simple cycle containing a toggle input vertex cannot result in infinite cycle behavior occurring only within that simple cycle; infinite cycle behavior can only occur if it is a complex cycle covering all outputs of the toggle. If other multiple output components are included which can also deterministically generate a different output token for every input token, such as a component which generates a sequence of outputs, then the complex cycle must also include all the output vertices of that component in which output tokens can be produced.

The insertion of an observation point causes a non-terminal component to appear as a terminal component at initialization so that the spurious token can be observed. In the structural graph, the effect is the deletion of an input arc at some

vertex, thereby cutting the directed cycle and disallowing infinite cycle behavior. The algorithm for detecting these two structures is not difficult since it is essentially the detection of reconvergent branches and directed cycles in the structural graph.

We know that spurious token cancellation and infinite cycle behavior must be the only two cases in which observing terminal component inputs is not sufficient to detect the spurious token because of the following: Given the universe of possibilities, the circuit must be in either an unstable or stable state. Infinite cycle behavior must occur if the circuit is in an unstable state. Of the remaining stable states, suppose there are k stable states; tokens can only exist at the inputs of terminal components, otherwise the circuit is not stable. One of these stable states is the one in which there are no spurious tokens remaining. This must be the case were the spurious token has reached the interface output where the environment detects the fault, or else it must be due to spurious token self-cancellation. The remaining $k-1$ stable states must have at least one spurious token at a terminal component input, which will be detected by the observation points.

The area overhead of observing every terminal component input may range from 0% overhead if the circuit consists of only non-terminals, to possibly around 20% when considering the number of transistors required to implement a pair of OR inputs relative to the number of transistors of a typical terminal component (of course, this is highly implementation dependent). On average, the overhead will not be as high as the worst case, and the extreme of a circuit containing only terminal components will not require any observation points at all, as will be shown. The large OR gate can be implemented as one large complex gate at the transistor level. The speed of this gate is not important since it is not used during normal operation and is observed only at initialization. However, there can be a speed overhead to the regular circuit because of the extra input capacitance of the large OR gate.

4.2 Structural Theorems

The following sections will give structural theorems to determine exactly when observation points are not required in an at-least-2/at-least-1 test to obtain 100% fault coverage. Possibly, design for testability is not available, so it would be useful to determine exactly what class of circuit or circuit structures in which no observation points are required. Structural checks are inexpensive and usually require low order polynomial time. Many can be performed similar to finding a spanning tree. One important theorem simply checks for component set inclusion.

It is possible to use behavioral simulation to check whether a saf at a particular location is covered by an at-least-2 test (as is done in [31]), but this can potentially require exponential time because a saf can cause races/hazards. A branching in the behavior is created every time there is a possibility of a race or hazard. [74] gives a summary of the computational complexity of race detection (unique stable state reachability) under different race and delay models, with many of them being intractable. Race simulation is generally very expensive compared to static structural checks. A *critical race* is defined as a race condition which can cause the circuit to settle into more than one possible stable state. A *noncritical race* is a race in which the circuit always settles into the same stable state; for example, two concurrent tokens traversing toward a C-element is a noncritical race because they will always be joined into a single token, independent of which token reaches the C-element first. It is important to distinguish between which critical races affect the test and which do not. Simply detecting existence of a critical race or hazard is not sufficient to conclude that a saf will or will not be detected by the environment. For example, the demultiplexer circuit shown previously in Fig. 3.3 contains a critical race which can cause the system to settle into more than one possible stable state, but the saf is not guaranteed to be detected. Similarly, there are many circuits in which a saf is likely to cause a critical race or hazard (see for example Fig. 4.5)

and we can prove that despite this possibility, the saf is guaranteed to be detected under every possible delay condition. In testing, race simulation therefore implies not just detection of race/hazard, but a continuation of simulation after every branching of behavior caused by a race/hazard in order to determine whether a saf is guaranteed to be detected by the environment.

Theorem 4.2:

At-least-1 test is sufficient to detect every single saf in non-terminal circuits which don't have the token cancellation structure nor the infinite cycle structure.

Proof:

From proof of Theorem 4.1, all inhibitory faults will be detected in the first pass of the circuit, where the first token entering the inhibitory fault will lead to progress violation at the output. All excitory faults will be detected at initialization after stable state is reached because any spurious token will traverse through the non-terminals and emerge at the interface output. □

This class of non-terminal circuits include mod-n counters, sequence generators, and others involving demultiplexers and RCEL's. The at-least-1 requirement instead of at-least-2 will be useful in a future chapter on control point insertion in which the control point needs to be exercised only once for a fault on it to be detected. At-least-2 will be required for any circuit containing a C-element because an excitory fault at one of its inputs is not guaranteed to be detected by at-least-1.

4.2.1 CRF Components and Transitions

We want to precisely identify what properties exist which make the at-least-2 test sufficient to detect all saf's for some circuits and not for others. Since it is the possibility of races which causes problems, we clarify when high-level transitions/components are not affected by possibility of races.

Definition:

A high-level Petri net transition is a *CRF (Critical Race Free) transition* if the consumption of the i^{th} token at input place j due to the firing of the high-level transition always produces a token at output places $f_j(i)$. A *CRF component* is a component which contains only CRF transitions. □

CRF means that the exact consumption schedule of tokens or the firing of other transitions does not affect the CRF transition. Output tokens are produced deterministically. The following components from Fig. 2.1 contain high-level transitions which are CRF transitions.

fork: $f_a(1) = \{x, y\}$, $f_a(2) = \{x, y\}$, ...

C-element: $f_a(1) = \{x\}$, $f_a(2) = \{x\}$, ...
$f_b(1) = \{x\}$, $f_b(2) = \{x\}$, ...

toggle: $f_a(1) = \{x\}$, $f_a(2) = \{y\}$, $f_a(3) = \{x\}$, $f_a(4) = \{y\}$, ...

XOR: $f_a(1) = \{x\}$, $f_a(2) = \{x\}$, ...
$f_b(1) = \{x\}$, $f_b(2) = \{x\}$, ...

arbiter:

(first arbiter transition): $f_a(1) = \{x\}$, $f_a(2) = \{x\}$, ...

(second arbiter transition): $f_b(1) = \{y\}$, $f_b(2) = \{y\}$, ...

demultiplexer:

(first control transition): $f_s(1) = \{t\}$, $f_s(2) = \{t\}$, ...

(second control transition): $f_u(1) = \{v\}$, $f_u(2) = \{v\}$, ...

RCEL:

(first non-join transition): $f_a(1) = \{y\}$, $f_a(2) = \{y\}$, ...

(second non-join transition): $f_b(1) = \{z\}$, $f_b(2) = \{z\}$, ...

All of these are CRF components except for the demultiplexer and RCEL. They are also the ones containing symmetric confusion. The control transitions of the demultiplexer are CRF transitions since they behave as wires. The data transition is

not CRF because the production of tokens at its two output places is dependent on when the control transitions have fired. If the control transitions do not fire at exactly the same order as the fault-free case, the data tokens produced will be different. Similarly, the non-join transitions in the RCEL behave as wires. It is interesting that the arbiter is a CRF component, even though it contains symmetric confusion and output non-determinism. Independently when requests are sent in serially, the high-level transitions behave like wires. When requests are sent in concurrently, one transition will be delayed. However, that transition will eventually be enabled and will still behave like a wire in terms of input consumption/output production. It may be possible for a saf to cause one of the transitions to be permanently disabled, but this is equivalent to a wire having an indefinite delay and is allowed in DI behavior.

We assume that inputs to the arbiter are serialized when deriving the test from the fault-free circuit so that no metastability occurs during the test. Under faulty conditions there can be concurrent inputs and metastability, but the resolution time is not important since a long indefinite delay will appear as progress violation. If we cannot guarantee input serialization of the arbiter in the fault-free circuit, either (i) control points are inserted to enforce serialization, or (ii) a realistic time bound is assumed for metastability resolution to determine the iteration time to wait in the test, even though theoretically the resolution time can be infinite.

Note that even though the individual CRF components are free of critical races, a network formed out of these components will not be guaranteed to be free of critical races. Fig. 4.5 illustrates this possibility. Initially, the sa1 fault causes a spurious token to be deposited at the input of the C-element. Under fault-free conditions, tokens at a and b will be sent in concurrently. Under this faulty condition, there can be two possible stable states: (i) token a is faster, causing cancellation with the spurious token; token b remains at input of C-element, and

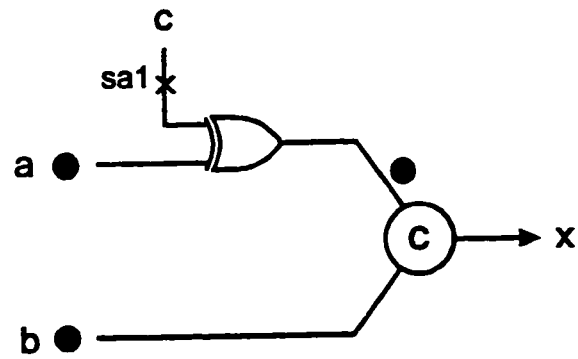


Fig. 4.5: Possibility of critical race in network of CRF components.

(ii) token *b* is faster, joining with the spurious token to produce an output *x*; token *a* remains at input of C-element. Individual CRF components are free of critical races in that given a set of input tokens, the order in which they arrive does not change what outputs are produced. Critical race can occur in the network because token cancellation occurring on interconnecting wires can change the set (number) of input tokens which arrive at a CRF component.

The hypothesis is that at-least-2 will be sufficient to detect every single saf in a circuit containing only CRF components/transitions. The proof will be given in the following sections.

4.2.2 Trace Model

The trace model will be used for proofs involving circuits with faulty behavior.

Definition:

A *trace structure* is defined as a pair (Θ, A), where A is a finite set of alphabet symbols corresponding to high-level net transitions, and $\Theta \subseteq A^*$ is a *trace set*, where A^* denotes the set of all sequences of elements of A. Each element of Θ is called a *trace* $T = t_1; t_2; t_3; \ldots t_n$, where $t_{i=1..n} \in A$ and the semi-colon denotes sequential concatenation. Each transition *t* corresponds to a *consumption-production pair* $(\{a_1,a_2,\ldots\}; \{x_1,x_2,\ldots\})$ where the firing of net transition *t* consumes tokens from the set of input places $\{a_1,a_2,\ldots\}$ and produces tokens in the set of

output places $\{x_1,x_2,\ldots\}$. The sequence $p = u_1; u_2; u_3; \ldots$ is called an ***event sequence*** where u_i is a consumption-production pair. □

A trace represents a particular firing sequence of the high-level Petri net and is obtained by sequentially choosing one enabled high-level net transition to fire at a time. Starting from the initial marking, the set of all possible traces is the trace set. Each trace directly corresponds to an event sequence which describes the behavior of signal transitions on wires. Since we are dealing with a finite test, all trace sets used in this paper are assumed finite. Note that the trace structure used here is slightly different from that used in [81] in that the alphabet symbols represent signal transitions on wires rather than high-level Petri net transition firings used here.

Each net transition firing corresponds to a consumption-production pair where the set of consumption actions are the set of input places which enable the transition, and the set of production actions are the set of places in which output tokens are deposited. For example, the event sequence $p = (\{a,b\};\{x\}); (\{a,b\};\{x\}); (\{a,b\};\{x\}); \ldots$ represents the firing sequence of a C-element. The arbiter behavior is represented by a set of event sequences, eg. $\{p_1, p_2, p_3, \ldots\}$ where $p_1 = (\{a\};\{x\}); (\{a\};\{x\}); (\{b\};\{y\}); \ldots$, $p_2 = (\{b\};\{y\}); (\{a\};\{x\}); (\{a\};\{x\}); \ldots$, etc. because it has an input choice. Superscripts on events will also be used in traces to denote the event occurrence number, similar to that done for pomsets eg. $(\{a^1,b^1\};\{x^1\}); (\{a^2,b^2\};\{x^2\}); (\{a^3,b^3\};\{x^3\}); \ldots$ for the C-element.

An event sequence is chosen to be a sequence of consumption-production pairs for the following reasons. Suppose an event sequence represents the sequence of only consumption of tokens as shown in Fig. 4.6(a). The firing of transition t_1 consumes the token at place a. This representation is inadequate because the current event sequence does not reflect the current state of the net, ie. the firing of transition t_2 instead of t_1 also gives the same event sequence $\{a^1\}$. The

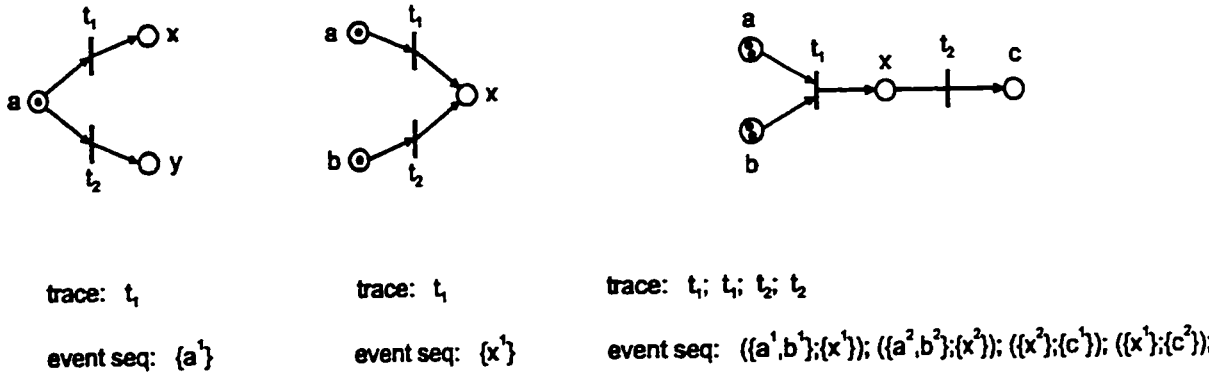


Fig. 4.6: Example event sequences (a) consumption, (b) production, (c) consumption /production pair.

representation of an event sequence as a sequence of only production of tokens is inadequate as well. The firing of either t_1 or t_2 in Fig. 4.6(b) both produce the same event sequence $\{x^1\}$; given only the sequence $\{x^1\}$, the current state of the net is ambiguous since a token can be in place *a* or *b* after the firing. In addition, when two tokens are produced at a place *x*, we can't distinguish whether there is token cancellation or not given some event sequence. Representation of cancellation as the removal of two *x*'s which may not be neighbors in the sequence is cumbersome since we must go back and delete preceding events in the sequence.

Taking an event sequence as a sequence of consumption-production pairs removes the ambiguity. Fig. 4.6(c) illustrates an example sequence containing multiple tokens. Each time a token is *produced*, the token is labeled with an occurrence number of the place, where the occurrence number is the i^{th} time that a token has been produced at that place (event has occurred at that place). In the initial marking, tokens at each place are labeled from 1 to the number of tokens in that place. In any given event sequence, the occurrence number of produced events will always be ordered, but the occurrence number of consumed events may not. This is because two labeled tokens produced and appearing at the same place may proceed concurrently through different paths, and the concurrency will be

represented as a set of event sequences containing interleavings which represent all possible consumption orders of the tokens.

Suppose we have the event sequence: $(\{u\}; \{x^i\}); \ldots; (\{v\}; \{x^{i+1}\}); \ldots; (\{x^i\}; \{w\}); \ldots; (\{x^{i+1}\}; \{y\})$. Multiple token occurs at place x in the net if (i) two tokens x^i and x^{i+1} are produced and there is no consumption of x^i before the production of x^{i+1}, ie. $(\{x^i\}; \{w\})$ does not occur between $(\{u\}; \{x^i\}); \ldots; (\{v\}; \{x^{i+1}\})$, and (ii) x^i or x^{i+1} is eventually consumed in the event sequence. Token cancellation occurs at place x if (ii) does not occur.

Note that it is also possible to reverse the roles of place and transition where net transitions instead of places are labeled with I/O port actions, as is done in Signal Transition Graphs, so that a trace directly represents an event sequence and consumption-production pairs are not required. However, a high-level component transition must then be converted into a set of places which will also complicate matters.

Additional trace operations are defined:

Definition:

Let Θ, Θ' be trace sets and p, q, r be traces. $p;q$ is sequential concatenation of traces and $\Theta;\Theta' = \{p;q \mid p\in\Theta \wedge q\in\Theta'\}$ is concatenation of trace sets. Let $r = p;q$. p is a *prefix* of r, denoted $p \le r$. If q is not empty, then p is a *proper prefix* of r, denoted $p < r$. When $p < r$, p ***subtracted*** from r gives q, denoted $r - p = q$. Θ is ***prefix closed*** if $\forall p,q\in\Theta$: $p;q\in\Theta \rightarrow p\in\Theta$. All trace sets are prefix closed. p is a ***complete trace*** if $p\in\Theta \wedge \neg\exists q$: $[p;q\in\Theta \wedge q\neq\varepsilon]$, where ε is the empty sequence. The ***projection*** of trace T on a set of symbols A, denoted $T\lceil_A$ is defined as follows: If $T = \varepsilon$ then $T\lceil_A = \varepsilon$. If $T = p;a \wedge a\in A$ then $T\lceil_A = (p\lceil_A);a$. If $T = p;a \wedge a\notin A$ then $T\lceil_A = (p\lceil_A)$.

Let Θ^f and Θ^n be the trace set of the faulty and fault-free nets respectively. Θ^f contains ***progress violation*** if $\exists p\in\Theta^f\ \exists q\in\Theta^n$: [$p$, q are complete traces $\wedge$ $p < q$].

Θ^f contains ***safety violation*** if $\exists p \in \Theta^f$: $[p \notin \Theta^n]$. Θ^f contains ***output progress violation*** (***output safety violation***) when $\Theta^f \lceil_{I/O}$ contains progress violation (safety violation), where I/O are the interface input/output port actions.

The definitions for event sequences which directly correspond to traces are defined similar to above. Given the event sequence $p;u$ where $u=(\{a,b,...\}; \{x,y,...\})$ is a consumption-production pair, we say u is enabled at p. u ***in*** p means that u is an element in the event sequence of p. Event (action) a ***in*** p means that a is some event (action) of u where u is in p. $\mathcal{A}(\Theta)$ denotes the set of all actions contained in the consumption-production pairs of every event sequence corresponding to every trace in Θ. □

In terms of nets, every prefix p of a trace gives rise to a net marking, including ε which corresponds to the initial marking. When consumption-production pair u is enabled at p, it also means that the net transition corresponding to u is enabled in the net. Θ^f containing progress violation means that less than the required behavior is generated and implies that some transition is disabled in the faulty net which terminates the execution prematurely. A complete trace p in Θ^f will be a proper prefix of a trace in Θ^n which means that p cannot progress any further in the faulty net but can in the fault-free net. Θ^f containing safety violation means that more than the allowed behavior is generated. Θ^n specifies all possible allowable behavior, and any trace p which appears in Θ^f that does not appear in Θ^n will constitute a violation of allowable behavior. In terms of nets, this implies either that (i) multiple token / token cancellation has occurred at a place which generates behavior in addition to that allowed in a DI net, or (ii) a token is produced at a place when it is not expected; for example, tokens are produced at two different interface output places simultaneously, but a token at only one place is expected. This definition of safety is more general than that of conventional 1-safeness of a net which implies only (i).

4.2.3 Proof requirement

Let $\Theta^f\lceil_{I/O}$ and $\Theta^n\lceil_{I/O}$ be the set of complete trace behaviors of the faulty and fault-free nets projected onto the I/O ports respectively, Fig.4.7. The projection is taken since the tester can only observe any violations at the interface I/O ports. The faulty trace set can only contain three disjoint subsets of traces, traces containing output safety violation, output progress violation, and no output safety/no output progress violation. In order for the test to be guaranteed to detect the saf, the set no

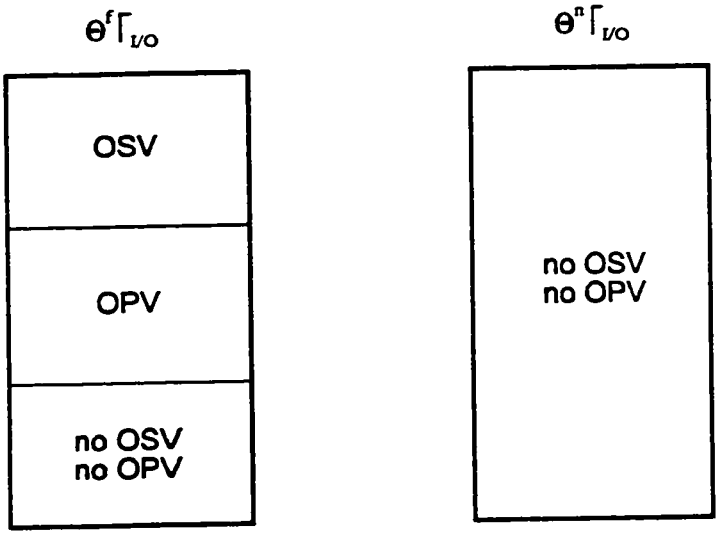


Fig. 4.7: Faulty and fault-free trace sets.

OSV/no OPV must be empty since the existence of a single trace in that set implies that particular trace execution will be enough to fool the tester into not detecting the saf. The emptiness of the set will be proved by showing that if a trace in $\Theta^f\lceil_{I/O}$ does not contain OSV (if it does, then the fault will immediately be detected), then it must contain OPV.

4.2.4 Initial axioms

The following axioms apply for fault-free DI nets.

[A1]: (1-safeness axiom)
The composed high-level net is 1-safe. □

[A2]: (Net composition axiom)

Open nets are composed by connection of single output place to single input place only (co-location of output place and input place). □

[A3]: (Choice free axiom)

The high-level net representation of basic components do not contain choice places. □

[A4]: (CRF axiom)

All high-level transitions in the net are CRF transitions. □

Axioms [A1] to [A3] ensure that the high-level circuit net is both 1-safe and *persistent*. The common definition of persistency is that any transition which is enabled remains enabled until it fires. This implies that any existence of a choice place will make the net non-persistent since the choice place enables two or more transitions and the firing of any one of these transitions will disable the other transitions. Requiring that the net be 1-safe and choice free in [A1] to [A3] guarantees that the high-level net is persistent because once a token appears at a place to enable a high-level transition, that token cannot be removed through token cancellation (violation of 1-safeness) or the firing of another transition to remove that token, where the removal can only occur if that place was a choice place.

The low-level net on the other hand is not persistent since it contains choice places. Even "deterministic" components such as the toggle require choice places in its low-level net to specify because a token at a toggle input chooses alternately to go to either one of its outputs.

The persistency property of the high-level net will be defined in terms of traces. Let Θ be a trace set, p,q be event sequences corresponding to traces of Θ, and u,v be consumption-production pairs. An event sequence being an element of a trace set is assumed to mean that the trace directly corresponding to the event sequence is in the trace set. The following two lemmas apply for fault-free nets and

assume axioms [A1] to [A4] are true. In all the lemmas in this chapter, consumption-production pairs do not carry superscripts.

Lemma 1: (Persistency #1)

$\forall_{p,q,u}$: $p;u \in \Theta \wedge p;q \in \Theta \wedge \neg(u \text{ in } q) \rightarrow p;q;u \in \Theta$.

(If transition corresponding to u is enabled at p, then that transition remains enabled until it fires.)

Proof:

Suppose this is false, then $\exists_{p,q,u}$: $p;u \in \Theta \wedge p;q \in \Theta \wedge \neg(u \text{ in } q) \wedge \neg(p;q;u \in \Theta)$.

$p;u \in \Theta \rightarrow$

When the state of the net is at p, the net transition t corresponding to $u = (\{a,...\}; \{x,...\})$ is enabled to be fired and there is an input token at every place in $\{a,...\}$.

$p;q \in \Theta \wedge \neg(p;q;u \in \Theta) \rightarrow$

At state $p;q$, $\{a,...\}$ no longer enables t which means there is at least one missing token, say at $s \in \{a,...\}$.

$\rightarrow$ Since s is not a choice place by [A3] and t has not fired from $\neg(u \text{ in } q)$, the token at s could only be removed during execution of q if s has received multiple tokens causing token cancellation.

$\rightarrow$ violation of 1-safeness [A1]. □

Definition:

Let $u = (\{a,...\}; \{x,...\})$ and $v = (\{b,...\}; \{y,...\})$ be consumption-production pairs. u and v are *non-interfering* if and only if $\{x,...\} \cap \{b,...\} = \varnothing$ and $\{y,...\} \cap \{a,...\} = \varnothing$. □

u and v are *non-interfering* means that the firing behavior of transitions corresponding to u and v are independent of each other (concurrent) as well as not causing token cancellation or enabling/disabling of each other. In addition, $\{a,...\} \cap \{b,...\} = \varnothing$ must be true, otherwise the choice freeness axiom [A3] is violated. $\{a,...\} \cap \{x,...\} = \varnothing$ is not required in the case that some output of a component is fed back to one of its inputs.

Lemma 2: (Persistency #2)

$\forall_{p,q,u,v}$: $p;u;v;q \in \Theta \leftrightarrow p;v;u;q \in \Theta$, where u and v are non-interfering. (The firing of one transition does not disable the other.)

Proof:

Let transitions t_1 and t_2 correspond to $u = (\{a,...\}; \{x,...\})$ and $v = (\{b,...\}; \{y,...\})$ respectively. Suppose $p;u;v;q \in \Theta$ is true.

At state p, the net must contain input tokens at $\{a,...\}$ since we have $p;u$. At state p, the net must also contain input tokens at $\{b,...\}$ since at state $p;u$ after t_1 fires, we have $p;u;v$.

Therefore at state p we can have $p;v;u$ from the CRF axiom [A4] by firing enabled transitions t_2 then t_1. Since the net marking is the same at $p;u;v$ and $p;v;u$, we have $p;u;v;q \in \Theta \rightarrow p;v;u;q \in \Theta$ and similarly for the converse. □

Note that Lemma 2 would not be true if the CRF axiom did not apply because changing the firing order of transitions can cause the high-level transitions to produce different output tokens, such as the case of firing the data and control transitions in reverse order.

The following are the remaining axioms.

[A5]: (At-least-2 axiom #1)

$\forall$*complete traces* $p \in \Theta$ $\forall a \in \mathcal{A}(\Theta)$: a^1 in $p \wedge a^2$ in p, where a is a consumption event. (Every action appears at least twice in every complete trace.) □

At-least-2 is defined in terms of consumption because two tokens/events produced at a place may cancel before the tokens are consumed, and a faulty trace set would incorrectly satisfy this axiom.

[A6]: (At-least-2 axiom #2)

$\forall$*complete traces* $p \in \Theta$ $\forall a \in \mathcal{A}(\Theta)$: $(\exists x, e_{1..n}\ p = ...; (\{a^1,..\}; \{e_1,..\}); ...; (\{e_1,..\}; \{e_2,..\}); ...; (\{e_2,..\}; \{e_3,..\}); ...; (\{e_3,..\}; \{e_4,..\}); ...; ...; (\{e_{n-1},..\}; \{e_n,..\}); ...; (\{e_n,..\}; \{x,..\}); ...) \wedge (\exists x, e_{1..n}\ p = ...; (\{a^2,..\}; \{e_1,..\}); ...; (\{e_1,..\}; \{e_2,..\}); ...; (\{e_2,..\}; \{e_3,..\}); ...; (\{e_3,..\}; \{e_4,..\}); ...; ...; (\{e_{n-1},..\}; \{e_n,..\}); ...; (\{e_n,..\}; \{x,..\}); ...)$

where $e_{1..n}$ are any set of events, x is some interface output event, and a^1 (a^2) is the first (second) event of a.

(There exists some consumption-production path from the first and second event of every action to the environment.) □

Since the ordering of consumption of event numbers in a trace is not preserved in the presence of race in a faulty net, a^1 is also included in the axiom.

A test *Nt* is an *at-least-2* test if it satisfies [A5] and [A6]. Some of these axioms were used in Chapter 3 but is formalized in this chapter.

4.2.5 Sufficiency Proof

The following trace sets are defined:

Θ^n = $Nt \bullet Nc$, trace set of normal fault-free net.

Θ^f = $Nt \bullet Nc'$, trace set of faulty net with possible token cancellation.

Θ^{fb} = $Nt \bullet Nc'$, trace set of faulty net without token cancellation (infinite buffer net).

Θ^{r1} = trace set obtained by executing test *Nt* on *Nc'* where the initial spurious token (if any) at saf place *s* is removed from *Nc'*.

Θ^{r2} = trace set obtained by continuing execution of Θ^{r1} with the initial spurious token placed back in *s*.

Θ^r = $\Theta^{r1};\Theta^{r2}$

Θ^n is the trace set (event sequence set) obtained through normal execution of the test on the fault-free net. Since we are modeling a DI CRF high-level net, Θ^n satisfies all the axioms [A1] to [A6]. Θ^f is the trace set obtained by running the test on the faulty net where the faulty net may have both multiple token places or token cancellation. Θ^{fb} is the trace set of the faulty net which may have multiple token places (place is an infinite buffer), but no token cancellation. Θ^r is used as a representation trace set which is a special case firing of the faulty net where $\forall p \in \Theta^r$: $p \in \Theta^f$. Suppose there exists a spurious token at place *s* in the faulty net at

initialization. Θ^r will consist of two phases, Θ^{r1} which is the execution of the test with the same initial state as the faulty net but with the spurious token at *s* removed or equivalently held in place, and in phase 2, Θ^{r2} which is the continued execution of the net with the spurious token placed back in *s*. In the case that the faulty net does not contain an initial spurious token, Θ^{r2} is empty. Θ^r is an intermediate trace set used to relate Θ^f to Θ^n in proving that the at-least-2 test is sufficient for detecting all single saf's in CRF networks. The proof strategy relates pairs of trace sets to link Θ^f with Θ^n and roughly involves the goal of proving

$$\#E(\Theta^f) \leq \#E(\Theta^{fb}) \leq \#E(\Theta^r) < \#E(\Theta^n)$$

where $\#E(\Theta)$ denotes the number of interface output port events in any single complete trace of Θ, ie. that the number of output events produced by every trace in Θ^f, Θ^{fb}, and Θ^r must always be less than that produced in testing the fault-free circuit; hence output progress violation will be detected, if output safety violation has not been detected.

Lemma 3 shows that the infinite buffer net always contains every possible behavior of the net which can have token cancellation. The buffered net behavior can then be related to the representation trace behavior without the extra complications of token cancellation.

Definition:

Let Θ be a trace set and the complete traces of Θ be numbered in some arbitrary order. Θ_i denotes the i^{th} complete trace of Θ. □

Lemma 3:

$\forall_j \exists_k$: $\Theta^f_j \leq \Theta^{fb}_k$.

Proof:

The net and initial marking corresponding to Θ^f_j and Θ^{fb}_k are identical with the exception that at some places, token cancellation may occur in Θ^f_j. Suppose during the firing that a pair of tokens occur at a place in Θ^f_j which will cancel.

Holding the pair of tokens at that place for the trace of Θ^{fb}_k implies that every transition fireable for Θ^{f}_j will also be fireable for Θ^{fb}_k, resulting in the same trace. When the net of Θ^{f}_j contains no more enabled transitions, the net of Θ^{fb}_k contains pairs of tokens, allowing the extension of the trace of Θ^{fb}_k. Therefore, there must always exist a trace in which Θ^{f}_j will be a prefix of Θ^{fb}_k. □

The following Lemma 4 and Lemma 5 will be used as support lemmas for Lemma 6.

Lemma 4:

$\forall_{p,q,u}$: $p;u \in \Theta \wedge p;q \in \Theta \wedge \neg(u \text{ in } q) \rightarrow (\forall_{u' \text{ in } q}$: $u \wedge u'$ are non-interfering).

(If u is enabled at p, until u fires, no one that interferes with u is enabled.)

Proof:

Suppose otherwise and interference occurs in a sequence $q = q';u'$, i.e. u and q' are not interfering but u and u' are. Let $u = (\{a,...\}; \{x,...\})$ and $u' = (\{b,...\}; \{y,...\})$.

Case 1: $\{y,...\} \cap \{a,...\} \neq \varnothing$

$p;u \in \Theta \wedge p;q' \in \Theta \wedge \neg(u \text{ in } q') \rightarrow p;q';u \in \Theta$ (by Lemma 2)

But $p;q \in \Theta$

$\rightarrow p;q';u' \in \Theta$

$\rightarrow$ both u and u' are enabled at $p;q'$.

Firing u' first causes token cancellation at $\{y,...\} \cap \{a,...\}$ and thus $p;q';u';u$ cannot hold, contradicting persistency of Lemma 2.

Case 2: $\{x,...\} \cap \{b,...\} \neq \varnothing$

Similarly, firing u at $p;q'$ causes token cancellation at $\{x,...\} \cap \{b,...\}$, so u' cannot fire at $p;q';u$. Therefore $p;q';u;u' \in \Theta$ cannot hold.

But from the persistency property, we know

$p;q';u' \in \Theta \wedge p;q';u \in \Theta \wedge \neg(u' \text{ in } u) \rightarrow p;q';u;u' \in \Theta$. (by Lemma 2)

Thus a contradiction is also reached. □

Lemma 5:

Θ^r satisfies the persistency Lemmas 1 and 2.

Proof:

Given a faulty net corresponding to Θ^r and a fault-free net corresponding to Θ^n, Θ^r is derived by firing transitions of the faulty net which are identical to that of the fault-free net, and both net markings will be identical at every step (except possibly at saf place s). Suppose there is a violation of persistency for the faulty net where at some prefix p, u is enabled to be fired and then becomes disabled later at $p;q$. Because both net markings are identical, the same must hold for the fault-free net, which is a contradiction as Θ^n must satisfy Lemma 1 and Lemma 2 (the difference in state due to placing a token in s in phase 2 of Θ^r simply delays the enabling of a transition). □

Lemma 6 says that any event (consumption-production pair) which appears in the trace of the buffered net will also appear in the trace of the representation net. This means that the buffered traces cannot have more events, including interface output events, occurring than the representation traces. The representation traces are used as an upper bound on the possible number of events which may occur so that we can show that the representation traces can never have enough events to simulate the normal traces.

Lemma 6:

$\forall_j \exists_k \forall_i$: u_i in $\Theta^{fb}{}_j \rightarrow u_i$ in $\Theta^r{}_k$.

Proof:

We use induction. Suppose $\Theta^{fb}{}_j = u_1;u_2; \ldots ;u_i;u_{i+1}; \ldots$, and s is the saf place.

Basis:

Case 1: $\neg(s^1 \in u_1)$

Let t_1 be the transition corresponding to the consumption-production pair $u_1=(\{a\}; \{x,\ldots\})$. The initial state (net marking) of Θ^{fb} and Θ^r are identical except possibly at s; t_1 enabled in net of Θ^{fb} implies t_1 is also enabled in net of Θ^r. The persistency of Θ^r by Lemma 5 guarantees that t_1 will eventually

fire. Moreover, t_1 is a CRF transition by [A4] and the firing of t_1 always produces a token at output places $f_a(1)=\{x,\ldots\}$. u_1 therefore will appear in Θ^r_k.

Case 2: $s^1 \in u_1$

Place s must contain the spurious token, otherwise u_1 is not enabled. The spurious token will be put in s in phase 2 of Θ^r, and by Lemma 5 and [A4], u_1 will appear in Θ^r_k.

Induction Step:

Assume $u_1, \ldots, u_i$ appear in both Θ^{fb}_j and Θ^r_k. We need to show that if u_{i+1} appear in Θ^{fb}_j then it will also appear in Θ^r_k.

Let $\Theta^r_k = p;u_r;q;u_{i+1};\ \ldots$ where $u_1, \ldots, u_i$ appear in $p;u_r;q$ where u_r is the latest firing of Θ^r_k which enables u_{i+1}. (In general, the firing of more than one consumption-production pair may enable u_{i+1} but we take the u_r which represents the latest occurrence in the trace of Θ^r_k.)

(1) $\neg\,(p;u_{i+1} \in \Theta^r)$ must be true, otherwise

$p;u_{i+1} \in \Theta^r \wedge p;u_r \in \Theta^r \wedge \neg(u_{i+1}$ in $u_r) \rightarrow u_{i+1} \wedge u_r$ are non-interfering (by Lemma 4)

$\rightarrow$ contradiction

(2) $p;u_r;u_{i+1} \in \Theta^r$ must be true since the firing of u_r enables some transition corresponding to u_{i+1}

(3) $p;u_r;u_{i+1} \in \Theta^r \wedge p;u_r;q \in \Theta^r \wedge \neg(u_{i+1}$ in $q) \rightarrow p;u_r;q;u_{i+1} \in \Theta^r$ (by Lemma 1)

□

Lemma 6 is important because it makes use of the CRF axiom (as well as Lemma 2 and hence Lemma 4 which is used here). The representation trace set represents a particular firing sequence behavior in which the spurious token is held in place so that there cannot be multiple tokens at a place in its net. The buffered net may have multiple tokens, and tokens appearing at a place may arrive at a different order than that of the representation net (the ordering of the trace events

in Θ^{fb} may be different from Θ^r). But because of the functional property of the CRF transition, this out of order occurrence does not affect what is produced by the CRF transition. Any transition which can be enabled, regardless of when it is enabled due to the out of order input tokens, will always produce the same output tokens, and so the traces will always contain the same events, regardless of event ordering.

Lemma 7 now relates the representation trace set with the normal trace set. It proves that every trace in the representation trace set Θ^r must be a proper prefix of at least one trace in Θ^n. This will guarantee that Θ^r and thus the faulty net will not have enough events to simulate Θ^n.

Lemma 7:

$\forall_k \exists_j$: $\Theta^r_k < \Theta^n_j$.

Proof:

Case 1: *Nc* has same initial marking as *Nc'* (no spurious token)

Suppose $\Theta^r_k = u_1;u_2; \ldots ;u_m; \ldots ;u_n$. We show there exists $\Theta^n_j = u_1;u_2; \ldots ;u_m \ldots ;u_n \ldots$.

Basis: There exists $\Theta^n_j = u_1; \ldots$ since whatever transition is enabled in *Nc'* is also enabled in *Nc* at initialization.

Induction step: Assume there exists $\Theta^n_j = u_1;u_2; \ldots ;u_m \ldots$. We need to show the extension to $\Theta^n_j = u_1;u_2; \ldots ;u_m;u_{m+1}; \ldots$. *Nc'* differs from *Nc* only in a missing arc from a transition to its output place *s*. Thus the tokens produced in firing $u_1;u_2; \ldots ;u_m$ in *Nc'* is also produced in firing the same sequence in *Nc*. The persistency, Lemma 1 and Lemma 2, of Θ^n ensures that one of its event sequences, Θ^n_j, will fire u_{m+1} to become $\Theta^n_j = u_1;u_2; \ldots ;u_m;u_{m+1}; \ldots$, giving $\Theta^r_k \leq \Theta^n_j$. Since s^2 does not appear in Θ^r_k (*s* is disconnected from its producer transition) but s^2 does appear in Θ^n_j from [A5], $\Theta^r_k < \Theta^n_j$ for some *j*.

Case 2: *Nc* has initial token at *s* compared to missing one in *Nc'*

(spurious token cancels normal token at *s*).

Same as Case 1, since the extra token in *Nc* at most enables some additional transitions.

Case 3: *Nc'* has initial token at *s* compared to none in *Nc*

(spurious token placed at *s* at initialization).

Since the restricted firing of Θ^r inserts the spurious token into *s* of *Nc'* only in phase 2, the proof of phase 1 is the same as Case 1: the sequence of states (markings) of *Nc'* and *Nc* are identical in phase 1 and $\Theta^r_k = \Theta^n_j = u_1;u_2; \ldots ;u_m$.

In phase 2, the inserted token s^1 must appear in u_{m+1} of Θ^r_k since s^1 is required to continue the execution. Let *t* be the sink transition in *Nc'* which would have produced s^1. At the end of phase 1, *t* must have been fired in generating Θ^r_k since it prevents the continuation of the sequence. The corresponding *t* in *Nc* must also have fired, but in this case producing s^1. Thus u_{m+1} of Θ^n_j also enables some transition. The firing of *Nc'* and *Nc* thus continues in phase 2 with identical states until s^2 is missing in *Nc'*, giving $\Theta^r_k < \Theta^n_j$. □

Lemma 8 strengthens the result of Lemma 7: every trace in Θ^r containing less events than one trace in Θ^n must mean that one interface output event is not produced, which will lead to output progress violation.

Lemma 8:

If $\forall_k \exists_j$: $\Theta^r_k < \Theta^n_j$, then $\Theta^n_j - \Theta^r_k$ must contain at least one consumption-production pair in which an interface output event is produced.

Proof:

From [A6], s^2 must lead to some output x^i produced in Θ^n_j. Since s^2 is not in Θ^r_k, x^i is not in Θ^r_k. Therefore $\Theta^n_j - \Theta^r_k$ must contain the output x^i. □

The above lemmas apply to a single trace in Θ^n. The following shows that the events in one trace will appear in every other trace of Θ^n.

Definition:

$p \equiv q$ if and only if $p \in \Theta \wedge q \in \Theta \wedge (\forall_u$: *u* in $p \leftrightarrow u$ in $q)$. (*p* and *q* contain the same set of consumption/production pairs and both are in the same trace set Θ.) Let all the transitions in a net be labeled by strictly increasing integers start-

ing from 1, Θ^n be the corresponding trace set, and the set of complete traces in Θ^n be $\{\Theta^n_1, \Theta^n_2, \ldots, \Theta^n_j, \ldots, \Theta^n_m\}$. The trace Θ^n_c for some $j=c$ is said to be the (complete) ***canonical trace*** of Θ^n if the following firing rule is used to generate it: starting from the initial marking, the next transition selected to be fired is the enabled transition which is labeled with the lowest integer. □

Lemma 9:

$\forall_j$: $\Theta^n_c \cong \Theta^n_j$, where Θ^n_j and Θ^n_c are complete traces.

Proof:

Let $\Theta^n_j = u_1;u_2; \ldots$, and $\Theta^n_c = u_1';u_2'; \ldots$.

(1) Claim: u_1 must be in Θ^n_c.

Suppose otherwise.

$u_1 \in \Theta^n \wedge \Theta^n_c \in \Theta^n \wedge \neg(u_1 \text{ in } \Theta^n_c)$

$\rightarrow \Theta^n_c;u_1 \in \Theta^n$ (by Lemma 1)

$\rightarrow \Theta^n_c$ is incomplete

$\rightarrow$ contradiction.

Thus $\Theta^n_c = u_1'; \ldots ;u_i';u_1; \ldots$

(2) From lemma 4, u_1 cannot interfere with $u_1'; \ldots ;u_i'$.

Thus $\Theta^n_c = u_1'; \ldots ;u_i';u_1; \ldots$

$\cong u_1'; \ldots ;u_1;u_i'; \ldots$ (by Lemma 2)

$\cong u_1;u_1'; \ldots ;u_i'; \ldots$ (by Lemma 2)

(3) Repeating (1) and (2) on u_2,u_3, and so on, we get

$\Theta^n_c \cong u_1;u_2; \ldots$

$= \Theta^n_j$. □

Lemma 9 indicates that given a set of traces representing a CRF DI network, there always exists a single canonical trace representing the set of traces. This means that every complete trace in Θ^n must be of the same length and must contain an identical set of events (production-consumption pairs). (In terms of pomsets, the trace set Θ^n would be represented as a single deterministic pomset.)

Lemma 9 generalizes Lemmas 7 and 8 which applies for only a single trace in Θ^n. Since every trace in Θ^n contains the same number of events, every trace in Θ^r must contain less events than every trace in Θ^n from Lemma 7. These missing events must contain an interface output event from proof of Lemma 8.

Theorem 4.4:

At-least-2 test is sufficient to detect every single saf in circuits containing only CRF components.

Proof:

Suppose otherwise, ie. there exists a schedule Θ^f_i in which a saf is not detected. This leads to a contradiction as follows:

Lemma 3 $\rightarrow$ Θ^f_i does not contain more occurrences of any event than Θ^{fb}_k for some k.

Lemma 6 $\rightarrow$ Θ^{fb}_i does not contain more occurrences of any event than Θ^r_k for some k.

Lemma 7 $\rightarrow$ Θ^r_k and thus Θ^f_i does contain fewer occurrences of some events than *some* Θ^n_j.

Lemma 8 $\rightarrow$ Θ^r_k and thus Θ^f_i contains at least one interface output event fewer than *some* Θ^n_j.

Lemma 9 $\rightarrow$ Θ^f_i must contain at least one interface output event fewer than *all* Θ^n_j.

Thus the contradiction. The missing output event must eventually lead to output progress violation, if output safety violation is not already detected. □

4.3 Structural Theorems Involving Non-CRF Components

Theorem 4.4 only applied to circuits containing CRF components. It covers many types of circuits, including micropipeline buffers, arbitration circuits, and many deterministic control circuits. But some types of control circuits such as finite state machines involve choice of inputs and require components such as the

demultiplexer or RCEL which are not CRF components, but do contain some CRF transitions. We now consider structural theorems for circuits which contain non-CRF components.

It is sometimes possible to partition a circuit such that one part contains only CRF transitions and the other part can contain any type of transition, as shown in Fig. 4.8. T is the test net, net S contains only CRF transitions and the saf, and net A may contain any transition. The partitioning is such that there is no token flow between S and A. For the basic components shown in Chapter 2, the nets S and A

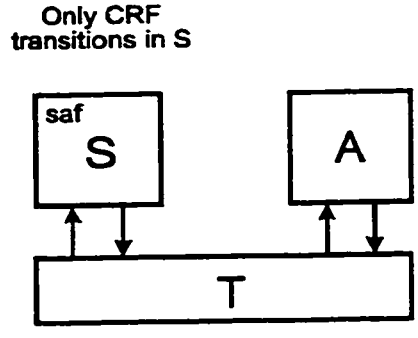


Fig. 4.8: Partitioned circuit structure.

will be disjoint involving all components except the demultiplexer and RCEL. It is possible to form a partition such that half a non-CRF component may be in S and the other half is in A. For example, the control lines of the demultiplexer consists of CRF transitions and can be placed in net S, while the data lines are non-CRF transitions and placed in net A. Since these high-level transitions have a low-level effect between each other, the partitioning can be justified by looking at the low-level net of, for example, the demultiplexer in Fig. 4.9. The component is partitioned such that the interface of the nets are two shared places, in the same manner that two open nets are composed, except that these are internal places rather than input/output places. These internal places record the state of the component. It can be seen that the firing of any low-level transition in A does not

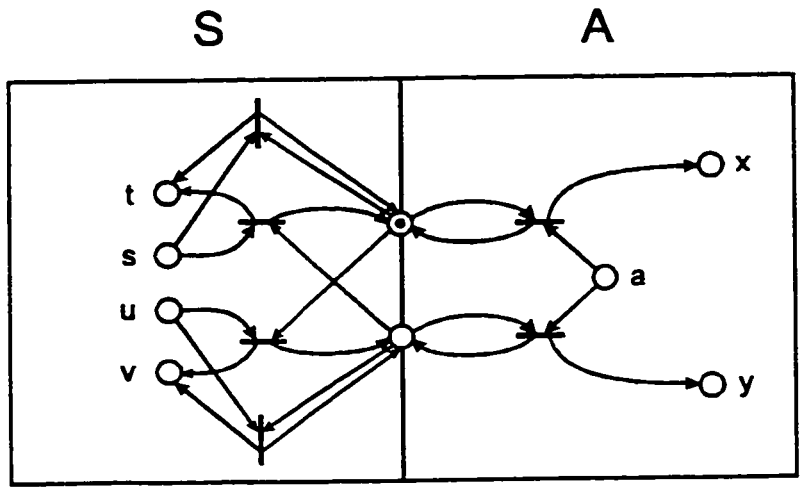


Fig. 4.9: Low-level partitioning of demultiplexer component.

affect *S*. Any consumption of the interface token immediately returns that token back to the same place, ie. no change of state. The firing of transitions in *S* on the other hand can change the state of the interface tokens, which is why the data transitions in *A* are not CRF. In effect, *S* is disjoint from *A* and cannot be affected by *A*. The same applies for the low-level transition partitioning of components such as the arbiter and RCEL.

Theorem 4.5:

At-least-2 test is sufficient to detect every single saf in *S*.

Proof:

Suppose the theorem is false and the saf is not detected, then interface of *S-T* and *A-T* must have no violation

→ interface of *S-T* has no violation

→ contradiction to Theorem 4.4 since composed net *S•T'* for restricted test *T'* is an at-least-2 test involving only CRF transitions. □

The following hypotheses are possible on the structural graphs of the circuit nets:

Hypothesis:

At-least-2 test is sufficient to detect a saf in which there does not exist a path from the saf to any vertex of a non-CRF component. □

⇒ Corollary to Theorem 4.5.

Hypothesis:

At-least-2 test is sufficient to detect a saf in which there does not exist a path from the saf to any vertex of a demultiplexer data node. □

⇒ False from *C2* for $sa1_3$.

Hypothesis:

At-least-2 test is sufficient to detect a saf in which there does not exist a path from the saf to any vertex of a demultiplexer control node. □

⇒ False from *C3*.

Hypothesis:

At-least-2 test is sufficient to detect a saf in which there does not exist a path from the saf to any vertex of a demultiplexer reset control node. □

⇒ False from *C3*.

Hypothesis:

At-least-2 test is sufficient to detect a saf in which there does not exist a path from the saf to any vertex of a demultiplexer set control node. □

⇒ False from *C3* and *C2* for $sa1_2$ and $sa1_3$.

Hypothesis:

At-least-2 test is sufficient to detect every single saf in a circuit whose structural graph is acyclic. □

⇒ False from *C3*.

Where *C1*, *C2*, and *C3* are the counterexamples of Fig. 3.3, Fig. 3.4, and Fig. 4.10 respectively. Several other more specific hypotheses were also explored, but

the result turned out to be negative. In fact, these counterexamples were derived from trying to prove these hypotheses. Even saf's in acyclic circuits are not always guaranteed to be detected by an at-least-2 test, which is somewhat counterintuitive since a spurious token would not have a chance to recycle and produce critical race inputs to components, such as concurrently to both the control and data of a demultiplexer. One would expect the behavior preceding an excitory saf in an acyclic circuit to be normal and the saf to be detected on the second phase when normal tokens reach the saf disconnect. But as seen from counterexample *C3*, the environment plays an important role; composing the environment with the open acyclic net closes the system and allows tokens to recycle.

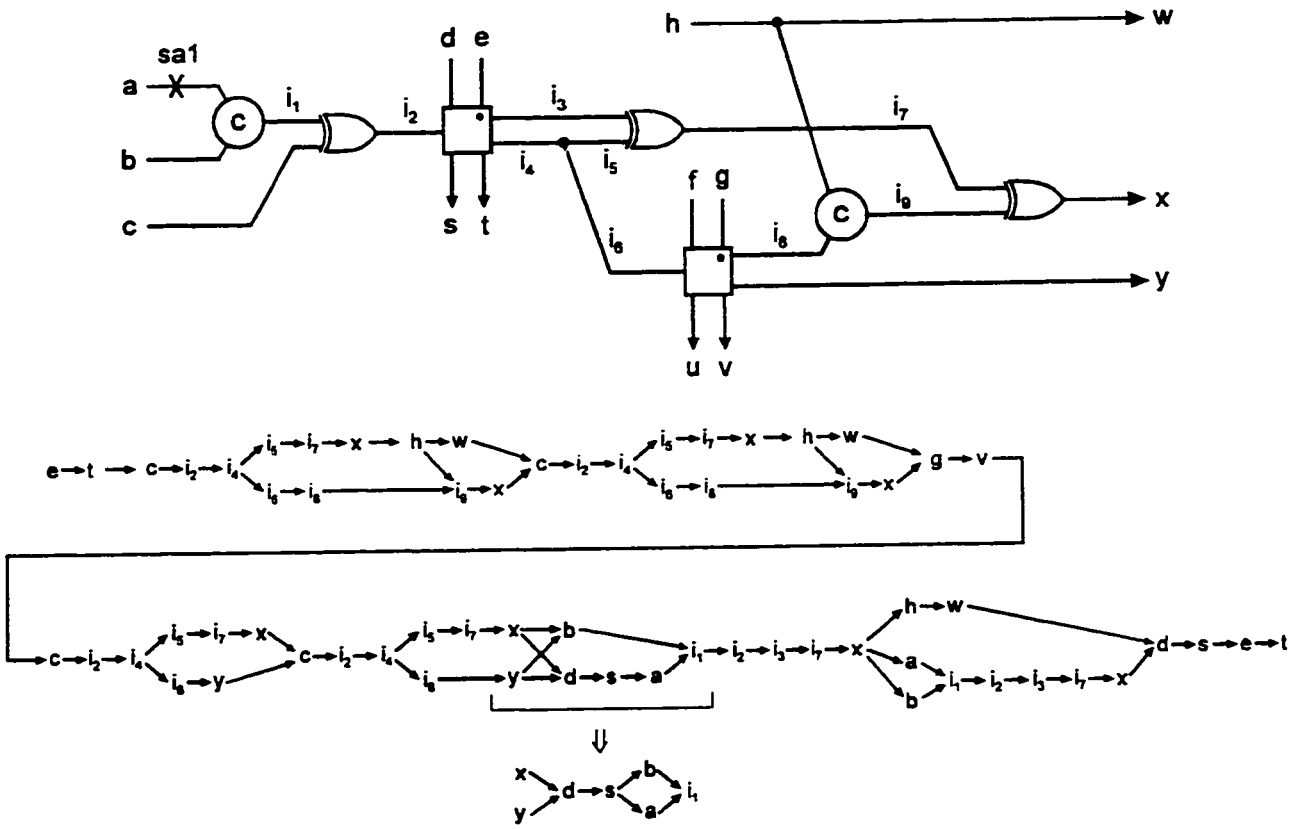


Fig. 4.10: C3: counterexample to at-least-2 detecting every saf in an acyclic circuit.

For the acyclic circuit of Fig. 4.10, the problem occurs when there is a critical

race when sending the *b* and *d* inputs concurrently. Under fault-free conditions, the *d* input would have reset the demultiplexer first before the *b* token enters the data input. Under the excitory saf fault, the spurious token prematurely enables the *a* input of the C-element so that the token from the *b* input of the C-element can traverse the demultiplexer data input before it is properly reset. The test can be changed by serializing the *b* and *d* inputs to make the at-least-2 test be able to detect the saf, as shown. This demonstrates that serialization of inputs can eliminate critical race conditions and make the circuit more testable. It remains to be seen how useful serialization can be. This also shows that single input fundamental mode used to minimize race conditions also can apply for DI circuits.

Conjecture:

A single input fundamental mode at-least-2 test is sufficient to detect every single saf in a circuit whose structural graph is acyclic. □

From these hypotheses, there does not appear to be many structural theorems possible. However, more detailed circuit/component dependent theorems may exist and is left for possible future research. Total coverage by structural theorems is not necessary, since uncovered nodes can be covered using observation points.

4.4 Behavioral Theorem

The following theorem can cover a saf reaching demultiplexer data nodes which could not be covered by the previous structural theorems. It is a behavioral theorem in that it checks for a property of the fault-free behavior rather than the static structure of the circuit. The check is not expensive because it is performed on the fault-free behavior. The proof relies on the partial state results of Chapter 5, so we defer the definitions etc. to that chapter.

Theorem 4.6:

Let an excitory saf be at location s, and p be the partial order (low-level net) generated by the spurious token at initialization. If p appears in the fault-free behavior (as a midfix starting at s^1), and $\mathcal{A}(p) \cap \mathcal{A}(\underline{\mathbf{M}}s^1)$ is empty, where $\underline{\mathbf{M}}s^1$ is the maximal prefix of s^1 in the fault-free behavior, then the at-least-2 test detects the saf.

Proof:

Let the fault-free behavior be $P = i \;;;\; q \;;;\; p \;;;\; r$ and the faulty behavior be $P' = i' \;;;\; p' \;;;\; q' \;;;\; r'$ as shown in Fig. 4.11. Prefix i' contains s^1 while i does not, q contains s^1 while q' does not, $p = p'$, and $r' = r - (P - \underline{\mathbf{M}}s^2)$. Prefix i' corresponds to the initial state before stabilization of the spurious token, ie. prefix i contains events of places containing tokens in the initial net marking, and $i' = i \cup \{s^1\}$ since the faulty circuit initially deposits a spurious token at place s. P' represents the new reordered behavior when the excitory saf exists. The requirement is to show that P' is a valid collision free behavior (1-safe execution) and will lead to progress violation. We proceed by checking each pomset segment in P':

(i) The initial prefix i' appears in the faulty circuit.

(ii) Can execute p' since

$\Pi(i',\mathcal{B}) = \Pi(q,\mathcal{B}) = \{s\}$, where $\mathcal{B} = \mathcal{A}(p) \cup \{s\}$.

The partial state (on set) must be $\{s\}$ because it is the only place containing a token in the subnet we are interested in and s generates p. Execution of p' will not lead to collisions since actions of p' are disjoint from q'.

(iii) Can execute q' since

$\Pi(i';;p',\mathcal{B}) = \Pi(i,\mathcal{B}) = \mathcal{A}(i^\circ)$, where $\mathcal{B} = \mathcal{A}(q) \cup \mathcal{A}(i^\circ)$.

Starting from the current prefix $i';;p'$, we jump to the new prefix i in the fault-free behavior in order to execute $q - \{s^1\}$. The partial state must be $\mathcal{A}(i^\circ)$ at prefix i of the fault-free circuit. It is also $\mathcal{A}(i^\circ)$ at prefix $i';;p'$ of the faulty circuit since s^1 has been executed and is no longer a last event. The partial set of actions $\mathcal{B}$ considered should include all actions in q which will be the pomset segment executed, plus the last events of the prefix preceding q which is i°.

(iv) Can execute r' since

$\Pi(i';;p';;q',\mathcal{B}) = \Pi(i;;q;;p,\mathcal{B}) = \mathcal{A}(q^\circ) \cup \mathcal{A}(p^\circ) - \{s\}$, where $\mathcal{B} = \mathcal{A}(r)$.

Starting from the current prefix $i';;p';;q'$, we jump to the new prefix $i;;q;;p$ in the fault-free behavior in order to execute r'.

Since the faulty net contains a disconnect immediately preceding place s, a token cannot be deposited at s to generate event s^2, and progress violation will be observed at the interface output. The fault-free circuit will execute P and not generate a violation. Note that the same test is given to both the faulty and fault-free circuits as observed by the environment. Only the internal behavior of p is reordered due to the saf. □

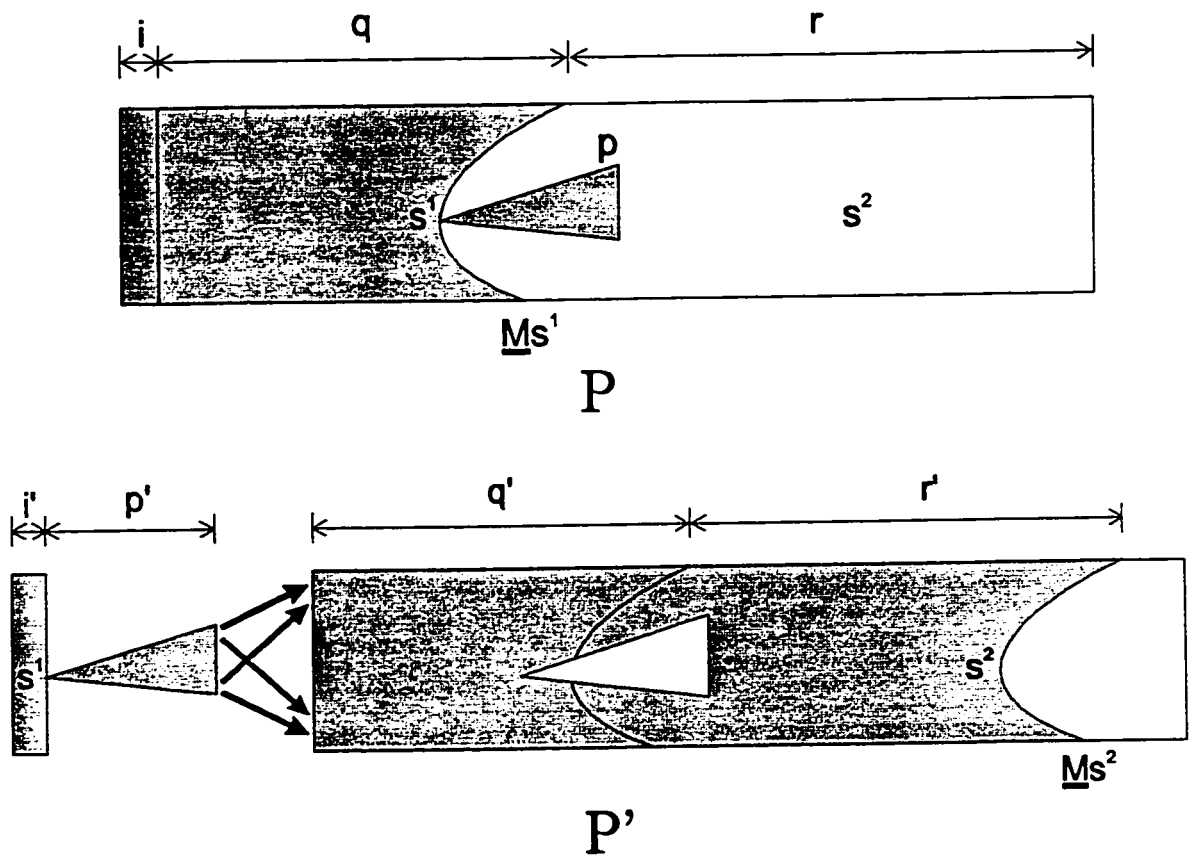


Fig. 4.11: Fault-free behavior P and faulty behavior P' resulting from an excitory fault at s.

Conceptually, the spurious token due to the excitory fault is following exactly the same path as would the first normal token deposited at place s in the fault-free circuit, only it is executed prematurely. After executing $i;;q;;p$ the global state (net marking) of the faulty and fault-free nets would be identical, except for the disconnect in the faulty net which prevents the firing of s^2. The theorem applies for the low-level net because the internal places of state components must be included; otherwise, the spurious token may pass through the control of a demultiplexer and affect normal tokens passing through the data, even though the external control and data actions are disjoint.

Chapter 5

Sensitization

The concept of sensitization is to detect excitory faults by sensitizing the spurious token which exists in the faulty circuit through some path(s) to the interface output. That is, we use the spurious token to try to produce output safety violation, as opposed to using the disconnection at the saf to produce output progress violation. Sensitization can be useful to detect saf's which the structural theorems in the previous chapter do not cover because of possibility of critical race. It makes use of the idea of partial states and the execution of only a partial segment of a normal pomset behavior. A polynomial time sensitization search algorithm will be given. Given a pomset behavior to be searched, it will be proved that the algorithm is guaranteed to find a sensitization test if it exists.

5.1 Sensitization Problem

Suppose the faulty circuit contains a saf at location s and whose circuit initialization produces a stable state with a spurious token at location b of the circuit (more than one spurious token may exist).

Problem 1: The problem is to derive an input test protocol which uses the spurious token at b such that a token at some output z is generated when run on the faulty circuit and no corresponding token at z is generated when the same test is run on the fault-free circuit.

Problem 2: The problem is to derive an input test protocol which propagates the spurious token at b to some output z in the faulty circuit, where propagation of a token is defined as the enabling of C-elements to allow the token to pass.

Problem 2 only guarantees that an output token at z is produced in the faulty circuit but does not guarantee that a token at z will be absent if the same test is run on the fault-free circuit. This is because the sensitization test may not follow the normal 1-safe behavior of the fault-free circuit. For this case, race simulation is required to check that z is not produced in the fault-free circuit. This chapter will concentrate on Problem 1 which guarantees that the derived sensitization test produces different externally observable behaviors.

Sensitization is useful when no design for testability is available and the strategy of the previous chapter of observing spurious tokens through a large OR gate cannot be used. Such a situation occurs when a circuit is already fabricated and we must use any test strategy available. An at-least-2 test may not cover all nodes when non-CRF components exist. The structural theorems in the previous chapter can be used to cover an initial set of nodes, and then sensitization can be used to cover the remaining ones.

Sensitization can also be used to reduce the number of observation points used. When there is design for testability and observation points are used for observing uncovered nodes after a preliminary test, sensitization can be used to reduce the number of observation points. However, there will be a trade-off of area overhead for observation points and reduction in speed due to capacitance of the observation network vs. the design cost for obtaining the sensitization test and

increase in test length. The simplicity of observation points may make sensitization too expensive when design for testability is available.

Deriving a sensitization test directly from the faulty circuit is too difficult because any arbitrary sequence of inputs sent into the circuit can produce races or token collision. Even with race simulation available, deriving the behavior of a sequential circuit (faulty or not) is already a difficult problem. Instead, the sensitization test can be derived with the aid of the usage protocol which is already given. The strategy is based on extracting a section of the original protocol so that it can be executed on only local regions of the faulty circuit, as opposed to executing the circuit globally using the original protocol. Local execution can avoid the race/anomalous behavior which prevented the saf from being detected. This test does not follow the normal protocol because the protocol section is not executed in the same order as the original protocol, that is, the protocol is not executed starting from initialization.

At circuit initialization, the saf at location s causes a spurious token to be produced which traverses through non-terminals until it stops at location b at the input of a terminal component; otherwise, it would have traversed until it reaches the output interface and be observed by the environment. The spurious token will be at the input of a C-element (we will concentrate on the C-element in sensitization since the other terminal component, the arbiter, is not usually a common component and only behaves as a terminal component under special conditions; we will mention how the results can be extended to the arbiter later). The goal will then be to find a section of the protocol which can send in a token to the other input of the C-element, say input a, and propagate the spurious token forward through some path to some interface output z. If there exists more than one spurious token at the initial stable state, the sensitization of only one of these spurious tokens will be sufficient in order to observe a difference between the

faulty and fault-free circuits.

Assumptions/Restrictions:

1) Input *a* must be independently controllable from input *b*.
2) The saf disconnection at *s* does not appear in the path (region) from *a* to *z*.
3) It is not difficult to predict (in polynomial time) the location of spurious tokens at initialization, given some saf location.

Input *a* must be independently controllable from input *b* since a token must be sent only to input *a* to propagate the spurious token; tokens sent simultaneously to both *a* and *b* can cause the undesired effect of canceling the spurious token at *b*. A case where input *a* is not independently controllable is when there exists a fork followed by the C-element (with other components in between) and in every possible execution, the fork always sends two concurrent tokens to the C-element.

If there are no critical races at initialization then a unique stable state will exist and the location of the spurious token can be easily determined. This can be checked in polynomial time by unfolding the net, either up until a critical race condition is reached (when the unfolding produces more than one behavior), or until the unique stable state is found. [74] proves that checking for reachability of a unique stable state requires polynomial time under the unbounded wire / unbounded gate delay model (although the results may have assumed classical single output gates and not multiple output components). If critical race can occur at initialization, then *k* possible stable states can be reached. A sensitization test must be run on each of these *k* possible states to detect the potential existence of spurious tokens. Since a saf is the source of only a single token, critical races at initialization usually do not occur. This is compared to the *N* iterations of a synchronous test where concurrent inputs are the sources of multiple tokens which must interact with the saf and faulty behavior caused by it; with increasing number of iterations executed, there is increasing probability of critical race and branching

of behavior.

A simplified example of when sensitization can be useful is shown in Fig. 5.1. The circuit C_1 contains an excitory saf which has produced a spurious token at b. C_1 contains a critical race such that an at-least-2 test still produces all required outputs of C_1. The output *z* can be produced twice by tokens going through either *b* or *d*. Since the critical race in C_1 prevents observing any output progress violation, we can instead observe output safety violation by sending an input to *a*. The faulty circuit produces an output *z* while the fault-free circuit does not because of absence of the spurious token.

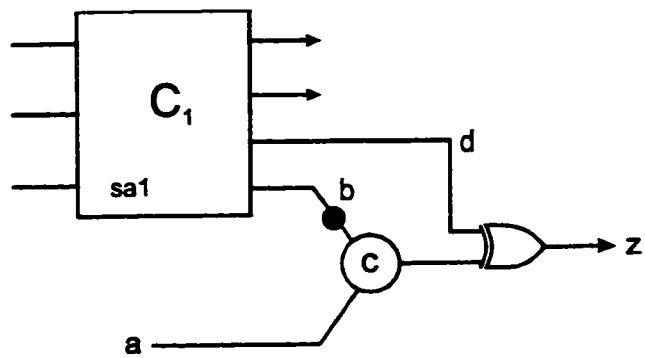


Fig. 5.1: Simplified example of when sensitization is useful.

5.2 Sensitization Test

Let *Nc* be the fault-free circuit net, *Nc'* the faulty circuit net, and *Nt* be an at-least-2 test net for *Nc*. The ***system behavior*** (***system pomset***) is defined to be the unfolded pomset of *Nt•Nc* projected onto circuit input, output, and internal actions. The ***interface behavior*** is the unfolding of *Nt•Nc* projected onto circuit input and output actions. The sensitization test will be extracted from the system behavior *P* as shown in Fig. 5.2, where *P* is the unfolding of *Nt•Nc* and is also an at-least-2 pomset (actually, *P* may be any deterministic behavior of the fault-free circuit; an at-least-2 pomset is used for convenience since it is likely to contain most behaviors

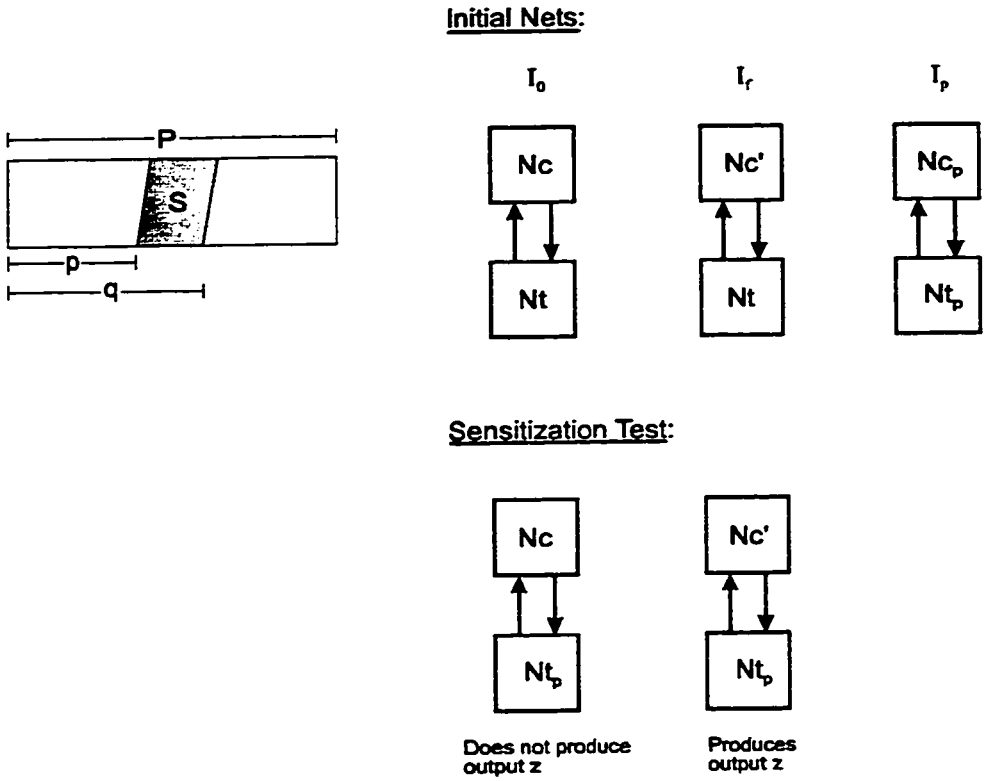


Fig. 5.2: Extracted sensitization test from pomset behavior and corresponding Petri nets used.

of the pomset tree and so gives a large search space for the sensitization test). The shaded region S indicates the segment of pomset which will be used as the sensitization test. p and q are prefixes of P where $p < q \leq P$; p is the start of sensitization test S and q is the end of S ($S = q - p$).

The initial markings of three Petri nets need to be considered in sensitization as shown in Fig. 5.2. Let $I_n = Nt \bullet Nc$ be the normal initial stable net (net with marking after circuit initialization and stable state is reached) and $I_f = Nt \bullet Nc'$ be the faulty initial stable net. The net $Nt_p \bullet Nc_p$ is the fault-free net whose initial marking is obtained from the marking of $Nt \bullet Nc$ after it has been unfolded up to prefix p, ie. $Nt_p \bullet Nc_p$ contains the initial marking of the sensitization test S. The initial net $I_P = Nt_p \bullet Nc_p$.

We want to "execute" the protocol segment S on the faulty and fault-free

circuits such that an output z is produced when S is run on the faulty circuit and no output z is produced when S is run on the fault-free circuit. Since the net marking of Nt_p corresponds to the state of the environment net at the start of S, the execution of S on Nc and Nc' is equivalent to composing Nt_p with Nc and Nc' as shown in the figure and unfolding till the end of S. However, certain conditions must be met before such a composition is valid because token collision/critical race may occur. It is only required that the subnet (subcircuit) that we are interested in for generating the output z be 1-safe, while the remainder of the net can contain hazards, as long as they do not reach the 1-safe subnet. Continuing the unfolding of $Nt_p \bullet Nc_p$ to the end of S is clearly valid since it is a continuation of the specified protocol. But the unfolding of $Nt_p \bullet Nc$ and $Nt_p \bullet Nc'$ does not necessarily unfold into the pomset S because their net markings or "global state" may not be the same. The sensitization test only sensitizes a subnet in Nc and Nc', so we only need the subnet marking or "partial state" to be the same when the test execution starts. The requirement is that the subnet traversed by S on I_p must have an identical partial state/subnet marking as the corresponding subnets in I_n and I_f; the marking of I_n and I_f are very often the same except where it is affected by the spurious token. Or equivalently, the partial states relevant to the sensitization test of I_p, I_n, and I_f must match. These are formalized below.

5.2.1 Partial States

Some necessary pomset operations are first given before defining partial state.

Definition:

Let u be an event in pomset P. The ***minimal prefix*** of u, denoted $\mathbf{m}u$, is the prefix of P containing exactly all the events preceding u. The minimal prefix $\underline{\mathbf{m}}u$ denotes the prefix that contains both $\mathbf{m}u$ and u. The ***maximal prefix*** of u, denoted $\mathbf{M}u$, is the prefix of P that contains all the events of P except u and any of its successors. The maximal prefix $\underline{\mathbf{M}}u$ denotes the prefix that contains both

M*u* and ***u***.

Let P be a pomset, u,v be events in P, and p be some prefix of P. The ***immediate successor*** events of p are the set of events $\{v \mid v \notin p \wedge \forall u\text{: } (u \text{ is immediate predecessor of } v) \rightarrow u \in p\}$. Event u is ***enabled*** (is ***firable***) at p if u is an immediate successor of p. The ***last events*** of p, denoted p°, are the set of events $\{u \mid u,v \in p \wedge \neg\exists\ v \text{ is immediate successor of } u\}$. The ***first events*** of p, denoted $^\circ p$, are the set of events $\{u \mid u,v \in p \wedge \neg\exists\ v \text{ is immediate predecessor of } u\}$. $\mathcal{A}(p)$ denotes the set of actions in p and $\mathcal{E}(p)$ denotes the set of events in p.

Let pomsets $P = [V_P, \Sigma_P, \Gamma_P, \mu_P]$ and $Q = [V_Q, \Sigma_Q, \Gamma_Q, \mu_Q]$. Pomset ***union***, denoted $P \cup Q$, results in the pomset $[V_P \cup V_Q, \Sigma_P \cup \Sigma_Q, \Gamma_P \cup \Gamma_Q, \mu_P \cup \mu_Q]$. Pomset ***subtraction***, denoted $P - Q$, results in the pomset $[V, \Sigma, \Gamma, \mu]$ where $V = V_P - V_Q$, $\Sigma = \mathcal{A}(V)$, $\Gamma = (\Gamma_P^+ \cap (V \times V))^-$, and $\mu = \mu_P - \mu_Q$. Let $v \notin p$: prefix p is said to ***advance*** past v, giving a new prefix $q = p \cup \underline{\mathbf{m}}v$. □

For Fig. 5.3, the last events of p are $\{a, x, y, f\}$, first events of p are $\{a, b, c, f\}$. Events d, e, and z are the immediate successors of p, but not h since not all of its immediate predecessor events are in p. Events d, e and z are enabled at p, but h is not enabled at p. Prefix $\underline{\mathbf{m}}g$ is the minimal prefix of g. Prefix p advances past event g to give prefix q. The last events of q are $\{a, x, g\}$. Event y is not a last event of q

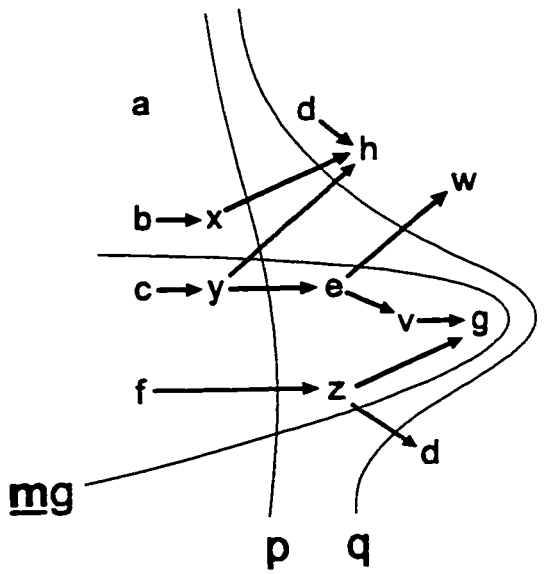


Fig. 5.3: Some operations on a pomset.

since it now precedes g.

Fig. 5.4 illustrates pomset subtraction where pomsets P and Q are indicated by rectangular boxes. All events in Q are deleted from P. When an event is deleted, all its successor events do not have to be deleted. Pomset union $P \cup Q$ results in the entire pomset shown in the figure. P and Q may or may not be disjoint.

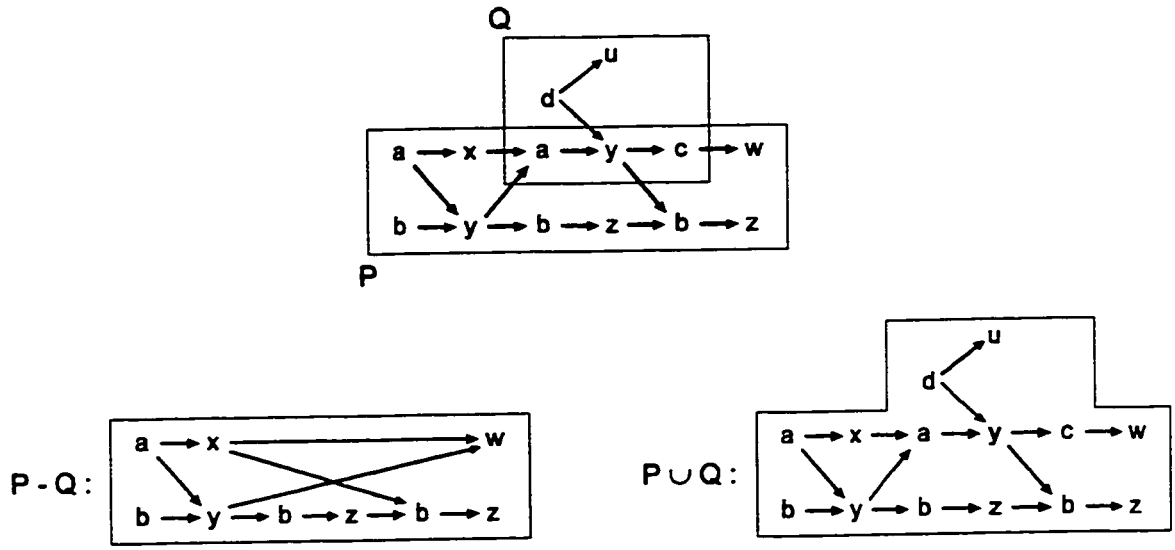


Fig. 5.4: Pomset subtraction and union.

Definition:

Let $\mathcal{U}$ be the universe of actions of a circuit and $\mathcal{B} \subseteq \mathcal{U}$ be a partial set of actions. The *partial state* of prefix p with respect to $\mathcal{B}$ is $\Pi(p,\mathcal{B}) = \{a \mid u \in p^{\circ} \wedge a = \mathcal{A}(u) \wedge a \in \mathcal{B}\}$. The *global state* of p is $\Pi(p,\mathcal{U})$. Let $S_p = \Pi(p,\mathcal{B})$ be the partial state of p, then the *complement* of S_p, denoted $\overline{S}_p$, is $\mathcal{B} - S_p$. S_p represents the places in $\mathcal{B}$ containing tokens in the subnet and $\overline{S}_p$ the places not containing tokens. The *global stable prefix* reachable from p is the prefix q obtained by advancing p past the maximum number of enabled events which are not input events. Let q be the global stable prefix reachable from p. Then the *global stable state* reachable from p is $\Pi(q,\mathcal{U})$, and the *partial stable state* reachable from p with respect to $\mathcal{B}$ is $\Pi(q,\mathcal{B})$. □

Fig. 5.5 illustrates the stable state of a prefix where $\{a, b, c\}$ are input actions,

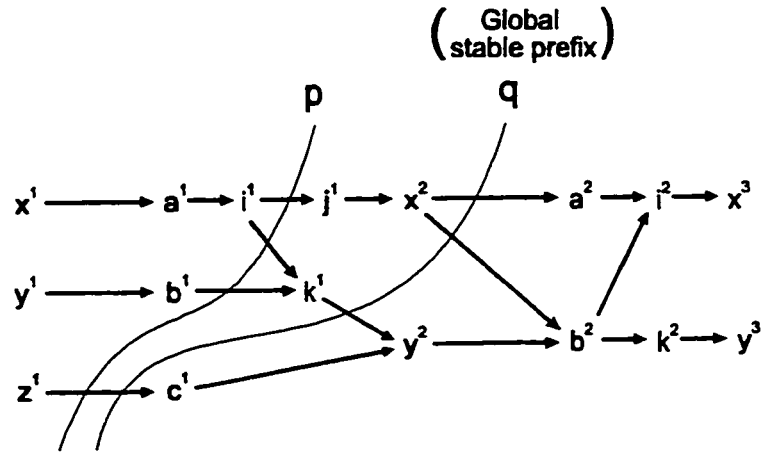


Fig. 5.5: Stable state of a prefix p.

$\{i, j, k\}$ are internal actions, and $\{x, y, z\}$ are output actions. The global state of prefix p is $\Pi(p,\mathcal{U}) = \{i^1, b^1, z^1\}$. The global stable prefix reachable from p is the prefix q after advancing past the enabled events $\{j^1, k^1, x^2\}$ up to input events. The global state of q is then $\Pi(q,\mathcal{U}) = \{x^2, k^1, z^1\}$. This is the same as finding the stable state of a circuit where initially there is an internal signal transition at node i and an input has been sent to b, and we wait for the circuit to stabilize. Suppose we are interested in a subcircuit /subnet containing only the actions $\mathcal{B} = \{a, i, j, x\}$. Then the partial state of p with respect to $\mathcal{B}$ is $\Pi(p,\mathcal{B}) = \{i^1\}$. The partial stable state reachable from p will then be $\Pi(q,\mathcal{B}) = \{x^2\}$ when p has advanced past the output x. We would not be interested in whether event k^1 has occurred in this case because it is not in the subnet; this would be allowed if k^1 does not lead to events which affect the subnet.

The global and partial stable state of a given prefix must be unique when the fault-free protocol is followed because the behavior is a single deterministic pomset and free of critical races. The global stable state may not be unique if the protocol is run on the faulty circuit which has a critical race. However, the partial stable state can still be unique if it can be guaranteed that the subnet exactly follows the normal fault-free behavior. For example, consider the circuit of Fig. 5.1. Even though there may be a critical race in subcircuit C_1 which can cause a non-unique

global stable state, the circuit outside C_1 can still follow the normal behavior and have unique partial states. In this case, we are "imagining" that there were no faults in C_1 in order to obtain a unique global stable state so that we can obtain the unique partial stable state outside C_1.

5.2.2 A Subtle Difference Between Labeling Actions on Places vs. Actions on Transitions

This thesis labels actions on Petri net places, but it is also possible to label the Petri net transitions. In most cases, the results are equivalent, however, there exists a subtle difference between the two models. Both nets will unfold into the same pomset behavior, but there is a difference in relating a net marking with a pomset prefix. Fig. 5.6 shows more precisely what this difference is. In a place labeled net, the last events of a prefix correspond to the net marking. In a transition labeled net, the cut of a prefix on an arc corresponds to the net marking. Every net marking must correspond to a unique prefix, however, the reverse may not be true in the place labeled net model. There can exist a prefix in which there does not exist a corresponding net marking. As shown in the figure, this occurs when the prefix cuts a fork where only one of two events has occurred. This non-existence is a result of the fork net transition producing two output tokens simultaneously when it fires; it cannot delay the production of one of its outputs. The delay of one output in a DI fork is implemented by composing a wire component with the fork output.

Definition:

Let P be a pomset, u,v,w be events in P, and p be a prefix of P. p is a ***reachable prefix*** if and only if $\forall u,v \in p\ \forall w \notin p$: $\neg\exists$(v,w are immediate successors of u). Otherwise, p is an ***unreachable prefix***. The ***reachable prefix operation***, $q = p^{\circ}$, gives a new prefix q where q is p advanced past the set of events $\{w \mid u,v \in p \wedge w \notin p \wedge v,w$ are immediate successors of $u\}$. □

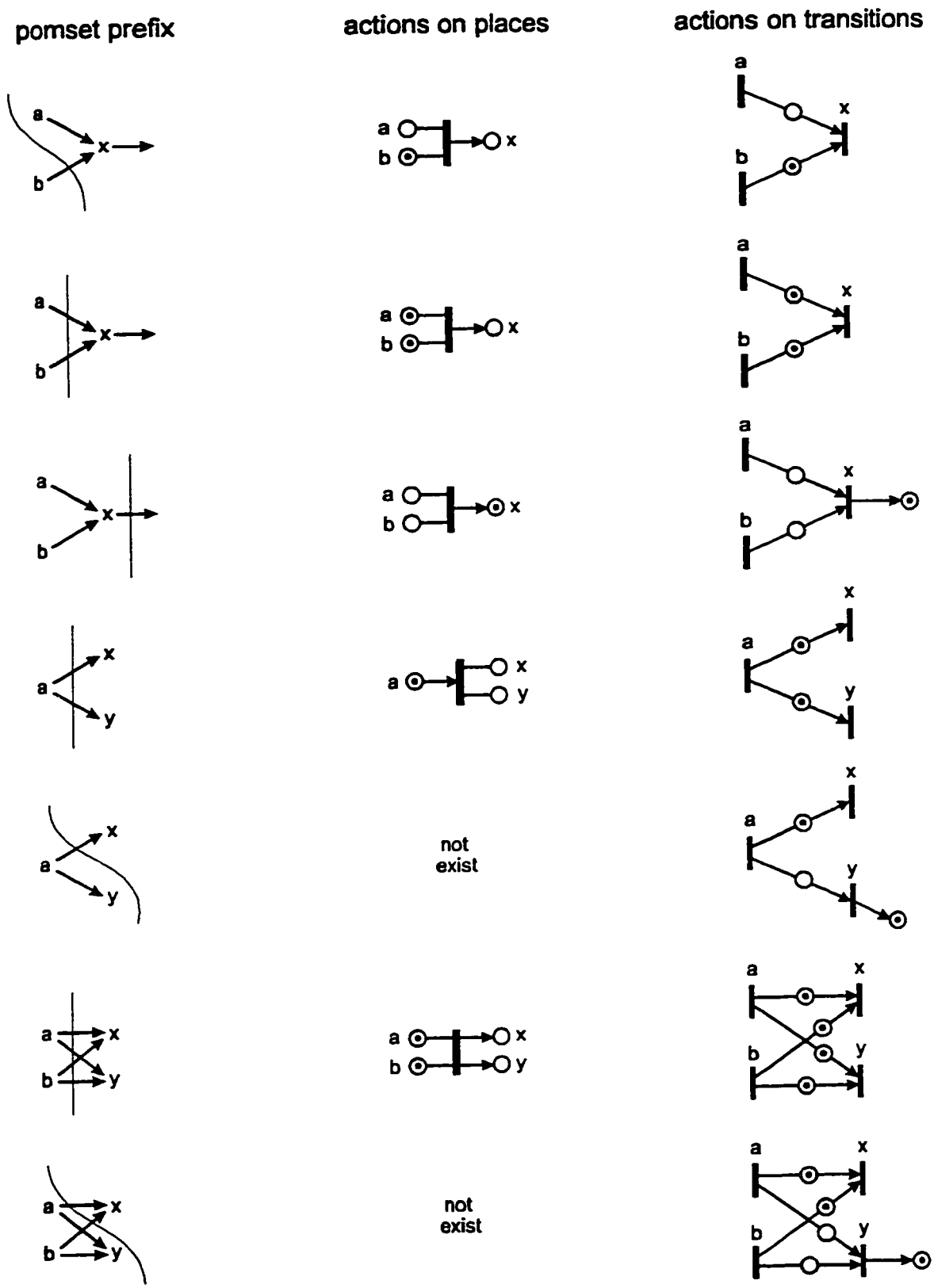


Fig. 5.6: Token markings of place labeled and transition labeled Petri nets corresponding to a pomset prefix.

The result is that some prefixes are not reachable in the pomset behavior of a place labeled net. Fig. 5.7 illustrates some reachable prefixes (solid lines) and an unreachable prefix (dashed line). A reachable prefix requires that when a prefix contains one output event of a fork, it must contain all output events of that fork. This simply means that unreachable prefix p must be advanced past these single events to reachable prefix q. Every net marking must have a corresponding prefix, and the operation $q = p^{\varrho}$ will guarantee that the last events of prefix q will directly correspond to a net marking. In the remainder of this thesis, we assume the use of reachable prefixes if it is not explicitly indicated.

(Eliminating some prefixes may be somewhat cumbersome in the place labeled net model, but we retain this model because the subtle difference was only discovered at a later stage of thesis preparation. Using the transition labeled net model may be just as cumbersome when each high-level transition is modeled as a set of places.)

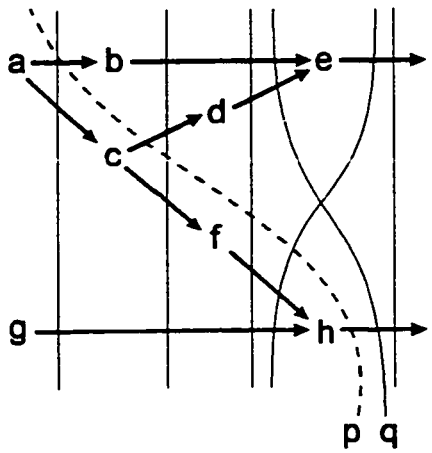


Fig. 5.7: Example reachable and unreachable prefixes corresponding to a place labeled Petri net.

5.2.3 Sensitization Region

We now define the sensitization region, the events in the pomset which will necessarily be executed in the test to produce the output z.

Definition:

Let P be a pomset and p be a prefix of P. The ***sensitization region*** of p with respect to output event z where $z \in \mathcal{E}(P-p)$, denoted $\mathcal{SR}(p,z)$, is a pomset consisting of the regions $R_1 \cup R_2 \cup R_3$ where

R_1: $\mathbf{m}z - p$.

R_2: $p^\circ \cap \mathbf{m}z$.

R_3: Let event $u \in (R_1 \cup R_2)$, event $e_1 \notin (R_1 \cup R_2)$, and e_1 be an immediate successor of u. For all linear chains of events $e_1, e_2, \ldots e_k, x$ where $e_{i=1..k}$ are non-C-element internal events, and x is an output event or a C-element input event: (i) if x is an output event, then $e_1, \ldots, e_k \in R_3$, (ii) if x is a C-element input event and x' is the other input of that C-element, then $e_1, \ldots, e_k, x' \in R_3$.

The partial state at $\mathcal{SR}(p,z)$ is $\Pi(p,\mathcal{B})$ where $\mathcal{B}=\mathcal{A}(R_1 \cup R_2 \cup R_3)$. □

Let pomset P be the unfolding of a net N, p be a prefix of P, and $S_p = \Pi(p,\mathcal{B})$ be the partial state of p. $\mathcal{B}$ is a set of actions which correspond to a subnet of N. S_p represents the current token marking of that subnet. S_p is a subset of p° since p° corresponds to the global net marking, ie. events of p° enable future events, corresponding to token places enabling net transitions.

To illustrate the use of partial state in sensitization, Fig. 5.8 shows the sensitization region of prefix p which represents the starting state of a sensitization test. Partial state rather than global state is used since only a local region of the circuit is required to be sensitized. a^i, b^i, and c^i correspond to some event occurrence of a C-element within the system pomset. Event b^i must be a last event of p since b contains the spurious token. The sensitization test proceeds by advancing p past a^i, thereby enabling the spurious token at b to pass through the C-

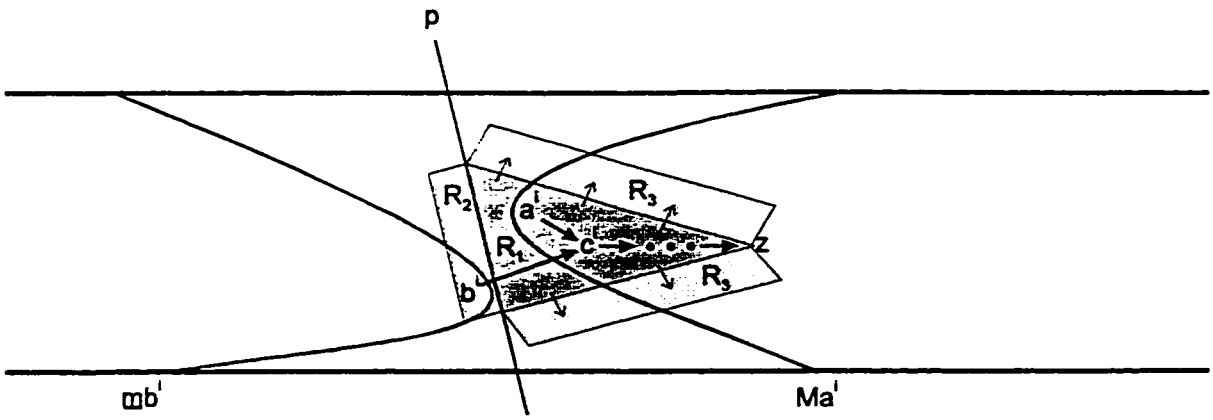


Fig. 5.8: Sensitization region $\mathcal{SR}$(p,z).

element. Prefix p advances until it generates an output z to the environment. Execution of the same pomset segment on the normal net, without the spurious token at b, will not fire the C-element and hence will not produce an output z. Prefix p may be valid within the range $\underline{\mathbf{m}}b^i$ to $\mathbf{M}a^i$: after b^i has fired, but not a^i. R_1 is the pomset segment which must be executed in order to fire output z; these are the minimum events which must be executed, so its region is $\mathbf{m}z - p$. We are only interested in the region R_2 which enables R_1 rather than all the last events of p; p only needs to advance from R_2 to z under external control. Although only R_1 and R_2 are necessary to enable z, R_3 must also be included in the sensitization region and represents the settling of the tokens from R_1 to a stable state. For example, the token state of R_3 may be different in the faulty and fault-free circuits such that the traversal of a token in R_3 may cause a critical race or token cancellation which prevents z from being produced in the faulty circuit or causes z to be produced in the fault-free circuit. Including R_3 in the sensitization region expands the subnet of $\mathcal{B}$ actions under consideration to include all places which are affected by the sensitization of R_1 so that the firing of events in R_3 due to the stabilization of internal circuit transitions do not cause any hazards for the sensitization of R_1.

The initial marking of the faulty subnet in Nc' and the fault-free subnet in Nc_p are identical. The structure of both nets are identical as well, since the saf

disconnect is required to not be in the sensitization region. Since both subnets are identical, the faulty subnet will have the same 1-safe behavior as in the original fault-free pomset segment. The structure of the subnet in $Nt \bullet Nc$ will also be identical, but since its behavior starts at the beginning of the at-least-2 pomset rather than the middle of the pomset as in $Nt_p \bullet Nc_p$, it will be missing the spurious token at place b and will not be able to produce the output z. The difference between $Nt \bullet Nc$ and $Nt_p \bullet Nc_p$ is that $Nt_p \bullet Nc_p$ has been partially executed which has placed a token at b while $Nt \bullet Nc$ has not yet been executed and does not contain a token at b.

The global nets of Nc and Nc' are in stable states at initialization, so the actual initial prefix of the sensitization test will always be a global stable prefix. This is important because any partial state of I_p which matches with I_n and I_f is guaranteed to result in a global stable state when the sensitization test starts. It does not matter whether the net marking of I_p is a global stable state or not. Moreover, any tokens in I_p which are not stable cannot adversely affect the sensitization region because these tokens will not exist in the actual net under test; only the tokens in the subnet corresponding to the sensitization region will exist, plus the stable tokens exterior to the subnet in I_n / I_f.

Fig. 5.9 and 5.10 show a more detailed example. $\{d,e,f,g,h\}$ are inputs, $\{i_1,i_2,i_3,i_4,i_5,i_6,i_7\}$ are internal nodes, and $\{x_1,x_2,z\}$ are outputs. R_1 contains events $\{a,c,e,g,i_1,i_3,i_4\}$, R_2 contains events $\{b,d\}$, and R_3 contains events $\{i_2,i_5,i_6\}$. $\mathcal{B}=\{b,d,a,c,e,g,i_1,i_2,i_3,i_4,i_5,i_6\}$, $S_p=\Pi(p,\mathcal{B})=\{b,d\}$, and $\overline{S}_p=\{a,c,e,g,i_1,i_2,i_3,i_4,i_5,i_6\}$. For the sensitization test to be valid, the token marking at the places of $\mathcal{B}$ in I_n and I_f should be identical, except at place b where $b \in S_p$ in I_f and $b \in \overline{S}_p$ in I_n. The actions of R_1 are elements of $\overline{S}_p$ in this example since a token in any place in R_1 will cause token cancellation when prefix p is advanced, which would prevent z from being produced. In R_2, only the last events $\{b,d\}$ of p which are input or internal actions

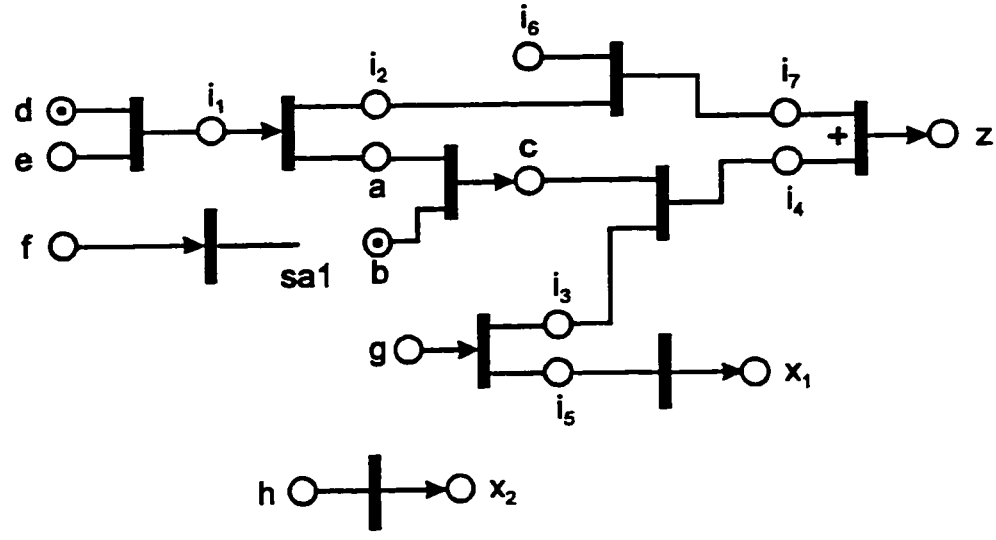


Fig. 5.9: Petri net of a faulty circuit to be sensitized.

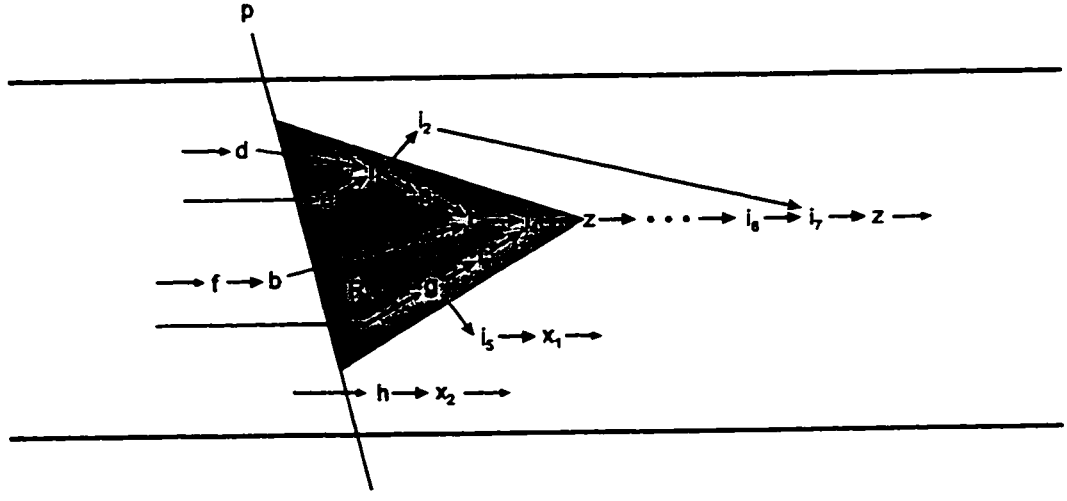


Fig. 5.10: Pomset behavior of Fig. 5.9.

need to be included. Since p is a stable state, any last event of p which is an output action corresponds to a token sent to the environment net Nt; the output places of the circuit at stable state must automatically contain no tokens because the environment will immediately consume them. Only the token marking in the subnets Nc, Nc', and Nc_p need to match, except at place b. The environment net may be different since $Nt \bullet Nc$ of I_n and $Nt \bullet Nc'$ of I_f will be replaced by the new sensitization net Nt_p, forming $Nt_p \bullet Nc$ and $Nt_p \bullet Nc'$. Actions of R_3 such as i_6 must be included in the subnet of $\mathcal{B}$ actions, otherwise a token at i_6 at a C-element input

may prematurely produce a z when i_2 fires.

The sensitization test projected onto interface ports will be $\{e,g\}$; z. At stable state, many places will be the same (ie. empty) in I_n and I_f since tokens in the circuit can only exist at the input of terminal components or inside the state holding components.

If arbiters (or any other terminal component) do exist in the circuit, it is possible to extend the results by treating the arbiter similar to the C-element. If at initial stable state, a spurious token exists at one input of the arbiter when it behaves like a terminal component, then sensitization would mean sending an input to the alternate input of the arbiter to release it and allow the spurious token to propagate through the circuit to an interface output.

It is possible to improve the results by reducing the size of R_3 which would reduce the size of the partial state which must match and increase the possibility of finding a sensitization test. Currently, R_3 includes all chains of events emanating from the $R_1 \cup R_2$ region, which results in only transitions in the subnet of $R_1 \cup R_2 \cup R_3$ actions taking place and not outside the subnet. Only some of these R_3 chains might affect the production of the output z, while the remaining chains are not required. These unimportant chains can be moved out of R_3 and the subnet, thereby allowing transitions to occur outside the subnet and even hazardous token collisions to take place. To determine when these chains are unimportant, an analysis of the Petri net can be performed (analysis of the pomset would not be sufficient because it does not contain all possible behaviors such as token collisions). There are two possibilities: (i) structurally, by checking whether the chain of places can return back into the subnet, or (ii) behaviorally, which is equivalent to race simulation to determine that all behaviors do not affect the production or required absence of output z. We will not explore this further.

5.3 Correctness Proof of Pomset Segment Execution Using Partial State Matching

Some terms are first defined.

Definition:

Let $P = [V_P,\Sigma_P,\Gamma_P,\mu_P]$, $Q = [V_Q,\Sigma_Q,\Gamma_Q,\mu_Q]$, $R = [V_R,\Sigma_R,\Gamma_R,\mu_R]$ be pomsets and u,v be events. Pomsets P and Q are *disjoint* if $(u \in V_P \rightarrow u \notin V_Q)$. Let P and Q be disjoint pomsets. *Pomset event concatenation* of P and Q produces a new pomset R, denoted $R = P;;Q$ where $R = [V_P \cup V_Q, \Sigma_P \cup \Sigma_Q, \Gamma_P \cup \Gamma_Q \cup \Gamma', \mu_P \cup \mu_Q]$ where Γ' is a subset of arcs from $\{(u,v) \mid u \in P \wedge v \in {}^\circ Q\}$ such that $\forall v \in {}^\circ Q\ \exists u \in P$: $(u,v) \in \Gamma'$. □

Pomset event concatenation $P;;Q$ results in every event in Q being a successor of *some* event in p, and can be accomplished without restriction on the arcs from P to Q. It will be used to specify the pomset segments and the partial order between them. It is different from strictly ordered concatenation $P;Q$ in which every event in Q is a successor of *all* events in P.

Definition:

The *projection* of net $N = \langle P, T, F, M_o \rangle$ onto a set of places/actions A ($A \subseteq P$), denoted $N\downarrow_A$, produces a new net $N' = \langle P', T', F', M_o' \rangle$ where $P' = A$, $T' = \{t \mid (p,t) \in F \wedge p \in A\}$, $F' = \{(p,t) \mid (p,t) \in F \wedge t \in T'\} \cup \{(t,p) \mid (t,p) \in F \wedge t \in T'\}$, and M_o' = M_o restricted to places of A. □

Fig. 5.11 illustrates the projection of a net N onto a set of actions $A=\{a,b,d,f,g\}$. Note that the behavior of a net projected onto a set of actions may not necessarily be the same as the behavior obtained by unfolding the original net and then projecting onto the set of actions. We use net projection only in the following proof as a peephole to a subcircuit in which the only transitions that occur are within that projected subcircuit.

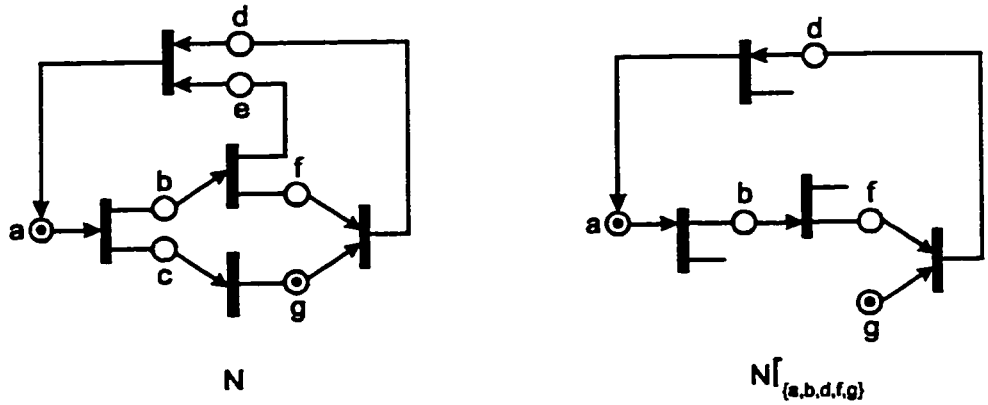


Fig. 5.11: Example of net projection.

The following theorem asserts that execution of a pomset segment is a 1-safe execution under the condition of matching partial states.

Theorem 5.1:

Let $p;;q;;r$ be the unfolded system behavior of a 1-safe net $N=Nt \bullet Nc$ where p,q,r are disjoint pomsets. Let $Nt_p \bullet Nc_p$ be the net after unfolding N up to p and $Nt_q \bullet Nc_q$ be the net after unfolding N up to $p;;q$. If $\Pi(p,\mathcal{B}) = \Pi(p;;q,\mathcal{B})$ where $\mathcal{B}=\mathcal{A}(r)$, then $Nt_q \bullet Nc_p$ (execution of $p;;r$ on Nc) is 1-safe and unfolds into r. □

Proof:

Let $N'=N$. Unfold N up to $p;;q$, giving net $Nt_q \bullet Nc_q$. Unfold N' up to p, giving $Nt_p \bullet Nc_p$. If we continue unfolding of $Nt_q \bullet Nc_q$ up to $p;;q;;r$, only transitions in subnet $(Nt_q \bullet Nc_q)\lceil_{\mathcal{B}}$ may fire. Since $\Pi(p,\mathcal{B}) = \Pi(p;;q,\mathcal{B})$, then $Nc_p\lceil_{\mathcal{B}} = Nc_q\lceil_{\mathcal{B}}$. Since $Nt_q \bullet Nc_q$ is 1-safe and will unfold into r, and only transitions $Nc_q\lceil_{\mathcal{B}}$ in Nc_q may fire when Nt_q is used, and $Nc_p\lceil_{\mathcal{B}} = Nc_q\lceil_{\mathcal{B}}$, then $Nt_q \bullet Nc_p$ is also 1-safe when unfolding into r. □

The theorem says that given a normal 1-safe behavior $p;;q;;r$, we can derive a new 1-safe behavior $p;;r$ by skipping the pomset segment q. The execution of r is restricted to the subnet involving only the actions of $\mathcal{B}$. Since the partial states at p and $p;;q$ are the same, the marking of the two subnets are the same, and the execution of r for both nets will involve identical transition firings within the subnet. No transitions outside the subnet can fire when executing r. If the behavior

were to continue, equivalent to enlarging r, then the size of the subnet would be enlarged to cover any new actions which are newly covered by r.

In testing, $p;;q;;r$ will be the at-least-2 normal pomset behavior, p will be the prefix at initialization, ie. p consists of only the set of events corresponding to the initial marking of the faulty/fault-free nets, and r will be the sensitization test. q is the pomset segment of the normal behavior which is skipped when we jump from the initial state directly to the sensitization test r. $\Pi(p;;q,\mathcal{B})$ will be a stable state in the test because the start of the sensitization test will be at circuit initialization at stable state, ie. p will be a global stable prefix and execution of the sensitization test of $p;r$ must be 1-safe, which is covered by $p;;r$ being 1-safe in the theorem.

The theorem applies to pure pomsets/Petri nets and makes no requirements on the pomset segments to be stable states. The notion of sensitization region and stable state is an implementation requirement of the circuit due to the fact that the environment cannot stop internal transitions. The use of sensitization region with stable state only increases the size of the above pomset segment r.

5.4 Sensitization Search Algorithm

We want to search the entire system pomset for a prefix which satisfies the sufficient conditions of a correct test. More than one such prefix may exist or none at all. Let a, b be the inputs and c be the output of a C-element. Let s be the saf location and a spurious token exists at b in the initial stable state of the faulty circuit. Let system pomset P be the unfolding of I_n, and prefix p be the start of the sensitization test to be searched for. Let P_n and P_f be the initial prefixes of I_n and I_f, ie. P_n and P_f contain events corresponding to the initial markings of I_n and I_f respectively. A correct test requires that $\underline{\mathbf{m}}b^i \leq p \leq \mathbf{M}a^i$ and $\Pi(p,\mathcal{B}) = \Pi(P_n,\mathcal{B}) = \Pi(P_f,\mathcal{B})$ where $\mathcal{B} = \mathcal{A}(\mathcal{SR}(p,z))$ for some interface output event z which is a successor of c_i for some event occurrence i.

Algorithm 5.1: (Sensitization Prefix Search Algorithm)

Input: P, P_n, P_f, s.

Output: p, z.

$i = 1$; $p = P_n$;

while $p \leq P$ **do**

Find events a^i, b^i, c^i in P;

foreach output event z which is a successor of c^i **do**

{Search for prefix p, where $\underline{\mathbf{m}}b^i \leq p \leq \mathbf{M}a^i$}

$p = (\underline{\mathbf{m}}b^i)^{\circ}$; $\mathcal{B} = \mathcal{A}(\mathcal{SR}(p,z))$; $S_p = \Pi(p,\mathcal{B})$, $S_n = \Pi(P_n,\mathcal{B})$, $S_f = \Pi(P_f,\mathcal{B})$;

{Advance p past initial set of events whose actions do not match in S_n and S_f}

foreach event u in $\mathbf{M}a^i - \underline{\mathbf{m}}b^i$ **do**

if $\mathcal{A}(u) \in (S_n \cap \overline{S}_f) \cup (\overline{S}_n \cap S_f)$ **then**

Let $\{e_1,...,e_{k2}\}$ be all the immediate successors of u. Advance p past $\{e_1,...,e_{k2}\}$, ie. $p = (p \cup \underline{\mathbf{m}}e_1 \cup ... \cup \underline{\mathbf{m}}e_{k2})^{\circ}$;

endif

endfor

{Main loop to advance p}

Loop

$\mathcal{B} = \mathcal{A}(\mathcal{SR}(p,z))$; $S_p = \Pi(p,\mathcal{B})$, $S_n = \Pi(P_n,\mathcal{B})$, $S_f = \Pi(P_f,\mathcal{B})$;

if $(S_p = S_n = S_f) \wedge s \notin \mathcal{B}$ **then**

Return p,z and exit **Algorithm**; {sensitization test found}

else

if $p \geq \mathbf{M}a^i$ **then** exit **Loop** **endif**;

foreach $a \in (\mathcal{B} - \{b\})$ **do**

if $a \in S_p \wedge a \in \overline{S}_n \wedge a \in \overline{S}_n$ **then** {S_p has extraneous token}

Let $\{e_1,...,e_{k2}\}$ be all the immediate successors of event $u \in p^{\circ}$ where $\mathcal{A}(u) = a$.

Advance p past $\{e_1,...,e_{k2}\}$, ie. $p = (p \cup \underline{\mathbf{m}}e_1 \cup ... \cup \underline{\mathbf{m}}e_{k2})^{\circ}$;

endif

if $a \in \overline{S}_p \wedge a \in S_n \wedge a \in S_f$ **then** {S_p missing token}

Advance p past event u, where u is the next occurrence of a after p and $\mathcal{A}(u)=a$, ie. $p = (p \cup \underline{\mathbf{m}}u)^{\circ}$;

endif

endfor

endif

endLoop

endfor

$i = i + 1$;

endwhile

Report sensitization test not found.

The algorithm proceeds by checking all the C-element occurrences in the system behavior P which potentially contains a pomset segment which can sensitize the spurious token. For each C-element occurrence, the spurious token needs to be propagated to only one of the interface outputs z. $S_p=S_n=S_f$ is the *detection condition* of Algorithm 5.1 for some prefix p and set of actions $\mathcal{B}$; $p \geq \mathbf{M}a^i$ is the *termination condition*. According to Theorem 5.1, p must satisfy the detection condition. The existence and non-existence of tokens in the places/actions corresponding to S_n and S_f remains the same throughout the algorithm since its global state is obtained from the initial prefix of P_n and P_f. As p advances towards z, the partial state action set $\mathcal{B}$ decreases in size since the sensitization region from p to z decreases in size. S_n and S_f are matched by reducing the size of $\mathcal{B}$ through the advancement of p. Initially, p is advanced past the initial set of events whose actions do not match in S_n and S_f. Since the global marking corresponding to S_n and S_f do not change, only the removal of these events from $\mathcal{B}$ can make S_n, and S_f match. Prefix p must be advanced past the immediate successors of u in the initial advancement, otherwise $\mathcal{A}(u)$ will still be in $\mathcal{B}$ in region R_2.

Loop is the main loop which performs the advancement of p. There are two conditions which are checked where p must be advanced: (i) S_p contains an

extraneous token, and (ii) S_p is missing a token. In the first case, when the event u of action a to be matched contains a token in S_p but not in S_n and S_f, the advancement of p past the immediate successors of u will remove that token since u is no longer in p°. In the second case, when the event u of action a to be matched does not contain a token in S_p but does in S_n and S_f, the advancement of p past event u which is the next occurrence of a will make u be in p° and so place a will contain a token. Each advancement is the minimum that is required to make a match. The action b in $\mathcal{B}$ is not included in the matching because b contains a spurious token in the faulty circuit but does not contain a token in the fault-free circuit.

The advancement of p can be interpreted as a sequence of token placements and removals to change the marking of the subnet corresponding to S_p to match with the fixed token marking of the subnet corresponding to S_n and S_f. The token marking of subnets corresponding to S_n and S_f are fixed after the initial advancement to make S_n and S_f match. The algorithm does not need to consider advancements to stable states because p is being matched with P_n and P_f which are stable states at circuit initialization. Every event which will be executed in the test is included in the sensitization region, and the test will terminate in a stable state since region R_3 includes execution to stable state. After each advancement of p to some minimum prefix, the reachable prefix operation $p^{\mathbf{Q}}$ is performed. The alternative of performing the reachable prefix operation only at the end of the algorithm on the final detected p instead of after each advancement would not be correct because advancing past these single fork events would add tokens to S_p and cause a mismatch; to eliminate these tokens, the advancement of p would need to be continued.

Fig. 5.12 and Table 5.1 illustrates the execution of the Sensitization Prefix Search Algorithm. Suppose the algorithm did not find a sensitization prefix for

a^1,b^1,c^1 and has advanced the prefix to a^2,b^2,c^2. Prefix p_1 is initially set to $\underline{m}b^2$, and its corresponding reachable prefix is p_1^{ϱ}. The corresponding token markings for the

			p_1^{ϱ}	p_2^{ϱ}	p_3^{ϱ}	p_4^{ϱ}
action	S_n	S_f	S_p	S_p	S_p	S_p
a	○	○	○	○	○	○
b	○	●	●	●	●	●
c	○	○	○	○	○	○
d	●	○	○	⇒ ×	×	×
e	●	●	○	○	●	●
f	●	●	○	○	○	⇒ ●
g	○	○	●	●	⇒ ○	○
h	○	○	○	○	●	○
k	○	○	●	●	⇒ ○	○
m	○	○	●	●	⇒ ○	○
n	○	○	○	○	●	○

Table 5.1: States of S_n, S_f, S_p at prefixes p_1, p_2, p_3, p_4.

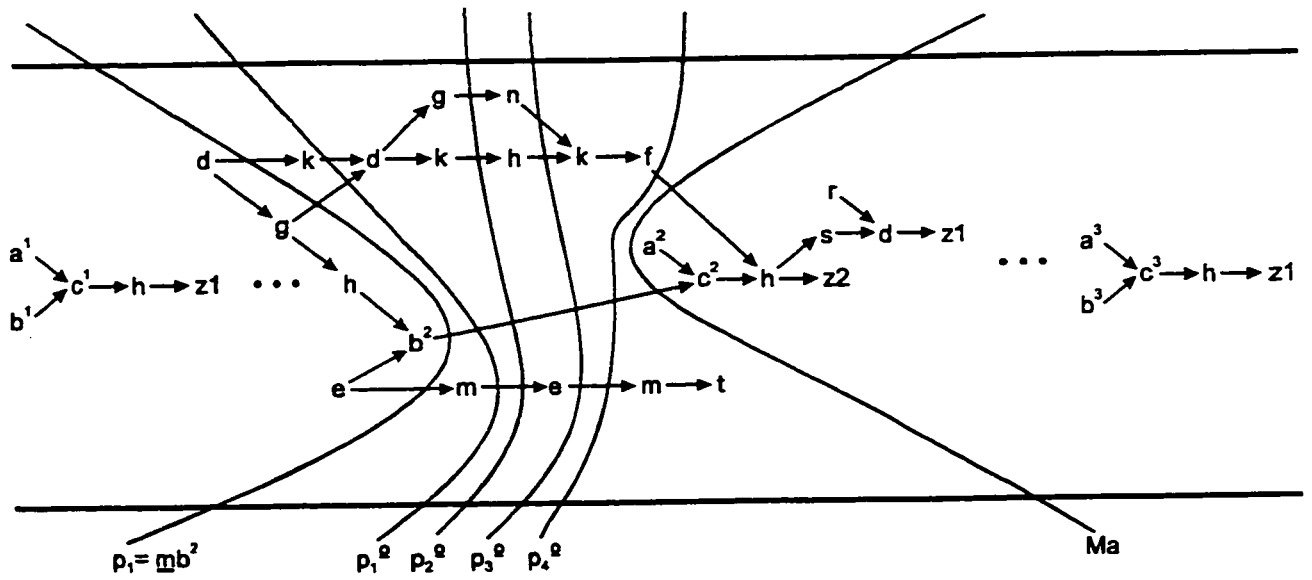


Fig. 5.12: Example execution of Sensitization Prefix Search Algorithm.

states S_n, S_f, and S_p are shown in the table where the black dots indicate the presence of a token and the white dots indicate the absence. Initially, p must be advanced past the initial set of events whose actions do not match in S_n and S_f; this is indicated by prefix p_2°. The events which must be advanced past is $\{d\}$ and its immediate successors $\{g,k\}$. Action d is then no longer in the partial state set $\mathcal{B}$, and this is indicated by an arrow in the table giving an X which is a don't care. S_n and S_f now match exactly, except for spurious token b. Prefix p_3° is the advancement of p when the first **if** statement in the last **for** loop of the algorithm is executed. Since a token exists in $\{g,k,m\}$ of S_p and does not appear in S_n and S_f, the prefix must advance past all the immediate successors of $\{g,k,m\}$ to eliminate these tokens. Prefix p_4° is the advancement of p when the second **if** statement in that **for** loop is executed. Since a token does not exist in $\{f\}$ of S_p and must appear in S_n and S_f, the prefix must advance to the next occurrence of $\{f\}$, making $\{f\}$ a last event of p and placing a token in this place. If output $z1$ was selected, then no matching partial state exists since the partial state must contain $\{d\}$ which is in $\underline{mz1}$, but $\{d\}$ cannot match in S_n and S_f. Selection of output $z2$ can give a matching partial state because $\underline{mz2}$ does not contain $\{d\}$. The partial states of S_n, S_f, and S_p now matches, except for b, where the partial state set for the sensitization region $S\mathcal{R}(p_4^{\circ},z2)$ is $\mathcal{B}=\{a,b,c,f,h,m,r,s,z2\}$ where m,r,s is in region R_3 containing no tokens which also matches with S_n and S_f, and t is an interface output.

Definition:

Event g^i is ***removed*** by the algorithm in step k when the token at place $\mathcal{A}(g^i)=g$ is consumed. Equivalently, this means that g^i is ***removed*** from p°. p is said to be a ***legal state*** if p satisfies the detection condition. □

Consider the advancement of p in the pomset segment after the initial advancement where the two possible conditions of $(g\in S_n \wedge g\in S_f)$ or $(g\in \overline{S}_n \wedge g\in \overline{S}_n)$ are true and p must still be advanced.

Theorem:

Any event g^i, such that $g=\mathcal{A}(g^i)$ and $(g\in S_n \wedge g\in S_f)$ or $(g\in \overline{S}_n \wedge g\in \overline{S}_f)$, cannot form a legal state in P if the algorithm removes it.

Proof:

Basis: The first event removed, g^1, must satisfy one of the following situations according to the algorithm:

(a) $g\in \overline{S}_n \wedge g\in \overline{S}_f$: obviously g^1 cannot form a legal state required and must be removed.

(b) $g\in S_n \wedge g\in S_f$: according to the algorithm, g^1 can be removed because of

(i) some event h^1 which is needed to form the legal state is missing in the initial prefix and in order to include h^1, g^1 must be consumed and thus removed: in this case, there is no legal state that can contain g^1 and some h.

(ii) some x^1 which should not be there to form a legal state is in the initial state and in order to remove x^1, g^1 must also be removed (such as in case of a C-element): thus, there cannot be a legal state which includes g^1 but not x^1.

(iii) g^1 removed because of reachable prefix operation: an unreachable prefix obviously cannot form a legal state.

Thus if g^1 is removed, there is no legal state that can contain it.

Induction hypothesis: Any event g^1, ..., g^k removed by the algorithm cannot be used in forming a legal state of P.

Induction step: Consider the removal of event g^{k+1}.

(a) $g\in \overline{S}_n \wedge g\in \overline{S}_f$: obviously g^{k+1} cannot form a legal state and must be removed.

(b) $g\in S_n \wedge g\in S_f$: g^{k+1} is removed because

(i) some event h^j is needed to form the legal state but g^{k+1} is needed to produce h^j. By the induction hypothesis, g^{k+1} cannot form a legal state with any of the events already removed (which includes h^{j-1} since h^j is

the next occurrence of h after p). If g^{k+1} is to form a legal state, the legal state must include some $h^{j'}$ ($j' \geq j$). But this cannot be a valid prefix since $h^{j'} \in p^{\circ}$ implies $g^{k+1} \notin p^{\circ}$ since g^{k+1} precedes $h^{j'}$.

(ii) some event x^j which should not be in the current state which includes g^{k+1}, but the removal of x^j requires the removal of g^{k+1} also. Since g^{k+1} cannot form any legal state with any event already removed by induction hypothesis, we must remove x^j to form some legal state with g^{k+1}. But this cannot happen without removing g^{k+1} as well.

(iii) g^{k+1} removed because of reachable prefix operation: g^k and g^{k+1} are the fork outputs where g^k has already been removed. If g^k cannot form a legal state by induction hypothesis, then g^{k+1} cannot form a legal state as well, otherwise the prefix will be unreachable.

Thus any event removed in the algorithm cannot form a legal state in P. □

The Algorithm 5.1 therefore is guaranteed to detect the desired p if it exists since p is advanced past any event which cannot be a last event of p. Once advancement terminates, the desired p is found if it is in $\underline{\mathbf{m}}b^i - \mathbf{M}a^i$. The existence of some p in the entire at-least-2 behavior will be dependent upon the given circuit and the restrictions outlined previously; the disconnect must not be part of the sensitization region, and both the inputs of the C-element must be independently controllable, which implies that a segment from the normal behavior exists which corresponds to a traversal of the circuit without traversing through the disconnect and one of the C-element inputs.

The following theorem shows that only checking the nearest occurrence of an output z from the C-element is sufficient.

Theorem:

If there does not exist a prefix p where $\underline{\mathbf{m}}b \leq p \leq \mathbf{M}a$ and $S_p = S_n = S_f$ where $S_p = \Pi(p, \mathcal{B})$, $S_n = \Pi(P_n, \mathcal{B})$, $S_f = \Pi(P_f, \mathcal{B})$, and $\mathcal{B} = \mathcal{A}(\mathcal{SR}(p, z^i))$, then p does not exist for any $\mathcal{B}' = \mathcal{A}(\mathcal{SR}(p, z^{i+k}))$ where z^{i+k} is any successor event of z^i of the same action.

Proof:

Let S_p=$\Pi(p,\mathcal{B})$, S_p'=$\Pi(p,\mathcal{B}')$, $R_{j=1..3}$ be the sensitization regions of $SR(p,z^i)$, $R'_{j=1..3}$ be the sensitization regions of $SR(p,z^{i+k})$, $\mathcal{B}_{j=1..3}$ be the action sets of $R_{j=1..3}$, and $\mathcal{B}'_{j=1..3}$ be the action sets of $R'_{j=1..3}$.

$\mathcal{B}_1 \subset \mathcal{B}_1' \wedge \forall a \in \mathcal{B}_1$: [$a \in \overline{S}_p \rightarrow a \in \overline{S}_p'$] : since $\mathbf{m}z^{i+k}$ must contain all events of $\mathbf{m}z^i$.

$\mathcal{B}_2 \subset \mathcal{B}_2' \wedge \forall a \in \mathcal{B}_2$: [$a \in S_p \rightarrow a \in S_p'$] : since last events of p are the same for $\mathcal{B}$ and $\mathcal{B}'$ and $\mathbf{m}z^{i+k}$ contains $\mathbf{m}z^i$.

$\mathcal{B}_3 \subset \mathcal{B}_3' \wedge \forall a \in \mathcal{B}_3$: [$a \in \overline{S}_p \rightarrow a \in \overline{S}_p'$] : since R'_1 contains R_1, any chain of events emanating from R_1 satisfying the definition of R_3 must also be contained in R'_3.

Which gives $\mathcal{B} \subset \mathcal{B}' \wedge \forall a \in \mathcal{B}$: [$(a \in S_p \rightarrow a \in S_p') \wedge (a \in \overline{S}_p \rightarrow a \in \overline{S}_p')$]. If p does not exist for $\Pi(p,\mathcal{B})$, then some action $a \in \mathcal{B}$ does not match in S_n and S_f, which means that the corresponding action $a \in \mathcal{B}'$ does not match as well. □

Consider the time complexity of Algorithm 5.1. Let m = # actions in system pomset P, and n = # events in P. Repetition of the while loop on all a,b,c's in the pomset takes $O(n)$ time since there can be at most n occurrences of any action. Repetition of the for loop on all outputs z takes $O(m)$ time since there are at most m outputs in the circuit. Obtaining the new $\mathcal{B}$ takes $O(n)$ time. Advancing p past the initial set of events whose actions do not match in S_n and S_f takes at most $O(mn)$ time. **Loop** takes $O(mn)$ time, where the advancement of p in the if statements take at most $O(m)$ time. The time for the algorithm is therefore $O(n) \times O(m) \times [O(n) + O(mn) + O(mn)] = O(m^2n^2)$. The existence of a polynomial time search algorithm which guarantees detection is important since a state based search usually requires an exhaustive search of all states which in the worst case is exponential to n. Algorithm 5.1 does not perform an exhaustive search of all states which arise due to concurrency (every possible prefix is not searched).

Chapter 6

Control Point Insertion

A technique for test length reduction is considered using control point insertion. Control points have so far been used only to make undetectable faults detectable in DI/SI circuits [31,52] and not to reduce test length. The problem is difficult because it requires deriving a hazard free protocol from an altered circuit when control points are inserted. The method used is based on extraction of the test from the original usage protocol. It involves three steps and algorithms are given for each step.

6.1 Motivation

The insertion of control points is a design for test strategy whose primary purpose is to reduce the length of a test, for example, the at-least-2 test. In synchronous circuits, one of the popular methods to reduce the test length (number of test vectors) is scan design. Such a technique is too expensive for SI circuits of [54] and DI transition signaling circuits because of the high ratio of memory components to combinational components, ie. scan design primarily tests

the combinational components. In addition, transition signaling circuits require input transitions rather than level values provided by scan design. Instead of using scan design, control points can be used to send input transitions to exercise nodes which are hard to exercise using the normal protocol. Questions which must be answered are:

1) Where should control points be inserted?
2) How does control point help in reducing test length?
3) How do we handle races/hazards if arbitrarily control points are inserted?
4) New circuit with control points does not follow usage protocol anymore, so how do we find a new race/hazard-free protocol?

As motivation for the problem, we illustrate the possibility of design for testability through the example of Fig. 6.1. It is a mod-32 counter and under normal operation, has only the I/O port set of $\{a, x, y\}$. Its specification is $(a; x)^{31};(a; y)$, generating 31 x outputs and a y output on the 32^{nd} count. Since it contains only non-terminal components, an at-least-1 test is sufficient. In order to exercise every

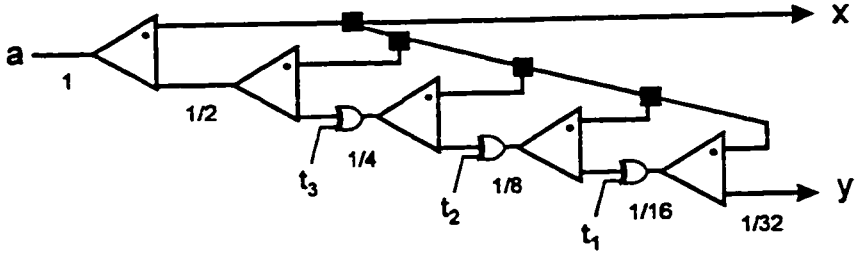


Fig. 6.1: Insertion of control points in a mod-32 counter (square black dots are XOR's as well).

node, the length of the test sequence ***must*** be 32 since it takes that long to exercise the last node y. Consider the insertion of one test control point t through the addition of a single XOR gate. Three possible insertion points are illustrated: t_1, t_2, t_3. The most obvious insertion point is at the least used node, which is t_1 (the fractions above the nodes indicate their utility, node a being 1 and node y being

exercised 1/32 times as much as node *a*). The test length using t_1=16+1=17: 16 inputs from *a* exercising the now partitioned mod-16 counter and one input from t_1 to exercise the last node *y*. The other partitionings give t_2=8+3=11 and t_3=4+7=11. For this example, inserting the control point at t_2 or t_3 gives the shortest test length. The most obvious place for insertion does not always give the most reduction in test length. For the mod-n counter, the best place for insertion is right in the middle, partitioning the counter in half. For a mod-1024 counter, the length of the test sequence is reduced from 1024 to 63 with the insertion of a single test point. To obtain a test length which is linear to the size of the circuit, the mod-n counter can be partitioned into fixed size mod-*k* counters for some constant *k* through the insertion of $2^n/k$ test points.

Fig. 6.2 illustrates the general case which partitions the circuit through the insertion of control points t_i. The area cost is a single input pad for *t*, an XOR for each t_i, and a sequence generator for scheduling transitions to t_i. An optimization strategy to synthesize sequence generators is given in [41]. The speed overhead is low because the average case time of the circuit is not affected much when an XOR is inserted into a low use node. These control points are "non-invasive" in the sense that no changes are required to switch the circuit between normal mode and test

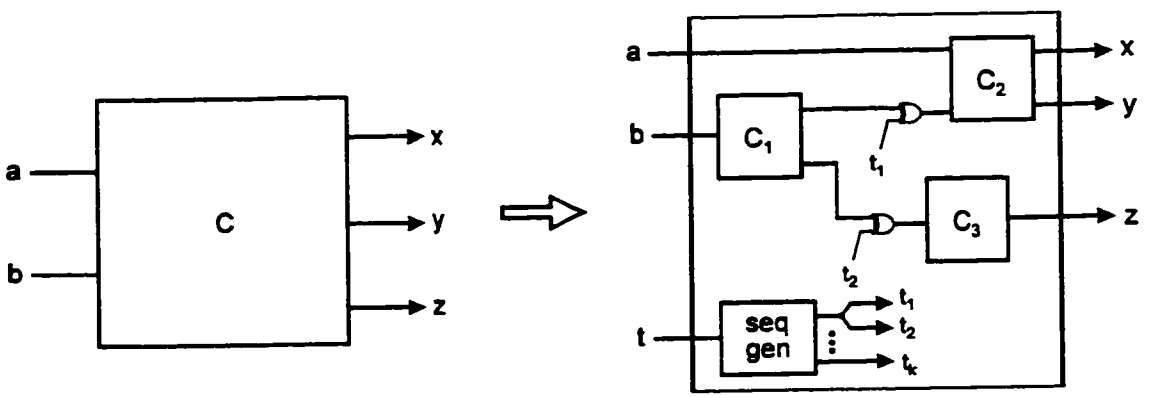


Fig. 6.2: General case of control point insertion.

mode. The same circuit is used when running the normal execution and the test execution. The test can just be considered another possible use of the circuit. This is different from "invasive" control points such as used in [31,52] in which a control point is a disconnection made on a wire to transform a node into an output and an input wire. Changing the circuit from test mode to normal mode requires that this disconnection be closed, either through (i) expensive I/O pads, two per control point, or (ii) a (DI/SI) shift register with a multiplexer for each control point. The disadvantage of multiplexer control is that even though the circuit has passed testing, the circuit will not be guaranteed to have fault-free behavior if there is a saf which prevents the circuit from properly switching from test mode to normal mode. A saf on the test control line may lock only one of many multiplexers in test mode. It is possible to introduce acknowledge signals to the test control line at every control point to guarantee that the mode is switched properly, but this requires extra hardware in the form of DI multiplexers and a completion tree for the acknowledges. The use of non-invasive XOR control points do not require switching between normal and test mode.

As with the case of sensitization, deriving the test only from the circuit will be too difficult without making use of information from the original protocol. The problem is then: Given an at-least-2 test behavior and allowing k control points to be inserted, find the best insertion that leads to a minimum length test. The test can be derived by extracting sections of the at-least-2 pomset, where the effect of control point insertion is the replacement of internal events with control point events. The extracted sections of the pomset must contain at least two occurrences of every action. This problem is more difficult than that of sensitization because it must extract a series of pomset segments instead of a single segment and the pomset segments must be compatible with each other during the test, ie. the partial states between each pomset segment must match. Control points will be used to

facilitate the matching of these states.

The general test derivation will follow a series of steps: gap finding, gap matching, and finding a tour of gaps.

6.2 Gap Detection

To derive the pomset segments which will be used in the test, we first extract a series of gaps, G_1, ..., G_n (Fig. 6.3) from the original at-least-2 pomset P, where a gap is a pomset segment containing events which are redundant and not required to satisfy the at-least-2 condition of the new test behavior. We want to repeatedly find the largest gaps in P to minimize the test length. Finding the largest gaps reduces the total number of gaps in P, so there are a smaller number of gaps to skip and hence fewer partial states to match, which usually implies fewer control points required. A larger total number of gaps of smaller size may possibly produce a shorter test length, but we don't know the actual test length and number of control points required until after the control points are inserted and the final test is derived.

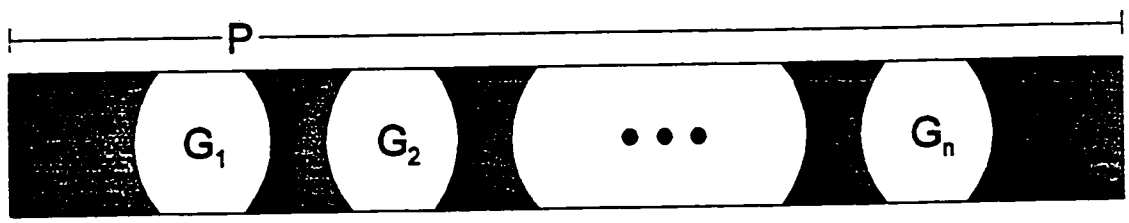


Fig. 6.3: Event gaps in an at-least-2 test behavior.

Problem 6.1: (Gap Finding)

Given a pomset P, decide if there exists prefixes r and s where $r \leq s \leq P$ such that

(i) the events in a given set E (those which must be covered to fulfill at-least-2 requirement) are found in $\mathcal{E}(r) \cup \mathcal{E}(P-s)$, and

(ii) $|\mathcal{E}(r)| \leq D$ and $|\mathcal{E}(s-r)| \geq GAP\_SIZE$. □

Given a pomset P (Fig. 6.4), the maximum size gap can be found by varying r and s. One can initially select a prefix r such that it satisfies the condition of having smaller number of events than D and then advance s forward from r to obtain the largest size gap. But since r is variable, there is potentially an exponential number of prefixes which can be selected. It is this variability of two prefixes simultaneously which makes the problem difficult.

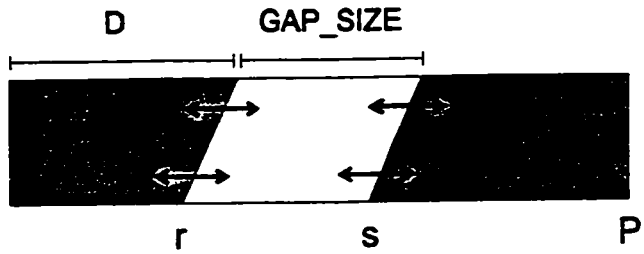


Fig. 6.4: Variability of two prefixes to find maximum gap size.

Theorem 6.1:

Problem 6.1 is NP-complete.

Proof:

A nondeterministic TM could make a guess of r and s and check for the satisfaction in polynomial time. To complete the proof, we use a polynomial time reduction from the knapsack problem.

Knapsack problem:

Given a set $X = \{x_1, x_2, ..., x_n\}$ and $w(x_i)$ = weight of x_i, $v(x_i)$ = value of x_i, $i=1,2,..n$.

Decide if there exists $Y \subseteq X$ such that

$\sum_{i=1..k} w(x_i) \leq B$ for $x_i \in Y$, and $\sum_{i=1..k} v(x_i) \geq K$ for $x_i \in Y$.

The Reduction Procedure: For a given instance of the knapsack problem, obtain an instance of the gap detection problem as follows:

(i) pomset P is given by Fig. 6.5.

(ii) E is given by: $\{x_1,x_2,...,x_n,u_1,...,u_n\}$.

(iii) $D = B$, and $GAP\_SIZE = K + (\sum_{i=1..n} w(x_i)$ for $x_i \in X) - B$.

Claim: the knapsack problem has a yes decision if and only if the gap detection problem has a yes decision.

If the knapsack problem has a yes answer, and without loss of generality, assume $Y = \{x_1,...,x_k\}$. Then we could find prefix r and s of P as follows: $\mathcal{E}(r) = \{x_i$ and its successor events up to and including the first u_i for $i = 1,...,k\}$, and $\mathcal{E}(P-s) = \{$second instance of u_i and its successor events for $i = k+1,...,n\}$. Then it is direct to verify the size of $\mathcal{E}(r)$ and $\mathcal{E}(s-r)$ exactly match the conditions required.

If the gap detection problem has a yes answer, and suppose r includes some x_i but not all of its successors up to the first occurrence of u_i, then we can find another r' by excluding that x_i and r' still satisfies the condition. Similarly, if s contains some second occurrence of u_i but not all its successors, we could include all its successors to form a new s' which still satisfies the gap condition. Moreover, the resulting r' and s' constitute a yes decision for the knapsack problem. □

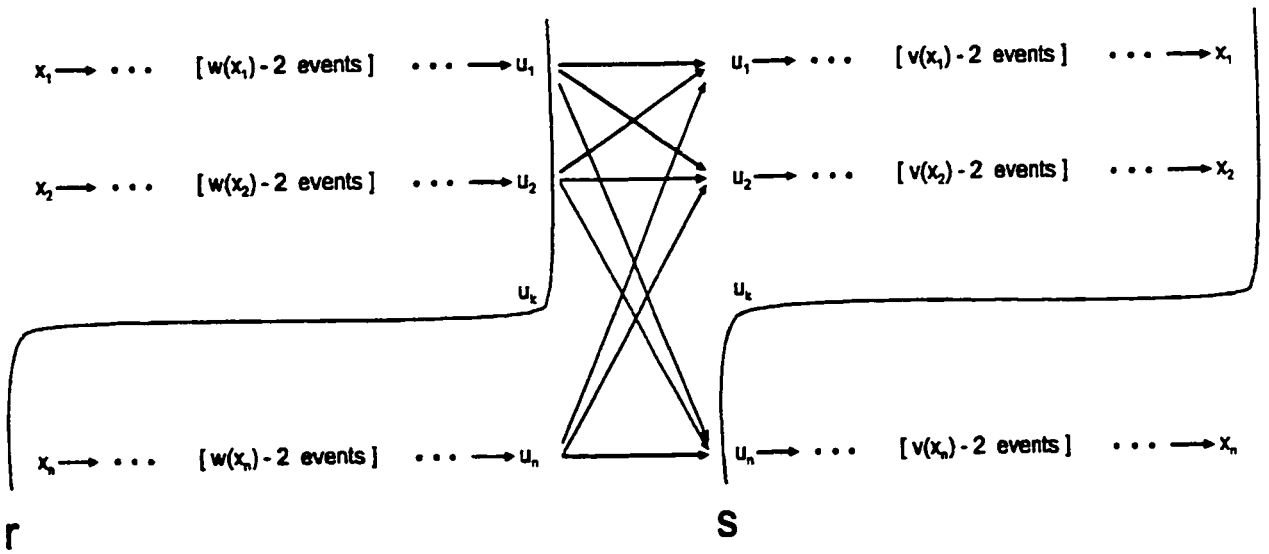


Fig. 6.5: Instance of gap detection problem in form of knapsack problem.

The pomset is structured in the form of Fig. 6.5. The left and right halves consist of linear orders representing the weights and values of each x_i. Each of these linear orders contain exactly $w(x_i)$ events on the left and $v(x_i)$ events on the right. Prefixes r and s is selected such that when r includes the weight of x_i, s also

includes the value of x_i. Prefixes r and s are moved down to maximize the value, but not so far as to go past the weight limit. The prefixes can select any subset of x_i.

Problem 6.2:

Solve Problem 6.1 given a fixed prefix r. □

Problem 6.1 is NP-complete, however, fixing prefix r first and then varying s to find the maximum size gap can be solved in polynomial time. Even though s can potentially have an exponential number of prefixes, s can still be found from the partial order without checking every prefix. As a heuristic, prefix r can be selected as the initial state/null prefix of the pomset and the largest gap can be found by advancing s from r. The next largest size gap can be found by fixing r immediately after s and then repeating finding the maximum size gap.

The following algorithm finds a sequence of gaps in P which consecutively satisfy the maximum gap size requirement.

Algorithm 6.1: (Gap Finding Algorithm)

$p = \Lambda$; $i = 1$; $S_A = \mathcal{A}(P)$;

while $p < P$ **do**

$r = p$;

Let $\tau(a)$ be the total order of the event occurrences of action a for $a \in S_A$.

$s = \bigcup_u \mathbf{M}u$ where (i) u is the last event of chain $\tau(a)$ if action a has been covered only once so far, and (ii) u is the second last event of chain $\tau(a)$ if action a has not been covered so far.

if $|\mathcal{E}(s - r)| \geq GAP\_SIZE$ **then**

$G_i = s - r$;

$i = i + 1$;

endif

$p = s$;

if $p < P$ **then**

Advance p past immediate successor events of s (if they exist);

Advance p to the next stable state;

S_A = set of actions which has not been covered at least twice so far;

endif

endwhile □

Initially, the algorithm sets prefix r as the null prefix Λ, and prefix p is used as a temporary prefix which advances from beginning to end of the pomset P. S_A is a set which records which actions has not yet appeared twice in the non-gap regions up to the current prefix p. Initially, S_A contains all actions.

Once r is fixed, the maximum size gap recorded by s can be found by taking the union of the maximum prefixes of the events not yet covered at least twice. These events can be easily located from the chain $\tau(a)$. Every DI pomset behavior must consist of chains/total orders of events with the same action label according to the DI axioms [45,46,47] because every signal transition firing at a circuit node must be ordered; there cannot be concurrent signal transition firings at the same node (no autoconcurrency in the partial order). Either the last or second last event of the chain is taken to ensure that the non-gap region contains at least two occurrences of that chain's action, as illustrated in Fig. 6.6 where the shaded regions contain at least two occurrences of every action. Taking the union of the maximal prefixes gives a cut s which must be consistent because there cannot be an arc from the right side of s to the left; otherwise as illustrated for event v in Fig. 6.7, $\mathbf{M}a$ would not be a maximal prefix since the event v is included in the successor set of a. The gap preceding the maximal prefixes is guaranteed to contain the maximum number of events which are redundant because taking the maximal prefixes is the minimum requirement for excluding the required at-least-2 events from the gap, hence the gap will be maximized; increasing the gap size by one more event will either make s inconsistent or violate the at-least-2 condition.

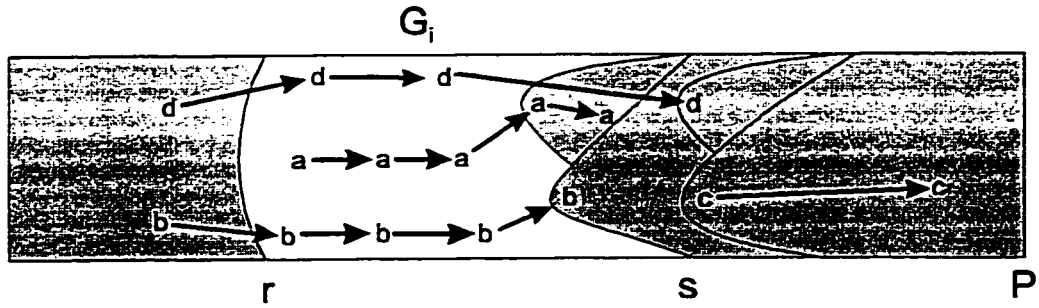


Fig. 6.6: Maximum size gap starting from a fixed prefix *r*.

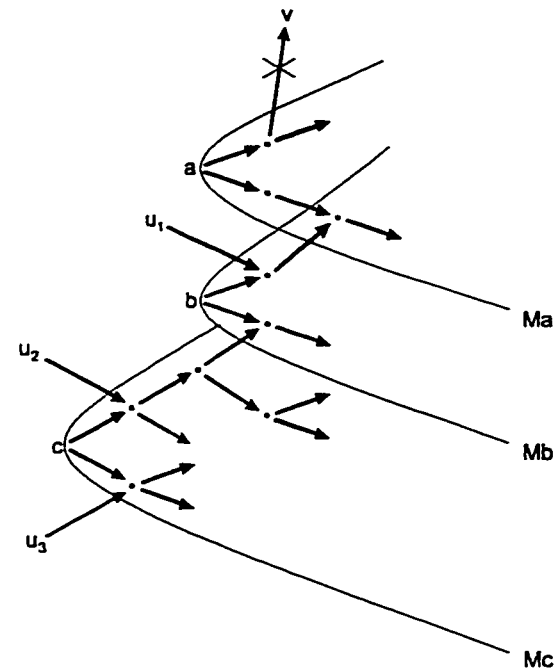


Fig. 6.7: Union of maximal prefixes.

The next prefix *r* for the next gap is found by advancing past the immediate successor events, which will be the last or second last events in the chains, thereby including these necessary events in the non-gap regions. This prefix needs to be advanced to the next stable state since internal transitions are not controllable by the test. Once the internal transitions have stabilized, the circuit/prefix will accept interface inputs and the search for the next maximum size gap proceeds from there.

Consider the time complexity of finding a single maximum size gap, or

equivalently the prefix s following r. Let m be the number of actions in P and n be the number of events in P. Assuming that the totally ordered chains have been stored, the last two events of every chain can be immediately accessible. Finding the union of all the maximal prefixes by marking all the successor events only require $O(n)$ time because the marked region contains at most n events and the marking of each maximal prefix does not need to overlap. The prefix s can be recorded as either its last events or the arcs crossing the cut s. In both cases, the events with arcs incoming to the marked events must be found; these are events $\{u_1, u_2, u_3\}$ in Fig. 6.7. $O(n)$ time is needed to find the immediate predecessors of the marked region. Finding a single maximum size gap therefore requires $O(n)$ time ([49] was overly pessimistic in reporting $O(mn)$ time).

Algorithm 6.1 repeatedly finds maximum size gaps. Since every iteration must increase the non-gap region by at least one required event in advancing p past the immediate successors of s, and there are $2m$ required events to cover, the gap finding algorithm requires $O(mn)$ time.

Fig. 6.9 demonstrates the gaps (shaded regions) found by Algorithm 6.1 in the pomset behavior of the circuit of Fig. 6.8. The value of *GAP_SIZE* used is 15 events.

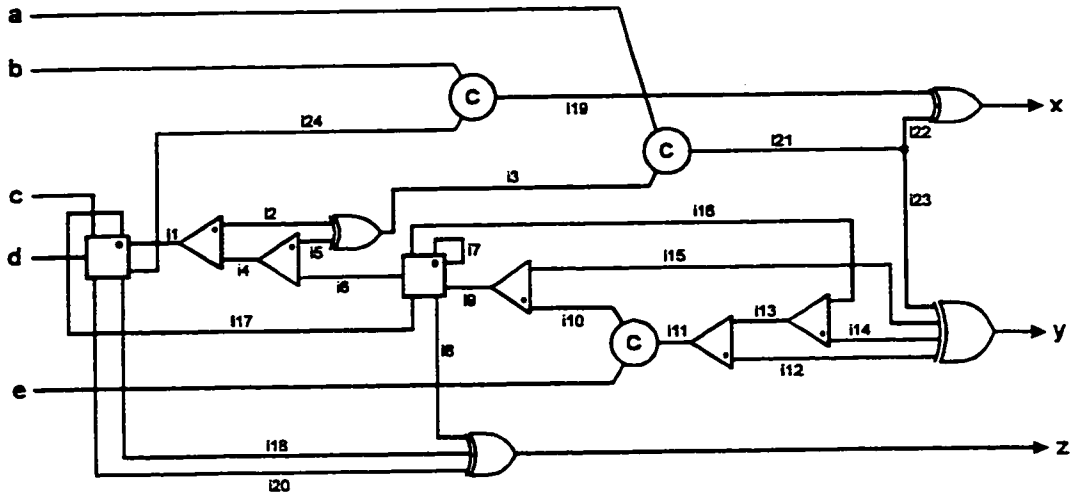


Fig. 6.8: Example circuit.

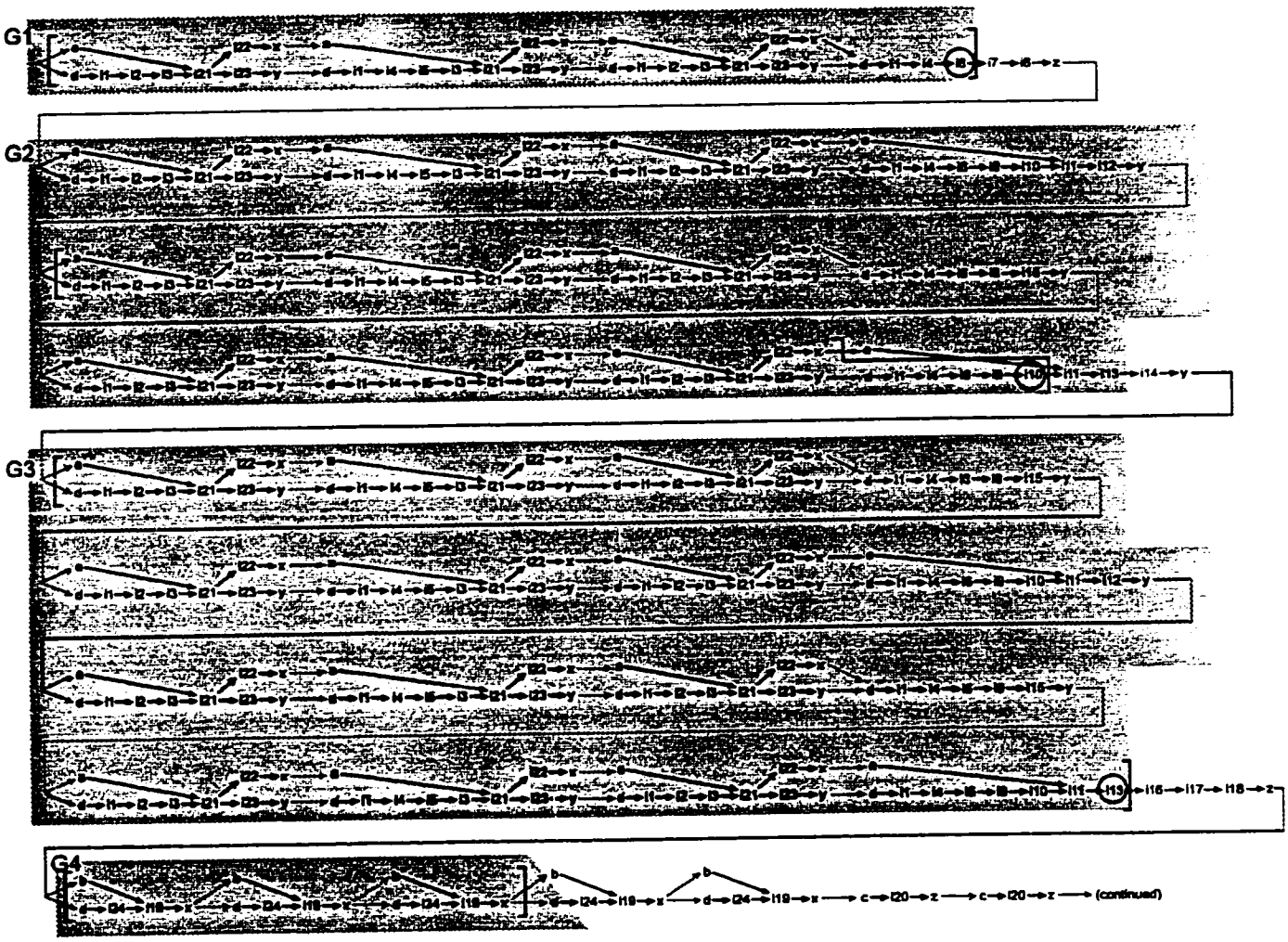


Fig. 6.9: Gaps extracted by the gap finding algorithm (continued).

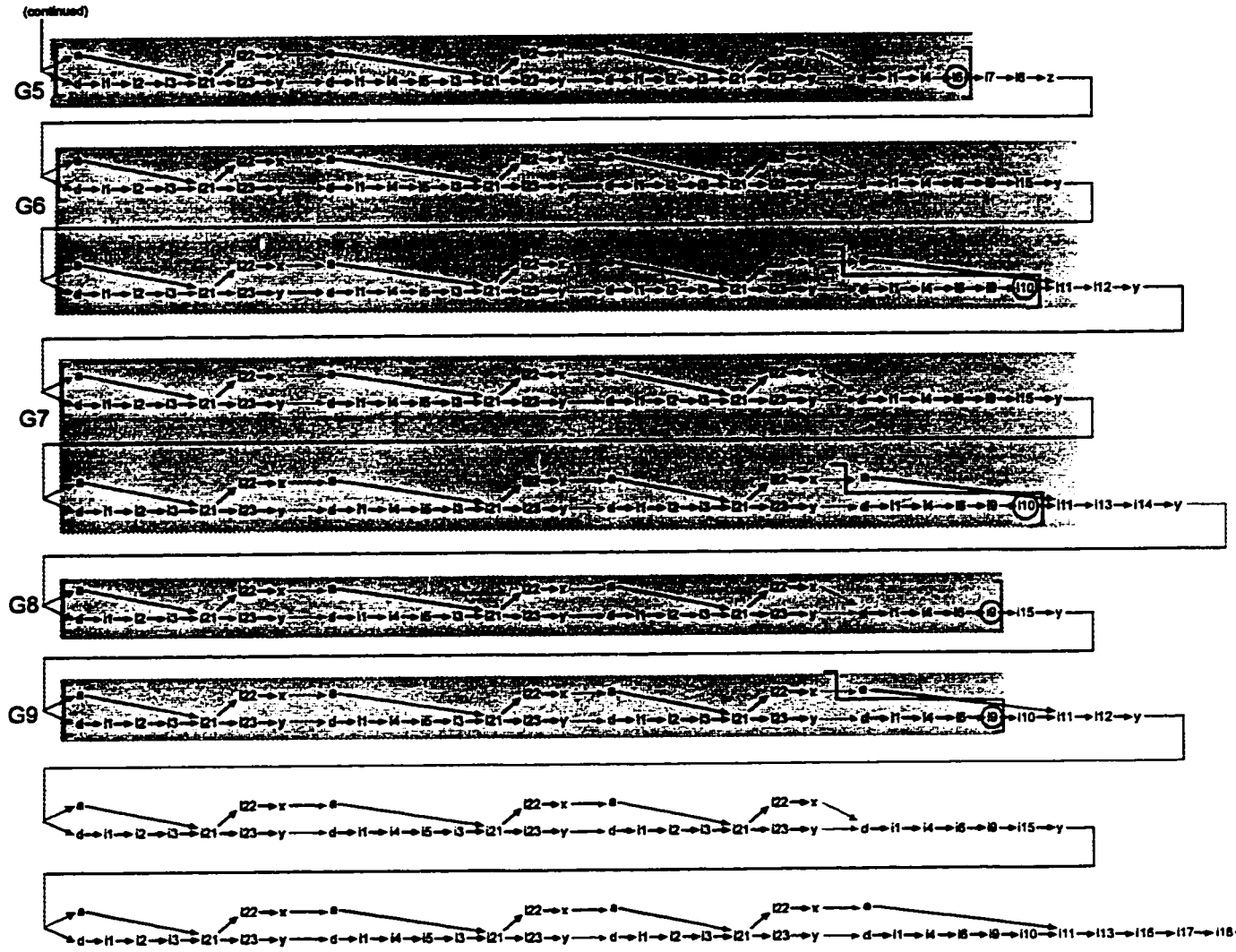


Fig. 6.9: Gaps extracted by the gap finding algorithm.

6.3 Gap Matching

Once all gaps are found, the test can cover the non-gap behavior by either sequentially skipping gaps from G_1 to G_n or non-sequentially jumping from one gap to another to cover the at-least-2 behavior. Suppose r_i ($1 \le i \le n$) is the prefix at the beginning of gap G_i and s_i is the prefix at the end of G_i. Sequential skipping of gaps is performed by: for $i=1$ to n, jump from r_i to s_i and execute the behavior from s_i to r_{i+1}. The more general non-sequential skipping of gaps is performed by repeatedly jumping from r_i to s_j ($1 \le i,j \le n$) and executing the behavior from s_j to r_{j+1} until all the non-gap behavior is covered.

A correct jump from r_i to s_j requires that the partial state at r_i must match the partial state at s_j, $\Pi(r_i,\mathcal{B})=\Pi(s_j,\mathcal{B})$ for some $\mathcal{B}$ which are actions subsequently executed after s_j. Prefix r_i must also correspond to a global stable state because any unstable tokens at r_i may interfere with input tokens sent in from s_j in the continuation of behavior execution and cause unpredictable results. This is similar to the previous sensitization chapter where the circuit after initialization is in a global stable state and some partial state needs to be found. If the partial states of r_i and s_j match exactly, then the gap can immediately be skipped according to Theorem 5.1. Otherwise, partial execution of the prefixes is performed to obtain a match: forward advancement of r_i and backward advancement of s_j. This is equivalent to shrinking the size of the gap to find a match. The second tool which can be used to find a match is the insertion of control points. Since we are jumping from r_i to s_j, control points can be used to inject tokens to force s_j to match r_i if the partial state of r_i contains a token/action while the partial state of s_j does not. The token is injected via an XOR at the circuit node corresponding to that action. Since the internal component actions of toggles and demultiplexers cannot be controlled (without changing the components by adding an explicit set/reset) these internal actions must be matched by partial execution of the gap.

Consider the three possible types of actions: (1) terminal component input actions, (2) non-terminal component input actions, (3) internal actions of memory components. Component output actions are not considered because every component output is also a component input, except the interface outputs; tokens at the interface outputs will never need to be matched at stable state since they will always be consumed by the environment. Let $S_g=\Pi(r_i,\mathcal{B})$ and $S_h=\Pi(s_j,\mathcal{B})$ be the partial states at prefixes r_i and s_j. In case (1), if $a\in\mathcal{B}$ is a terminal component input action and $a\notin S_g \wedge a\in S_h$, then a control point can be inserted at a to make the partial states match: since S_g is missing a token at partial state r_i, a token is injected into the control point at a to make $a\in S_g$. Execution of the behavior can then proceed. It is not necessary to wait for the injected token to stabilize because the token will be consumed as part of the new normal DI behavior. The inserted XOR will now be part of the new circuit, and the test will consist of the new shortened at-least-1 or at-least-2 test. Note that we must inject the token to change S_g to match S_h and not the reverse since the continuation of execution of the behavior past s_j requires that S_h must be true. If $a\in S_g \wedge a\notin S_h$ where a is a terminal component input action, then $a\in S_g$ indicates a pre-existing token at the input of the component. Inserting a control point at a and injecting a second token will cause token cancellation at a and make $a\notin S_g$, thereby forcing the partial states to match. However, insertion of a control point is not valid here because this occurs only in the fault-free circuit; token cancellation is not guaranteed to occur in the faulty circuit because a token may not exist at the input of the terminal component. For this case, partial advancement of the gap would have to be used to make the gap match.

In case (2), if a is a non-terminal input action, then $a\in S_g \wedge a\notin S_h$ cannot occur since r_i must be a stable state; a token at a non-terminal input implies an unstable state. If $a\notin S_g \wedge a\in S_h$ a control point can be inserted as in the case of the terminal

component to inject a token to make $a \in S_g$. Note that since the partial state at S_h is not stable in this case, prefix s_j is not required to be a stable state in the original pomset. However, in the altered circuit with the control point, the new test views S_h as a stable state since we wait for the circuit to stabilize at S_g and the environment is holding the token that will be injected into the non-terminal input place.

In case (3), the internal actions of memory components must also match in S_g and S_h, where memory components contain internal net places/actions such as in toggle, demultiplexer, arbiter, and RCEL components. Fig. 6.10 illustrates example system behaviors containing internal actions of the toggle and demultiplexer, where p is the initial prefix. Although the C-element is usually considered a memory component, its net model does not require internal actions (its state is recorded by

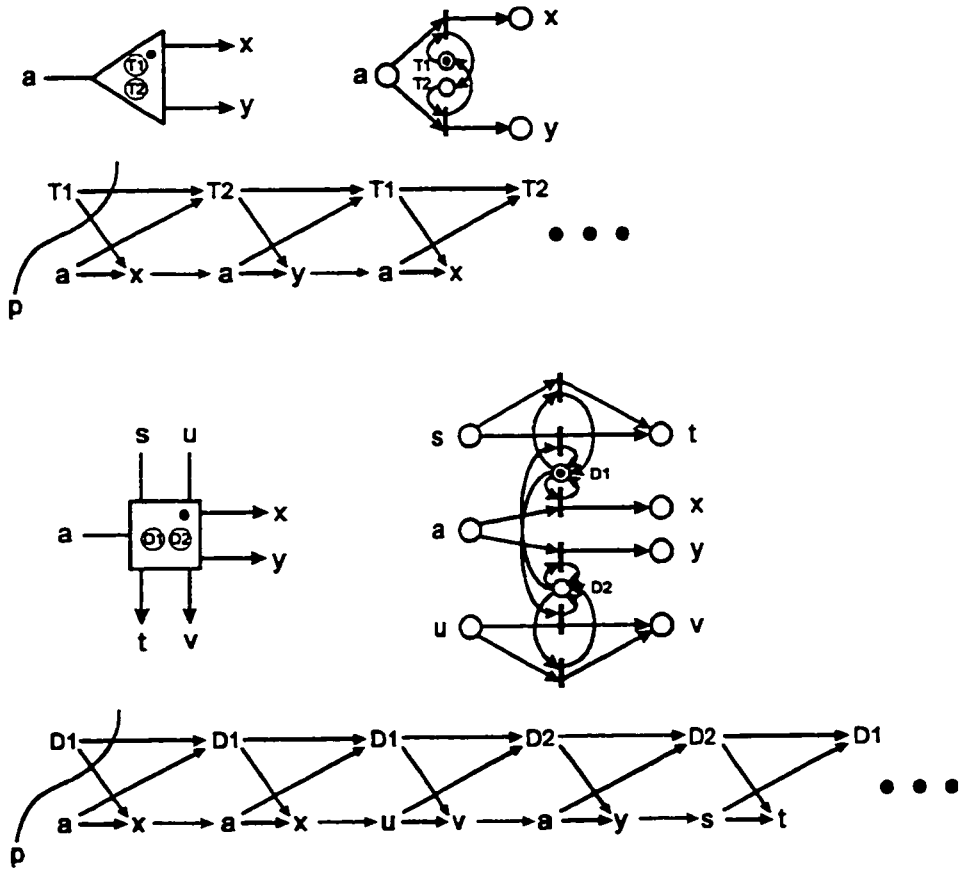


Fig. 6.10: Example system behaviors containing internal component actions.

the presence/absence of input tokens only). If $a \in \mathcal{B}$ is an internal component action and a does not match in both S_g and S_h, then a control point cannot be used to change the state of S_g (not unless the set of basic components is changed to allow explicit change of internal state through an additional control input in the memory component and fault-freeness of these inputs is assumed). In this case, internal component actions in S_g and S_h must be matched by partial execution of the gap: forward advancement of r_i and backward advancement of s_j. The partial advancements should be smaller than the size of gap G_j because the reduction in test length by deleting gap G_j is offset by the increase in test length by the partial advancements of the gap.

A component such as the arbiter can actually sometimes behave like a terminal component to stop a token and sometimes behave like a non-terminal component to let a token pass, depending on its internal state. For such a component, control points can be inserted according to either of the above case (1) or case (2). Its exact fault-free behavior will be known from its internal state and the appearance of a join in the pomset involving arbiter actions, where existence of a join indicates terminal component behavior. Matching of internal actions in case (3) will ensure that a jump from one prefix to another can be made.

Let G_1, ..., G_n be gaps in pomset P, $G_i = s_i - r_i$, $G_j = s_j - r_j$, $r_i \le g \le s_i$, $r_j \le h \le s_j$, $D_i = |\mathcal{E}(g-r_i)|$, and $D_j = |\mathcal{E}(s_j-h)|$ for $1 \le i,j \le n$. Let $S_g = \Pi(g,\mathcal{B})$, $S_h = \Pi(h,\mathcal{B})$, and $d(S_g,S_h) = |(S_g \cap \overline{S}_h) \cup (\overline{S}_g \cap S_h)|$ (number of actions which differ in partial states of g and h).

Problem 6.3: (Gap Matching)

For some set of partial state actions $\mathcal{B}$ required to be matched and some constant K, find prefixes g and h such that $d(S_g,S_h) \le K$ and $|\mathcal{E}(G_j)| - D_i - D_j \ge$ *MIN_GAP_SIZE*. □

K represents the number of control points available which is used to force the non-matching actions of g and h to match while D_i (D_j) is the number of gap events g has advanced past (h has backward advanced past).

Fig. 6.11 illustrates the problem where we want to jump from r_i to s_j (g to h) to skip gap G_i which would otherwise need to be executed. g is initially set to r_i and advanced forward while h is initially set to s_j and advanced backward.

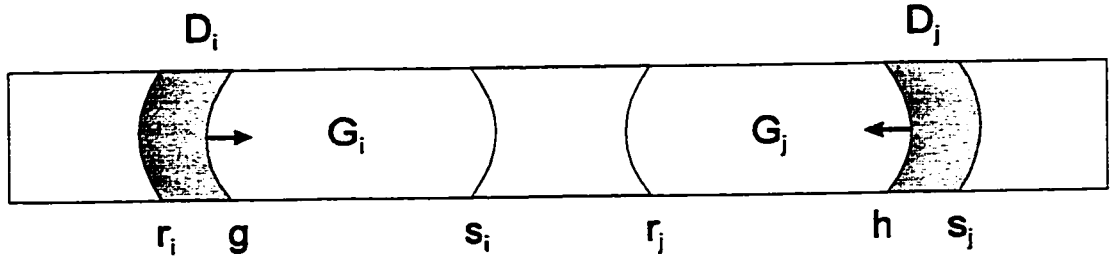


Fig. 6.11: Event gaps in an at-least-2 test behavior.

Theorem 6.2:

Problem 6.3 is NP-complete.

Proof:

Restrict problem to 3-SAT by constructing a pomset in the form of Fig. 6.12 for $\mathcal{B}=\mathcal{A}(P)$ where e is any input event and o is any output event with unique action label, and $\forall_i$: x_i and $\bar{x}_i$ are internal events. Pomset ($g - r_i$) consists of chain structures containing $x_i/\bar{x}_i$ events forming joins with e events representing clauses ($x_i + x_j + x_k$) which are repeated such that the number of events in each chain structure is $\geq D_i$. For every stable prefix g, g° must contain one event/literal in each clause, enabling that clause to be true. g° cannot contain any e since h° does not contain any matching e. Pomset ($s_j - h$) contains events which make every x_i true or false (x_i or $\bar{x}_i \in h^\circ$). Since DI behaviors disallow autoconcurrency, the x_i in every chain structure of ($g - r_i$) is distinctly labeled $\{x_i, x_i', x_i'', ...\}$ and appears concurrently in ($s_j - h$). h° containing one event in the set $\{x_i, x_i', x_i'', ...\}$ for some i implies all of these events which appear are in h°, otherwise the prefix is not a reachable prefix. The formula is satisfiable if and only if

$d(S_g, S_h)=0$. □

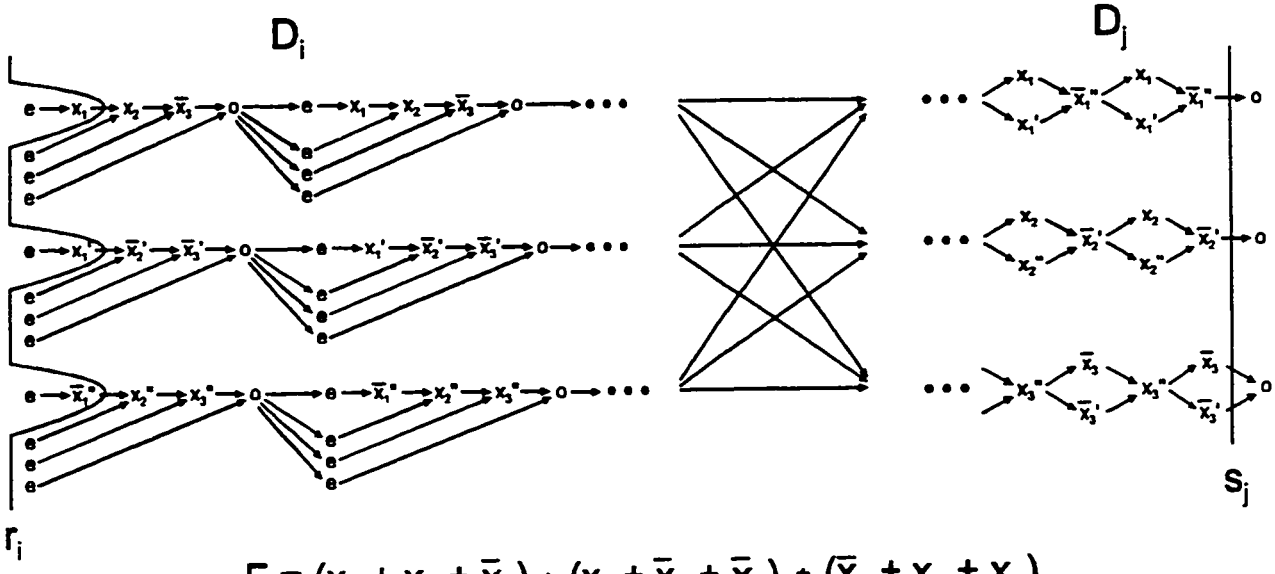


Fig. 6.12: Restriction of gap matching problem to 3-SAT problem.

The two prefixes r_i and s_j are moved at the same time. The left side of the pomset records the formula, with each chain structure representing one clause. The right side records the true or false value of each variable. The cut that g makes passes through each chain structure on the left; since it is sufficient for only one of the literals to be true within the three literal clause to make the entire clause true, we can let g select that one literal when it cuts that chain structure. The single, double, etc. apostrophes eliminate autoconcurrency and are only an implementation requirement of DI pomsets. The number of apostrophes increase as we go down for every chain structure on the left. The right side of the pomset takes all of the literals appearing on the left hand side and combines them into negated and non-negated concurrent groups.

A heuristic for matching gaps will therefore be used:

Algorithm 6.2: (Gap Matching Algorithm)

detect_state(g, h, $\mathcal{B}'$)

$P'=g-h$; $S_g = \Pi(g,\mathcal{B}')$; $S_h = \Pi(h,\mathcal{B}')$;

while $(S_g \neq S_h) \wedge (|\mathcal{E}(G_j)| - D_i - D_j \geq MIN\_GAP\_SIZE)$ **do**

foreach $a \in (S_g \cap \overline{S}_h) \cup (\overline{S}_g \cap S_h)$ **do**

Let u be the last event in chain $\tau(a)$ preceding h where $\mathcal{A}(u)=a$.

if $(\overline{S}_g \cap S_h)$ **then**

$h = h - (P'-\mathbf{M}u)$; {backward advance h before u}

if $(S_g \cap \overline{S}_h)$ **then**

$h = h - (P'-\underline{\mathbf{M}}u)$;

endfor

$S_h = \Pi(h,\mathcal{B}')$;

endwhile

if $S_g = S_h$ **then** Return (g, h) **else** Return (ϕ);

end {detect_state}

main($\mathcal{B}$, K)

$g = r_i$; $S_{sj} = \Pi(s_j,\mathcal{B})$

Repeat

Let $\{CB_1, ..., CB_m\}$ = sets of all combinations of K actions out of $\mathcal{B}$.

for $k = 1$ to m **do**

$h = s_j$; $\mathcal{B}' = \mathcal{B} - CB_k$

if detect_state(g, h, $\mathcal{B}'$) $\neq \phi$ **then** Return(g,h) and exit main.

endfor

Advance g to the next stable state and obtain new $S_g = \Pi(g,\mathcal{B})$ such that

$d(S_g,S_{sj})$ is minimal;

Until $|\mathcal{E}(s_i - g)| < MIN\_GAP\_SIZE$

Report no match found.

end. {main} □

Algorithm 6.2 fixes g first and backward advances h to find a match. We assume a corresponding reachable prefix operation is performed in backward

advancement in the same way it was performed for forward advancement in the previous chapter, ie. for every fork in the prefix which does not form a reachable state, backward advance the prefix to exclude that fork event and its successors. The subroutine **detect_state**() requires polynomial time and guarantees detection if a matching state exists. Combinations of K actions are subtracted from $\mathcal{B}$ (representing the actions which may be matched with control point insertion) so that the exact match algorithm **detect_state**() can be used. This is justified for small K, which is usually the case when we do not want to insert too many control points. Execution of **detect_state**() is similar to the Sensitization Prefix Search Algorithm of the previous chapter, except the search advances backward instead of forward and matches two partial states instead of three.

If a large OR gate is used to check for spurious tokens at all the inputs of terminal components at circuit initialization, then there are no constraints on how many times a control point must be exercised. All excitory faults will be detected by the large OR gate, and by Theorem 4.1, an at-least-1 test is sufficient to detect all the remaining inhibitory faults. The sequence generator used to distribute the tokens to the control points is also self-testing because it can now be considered as part of the circuit and at-least-1 is sufficient to detect any saf's in the sequence generator. It can be shown that for any given sequence, it is possible to synthesize a sequence generator in which every node is exercised once.

If a large OR gate is not used to check for spurious tokens at all terminal component inputs, for example if at-least-2 is sufficient for the original circuit or only some terminal component inputs are checked instead of all, then there are constraints which must be met before a control point can be inserted. When a control point is inserted at an internal node of a pomset behavior, it is possible if any one of these conditions hold (assuming that at-least-2 is sufficient to detect all saf's in the original circuit):

1) The node satisfies at-least-1 sufficiency condition of a structural theorem.

2) All paths leading from the node terminates at either an interface output or a terminal component which is check by the OR gate.

3) The insertion is at a node in which a control point already exists (this is the 2^{nd}, 3^{rd}, etc. insertion at this node).

4) If this is the 1^{st} insertion at this node, then the node event must appear at least 3 times in the non-gap behavior (two times to test the node normally + one to exercise the control node a second time).

5) If (3) is not satisfied, then to exercise the control point a second time we can create a new non-gap behavior to be traversed which exercises that node an additional time, ie. if j^2 is to be the second exercise of the control point, then the new non-gap behavior consists of predecessors of j^2 and successors of j^2 containing some output up to a stable state.

6.4 Tour of Gaps

Let P be an at-least-2 test behavior and $P = z_0 ;; G_1 ;; z_1 ;; G_2 ;; \ldots ;; G_n ;; z_n$ where $\{G_1,\ldots,G_n\}$ is the set of gaps extracted, $\{z_0, \ldots, z_n\}$ is the pomset segments between the gaps, and $G_i = s_i - r_i$ for prefixes r_i, s_i. A tour of the gaps is given by the sequence $<G_{\pi(1)}, G_{\pi(2)}, \ldots, G_{\pi(n)}>$ where $\pi(i)$ maps G_i to the new ordering. The test executes the pomset $z_0; z_{\pi(1)}; z_{\pi(2)}; \ldots; z_{\pi(n)}$. A jump of a gap from $r_{\pi(i)}$ to $s_{\pi(i)}$ requires $\Pi(r_{\pi(i)},\mathcal{B}) = \Pi(s_{\pi(i)},\mathcal{B})$ where $\mathcal{B} = \mathcal{A}(\cup_{j=i..n} z_{\pi(j)} \cup s_{\pi(j)}^{\circ})$. The size of $\mathcal{B}$ covers all the actions which appear at the tail end of the behavior $z_{\pi(i)}; \ldots; z_{\pi(n)}$ to ensure that there are no token collisions or races. Fig. 6.13 shows a tour and the partial state set $\mathcal{B}$ requirement for jumping from r_i to s_j.

Problem 6.4: (Tour Finding)

$\forall_{i=1..n}$: Let $S_{r\pi(i)} = \Pi(r_{\pi(i)},\mathcal{B})$, $S_{s\pi(i)} = \Pi(s_{\pi(i)},\mathcal{B})$, where $\mathcal{B} = \mathcal{A}(\cup_{j=i..n} z_{\pi(j)} \cup s_{\pi(j)}^{\circ})$.
Find a tour $<G_{\pi(1)}, G_{\pi(2)}, \ldots, G_{\pi(n)}>$ which satisfies
$\sum_{i=1..n} d(S_{r\pi(i)}, S_{s\pi(i)}) \leq K$. □

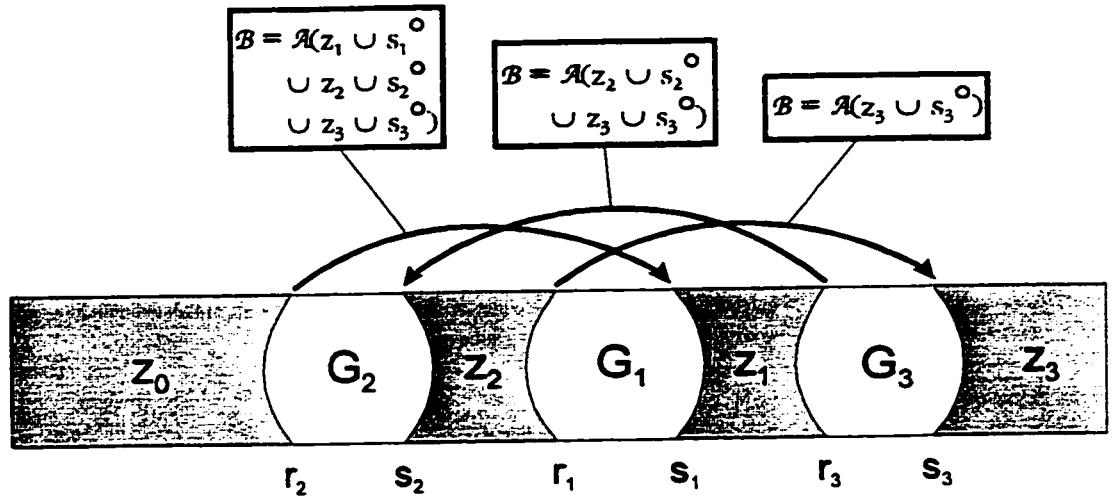


Fig. 6.13: A tour and its partial state set $\mathcal{B}$ requirements.

Assume that gap matching is performed by a polynomial time algorithm such as Algorithm 6.2 and fixes r_i, s_i.

Theorem 6.3:

Problem 6.4 is NP-complete. □

Proof:

Suppose we know in advance and fix the tail end of the optimal tour $<G_{\pi(i+1)}, \ldots, G_{\pi(n)}>$ which uses C' control points and wish to find $<G_{\pi(1)}, \ldots, G_{\pi(i)}>$. If $\mathcal{B}_i = \mathcal{A}(P)$, then the problem simply reduces to the traveling salesman problem. □

The following are two possible heuristic algorithms for finding a tour.

Algorithm 6.3: (Piecewise Subtour Algorithm)

Break $G_1, \ldots, G_n$ into subsequences containing a constant C number of gaps,

giving subsequences: $(G_1, \ldots, G_C)$; $(G_{C+1}, \ldots, G_{2C})$; ... $(G_{n-C}, \ldots, G_n)$;

$i = n$; $T = \Lambda$; K = number of control points;

while $i > 0$ **do**

Obtain the minimum cost subtour of subsequence $(G_{i-C}, \ldots, G_i)$:

$\min_m [\sum_{m=(i-C)..i} d(S_{r\pi(m)}, S_{s\pi(m)})]$, for all possible subsequences defined by π,

where r and s is obtained by gap matching (Algorithm 6.2).

$T = T + <G_{\pi(i-C)}, \ldots, G_{\pi(i)}>$; $i = i - C$;

$K = K - K'$ where K' is number of control points used in subsequence;

if $K < 0$ **then**

Report insufficient number of control points and exit;

endwhile

Return tour T. □

The algorithm moves backward, obtaining the tail end of the tour first so that only partial states need to match. Partial states cannot be used when scanning forward because the partial state requirement $\mathcal{B}$ cannot be known until the tail end of the tour is fixed. Forward scanning requires full global states to match which is more expensive. Alternatively, an insertion algorithm can be used which starts with a sequence containing the last gap G_n and inserts the other gaps one at a time into the sequence at the location which gives the least cost increase.

Algorithm 6.4: (Insertion Tour Algorithm)

1) $T = \langle G_n \rangle$; {start with tour consisting of the last gap}

2) $i = n - 1$; C = maximum number of control points allowed;

while $i > 0$ **do**

Insert G_i into $T = \langle G_{\pi(i+1)}, \ldots, G_{\pi(n)} \rangle$ which gives the minimum cost increase C': ie. insertion of G_i between G_j and G_k increases the total cost by cost of $\langle \ldots, G_j, G_i, G_k, \ldots \rangle$ – cost of $\langle \ldots, G_j, G_k, \ldots \rangle$;

$C = C - C'$;

$i = i - K$;

endwhile

if $C < 0$ **then**

Report insufficient number of control points and exit;

3) Return tour T. □

For both algorithms, if there are not enough control points, then the first gap G_1 can be deleted by direct execution of the behavior from initial state to G_2 and then the algorithm repeated. Alternatively, any gap G_i can be deleted. This process

of deleting gaps and re-running the algorithm is repeated until there are a sufficient number of control points available.

Other, more sophisticated tour finding algorithms are possible, such as branch and bound algorithms, but we will not explore this any further.

The example of Fig. 6.9 shown previously shows the simplest case where a linear tour is used and K=4 control points (setting any $K \geq 1$ also gives solutions which reduce test length). The location of control points which will be used to inject tokens are circled, {*i*6, *i*9, *i*10, *i*13}. The square brackets indicate the actual gaps used after gap matching is performed. With a more intelligent selection of control points, *i*9 can be deleted and replaced with *i*6. The reduction in the number of events required in testing is from 709 to 178 which is a significant reduction. This at-least-2 behavior is only one possible use of the circuit. Without control points, the length of this at-least-2 behavior cannot be reduced much more, except for the short redundancy at gap *G4*. Generally, circuits which have a high degree of encoding such as this circuit and the circuits of [41] will allow greater reductions. Since the length of the pomset represents the approximate time required for testing, the deletion of 75% of the test pomset gives a corresponding time reduction. When K is set to *0*, this strategy can be used to shorten test behaviors without using control points. Behaviors which contain redundancies in exercising the circuit by the environment will automatically be detected, such as gap *G4* which can be skipped without requiring any control points.

Chapter 7

Additional Examples

7.1 Example #1 (control point insertion)

Fig. 7.1 shows a circuit for checking whether an input sequence matches a predetermined sequence. The input consists of a sequence of symbols {*b*, *c*, *d*, *e*}. For each input symbol which matches the predetermined sequence, an f output is generated. When a symbol received does not match, an *n* output is generated and the circuit is reset to receive the next input sequence. This is different from a sequence detector in that it checks for a repeating sequence and makes a declaration as soon as there is a mismatch, rather than detecting the existence of a pattern.

Fig. 7.2 shows an example checker for a particular sequence. The implementation consists of a sequence generator and a functional block which compares it with the input sequence. Toggles with reset inputs are used, where an input transition resets the toggle to its initial state; the toggle reset can be easily incorporated with the circuitry which already exist for component reset at

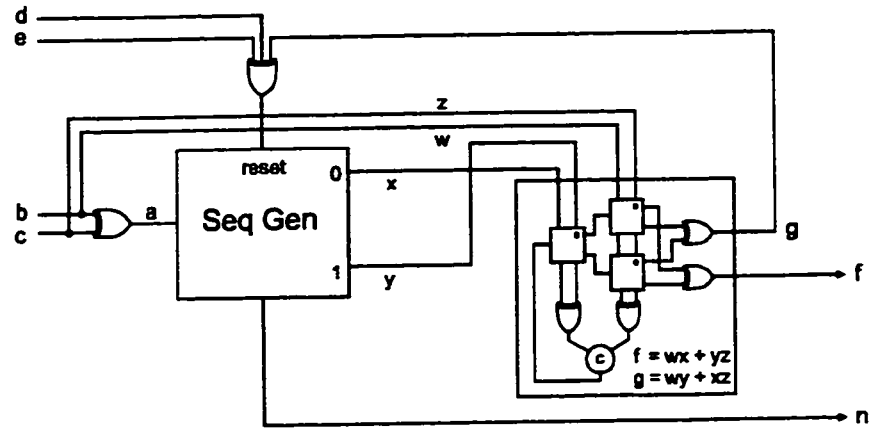


Fig. 7.1: Sequence checker.

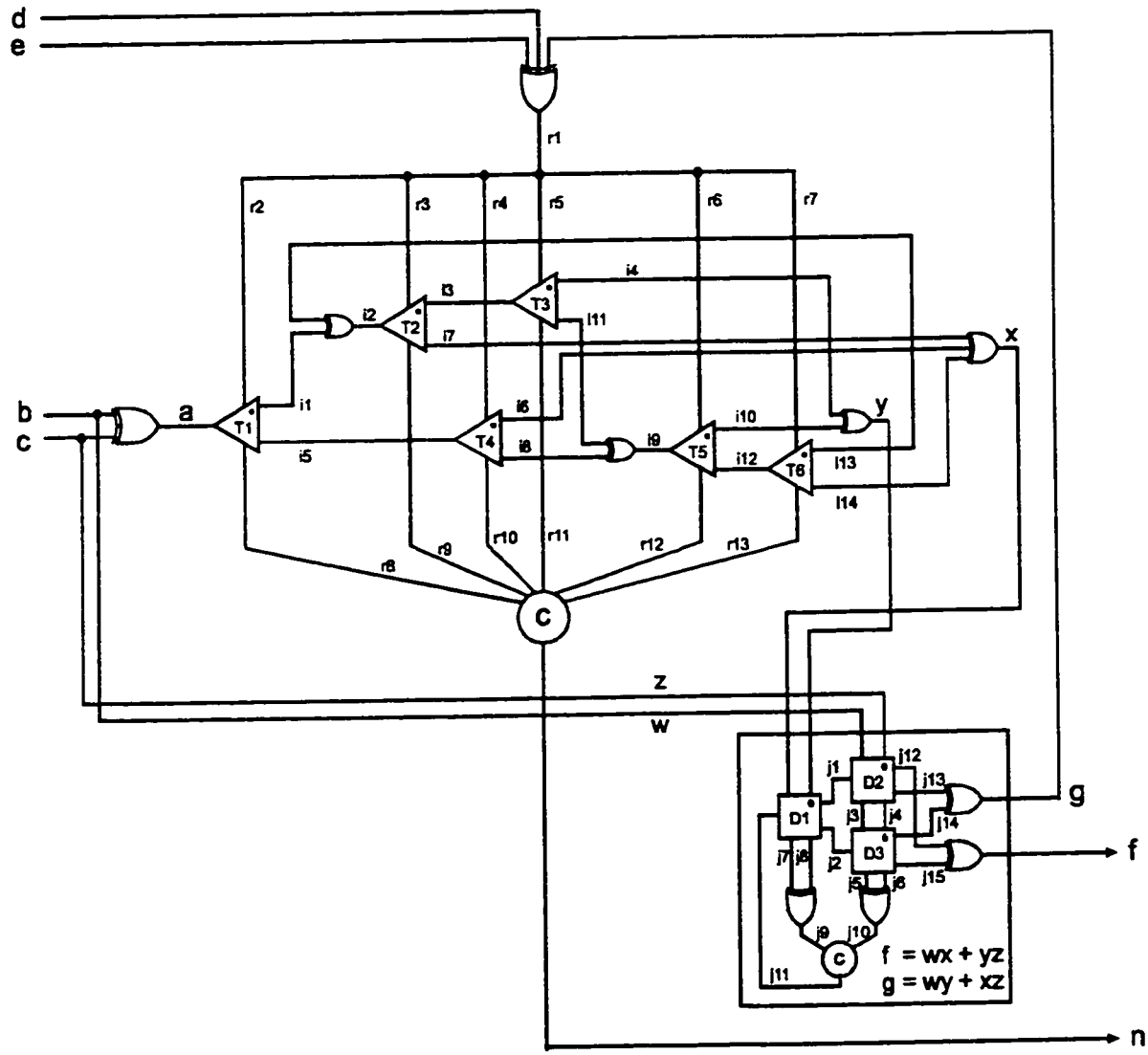


Fig. 7.2: Sequence checker for sequence (cbbcbbccbbbcbbcbcbbb)*.

initialization.

Fig. 7.3 shows a possible at-least-2 test which cannot be shortened any further. It illustrates control point insertion using *GAP_SIZE=15*, *MIN_GAP_SIZE=10*, and *K=5* control points. To keep track of the internal states of the toggles and demultiplexers, the behavior is labeled with state assignments indicating when a transition changes the internal state of a component. Backward gap extraction is used here, where we find the largest size gaps starting from the end of the pomset, a simple alteration of the forward gap extraction algorithm shown in Chapter 6. Backward gap extraction appears to give good results for deterministic circuits containing mostly toggles because the initial non-gap behavior is guaranteed to execute correctly due to the same initial state, and less gap matching is required. This is not as important for circuits which contain a lot of input choice and demultiplexers such as the next example. The square brackets indicate the actual extracted gaps with matching states and the circled events indicate control points inserted at nodes {*w*, *i3*, *i12*, *j1*, *j2*}. A sequential tour of gaps is used. We have used sequential tours because of the difficulty of keeping track of many states by hand and because sequential tours have given good results.

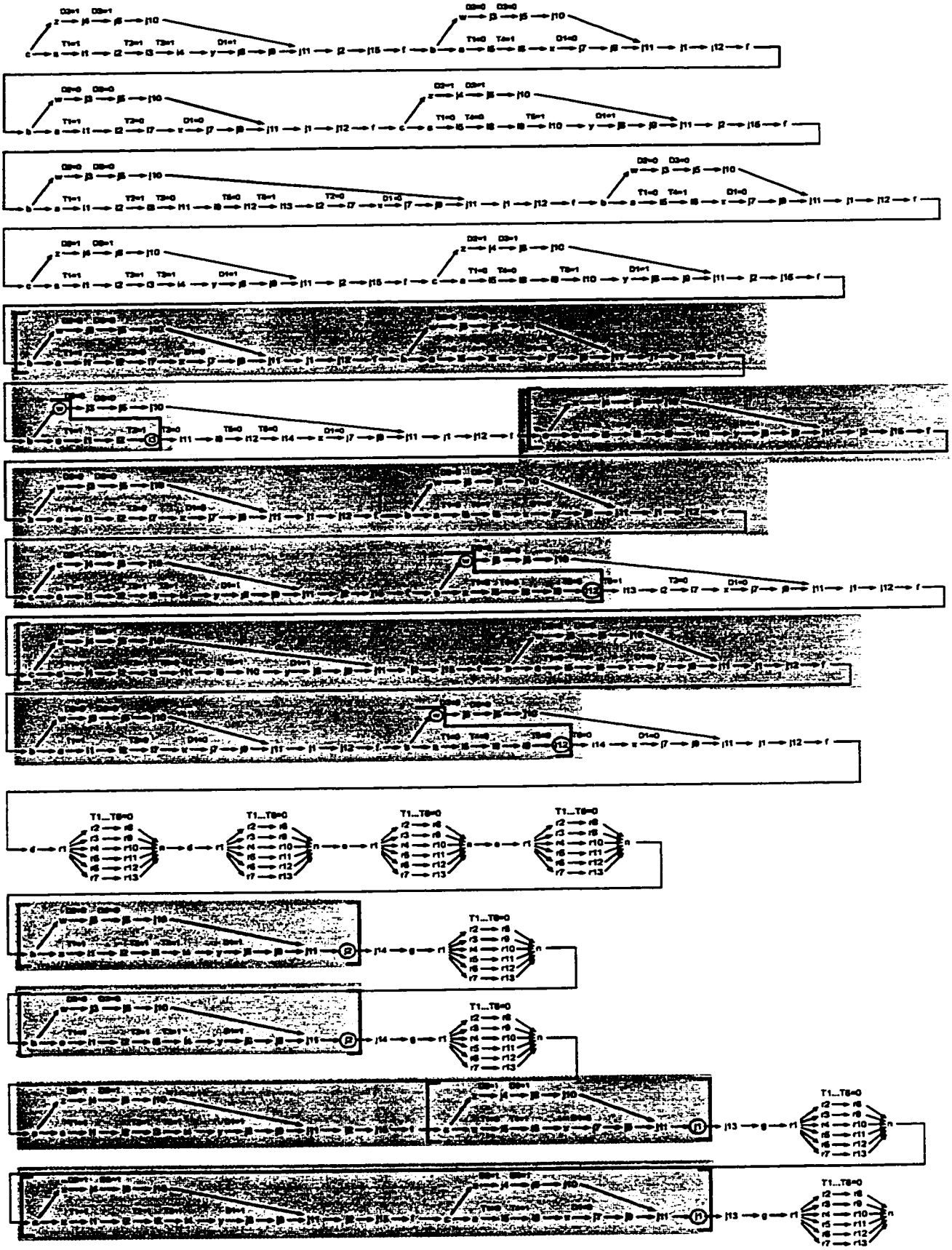


Fig. 7.3: Sequence checker behavior.

7.2 Example #2 (control point insertion)

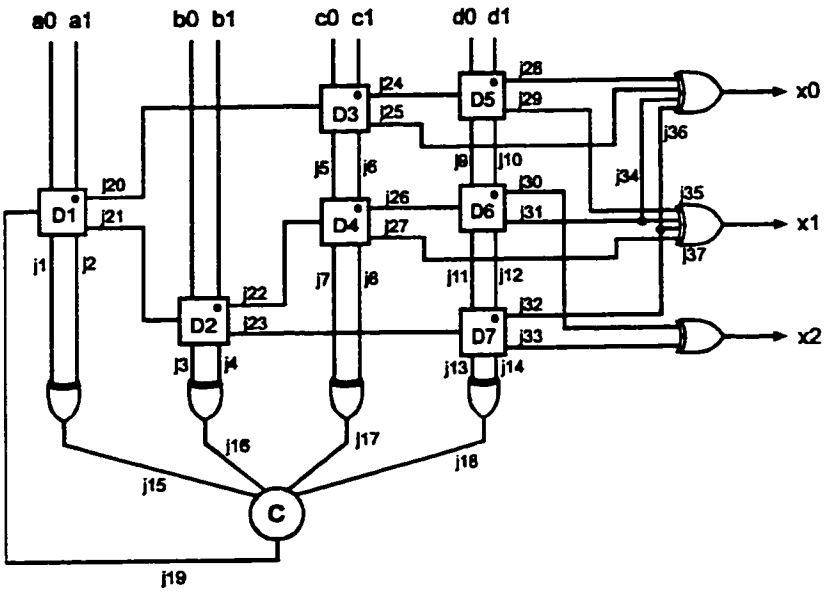


Fig. 7.4: Combinational function.

Fig. 7.4 shows a DI implementation of a three output combinational function. It consists of a tree of demultiplexers and a c-element to synchronize the inputs {*a*, *b*, *c*, *d*}. Optimization is performed by deleting demultiplexers in the tree according to the output function required. This example illustrates the effectiveness of the control point insertion algorithms when the circuit contains a lot of input choice.

Fig. 7.5 shows a possible at-least-2 test where we use the simplest method of taking all combinations of inputs and repeating the behavior twice. A shorter initial at-least-2 test can be obtained if we are given the actual function, but this may not be known in general when the choices are embedded within a larger circuit. Using forward gap extraction, *GAP_SIZE=15*, *MIN_GAP_SIZE=10*, *K=0* control points, and a sequential tour, the test is reduced from *654* events to *332* events. In this example, the choice of tour is not important because the inputs set the states of the demultiplexers through the control before sending a transition through the data lines. Inserting control points can reduce the test length here, but not by a significant amount; better improvements would be seen when this circuit is

embedded within a larger one and it is more difficult to control the exercise of certain choices to the function. Generally, the larger the circuit, the more difficult it is to control internal nodes and the better the test length reduction will be using control points.

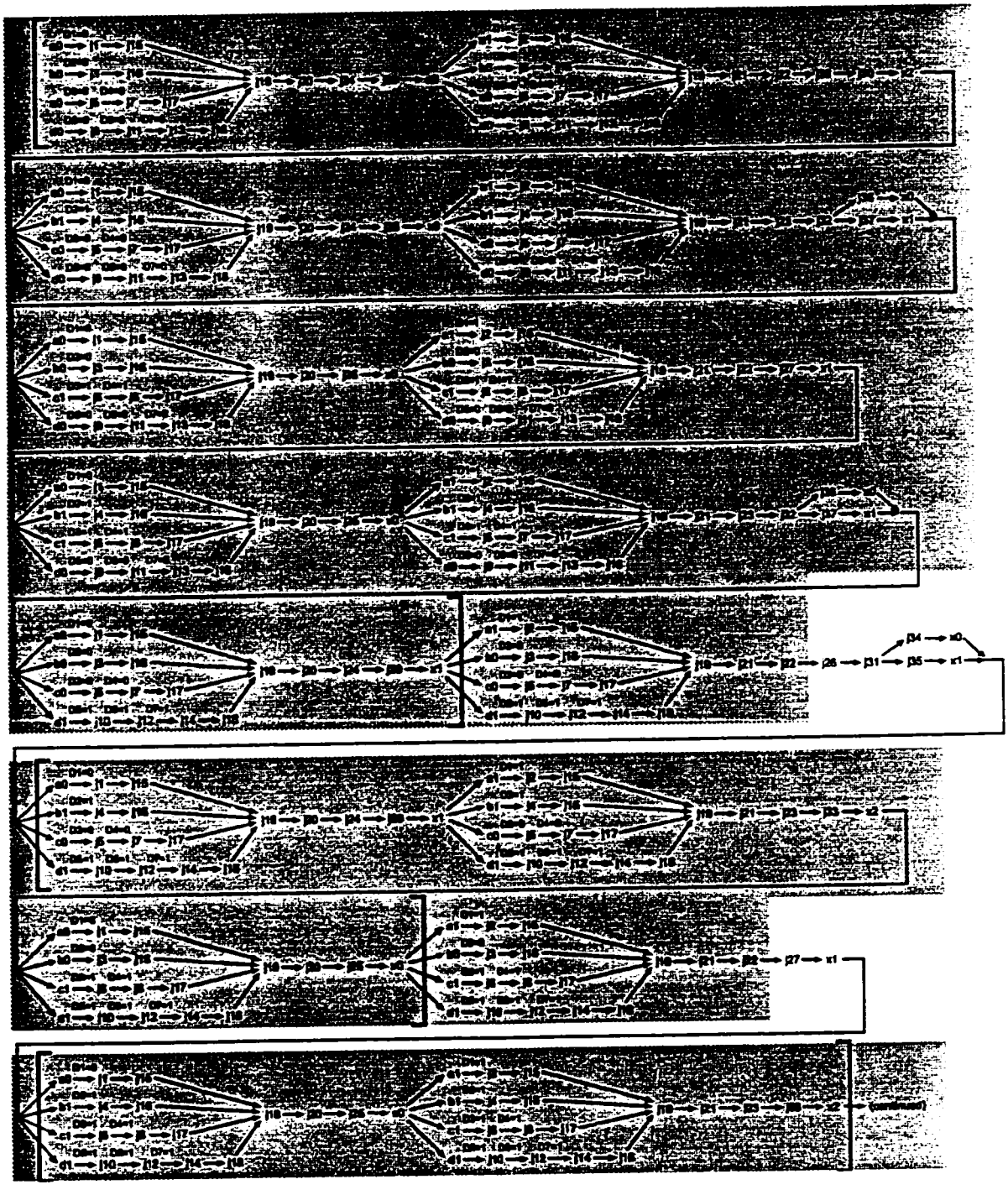


Fig. 7.5: Combinational function behavior (continued).

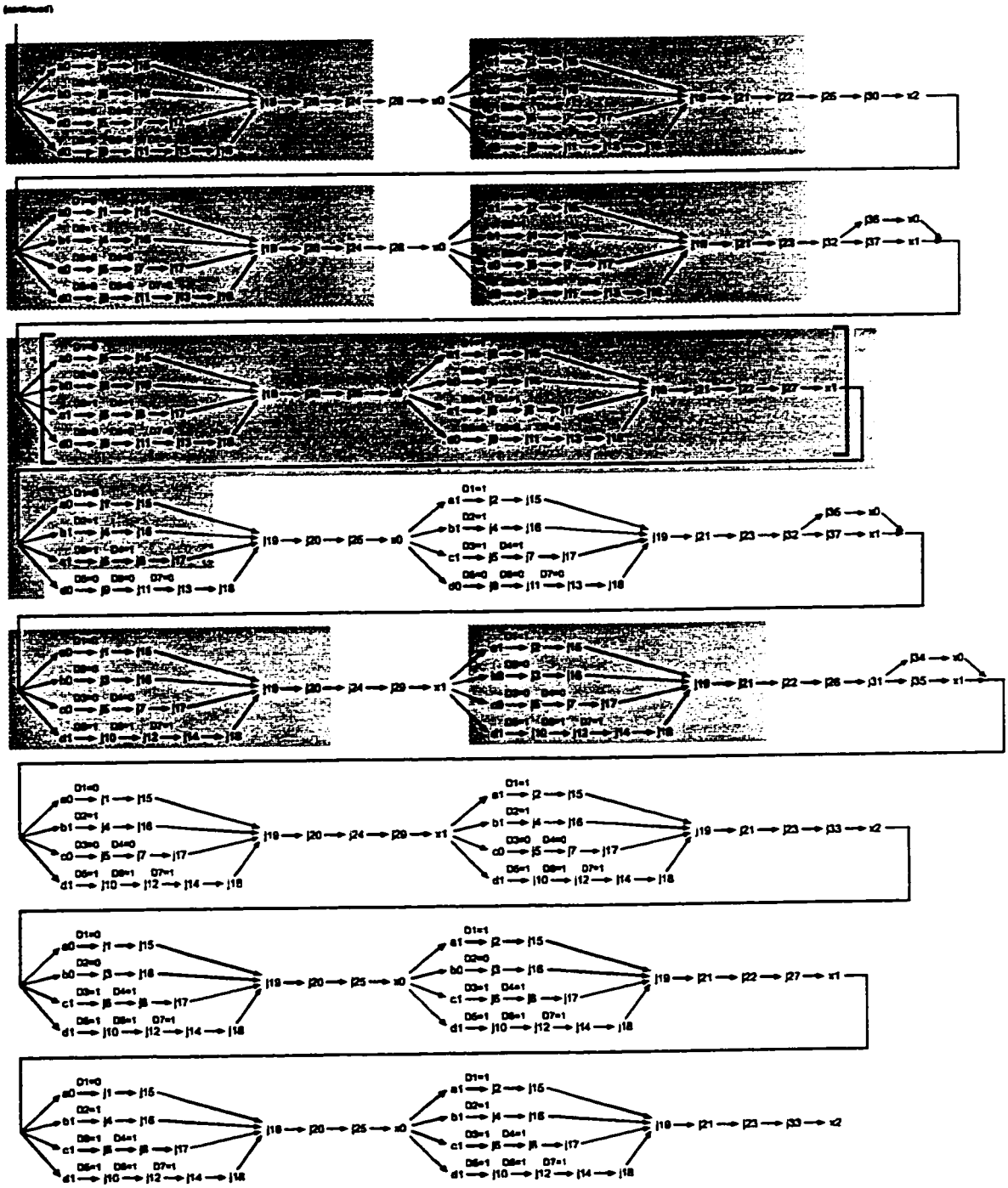


Fig. 7.5: Combinational function behavior.

7.3 Example #3 (control point insertion)

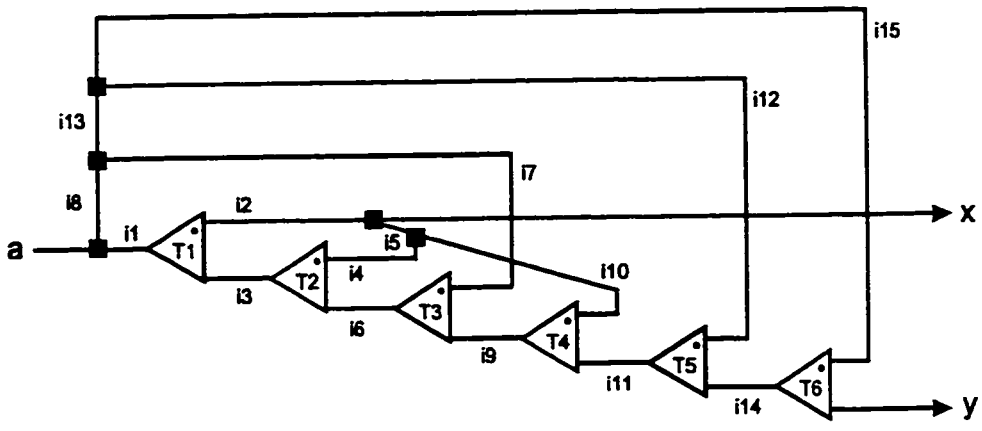


Fig. 7.6: Mod-53 counter.

This example runs the control point insertion algorithms on a mod-n counter with odd count and feedback loops and compares it with what we expect to be the ideal case of inserting control points in the middle of the counter. The algorithm is shown in Fig. 7.7 and uses backward gap extraction, *GAP_SIZE=15*, *MIN_GAP_SIZE=10*, *K=2* control points, and a sequential tour. The test is reduced from *642* events to *222* events. Using *K=1* control points inserted at node *i14* would reduce the test length by almost half. We have assumed that this circuit is composed with circuits containing c-elements so that an at-least-2 test is required rather than an at-least-1 test which would have been sufficient for this non-terminal circuit.

The algorithm does not insert control points at the best locations, ie. at node *i9* when *K=1* and *i6* and *i11* for *K=2*, but the resulting test length reduction is not that much worse. One of the reasons is that there are feedback loops sending transitions from the second half of the counter to the first half so that the two halves of the counter cannot be executed independently of one another. Execution of the second half of the counter using the algorithm requires that the partial state contain components following the feedback, such as toggle T1, which makes the

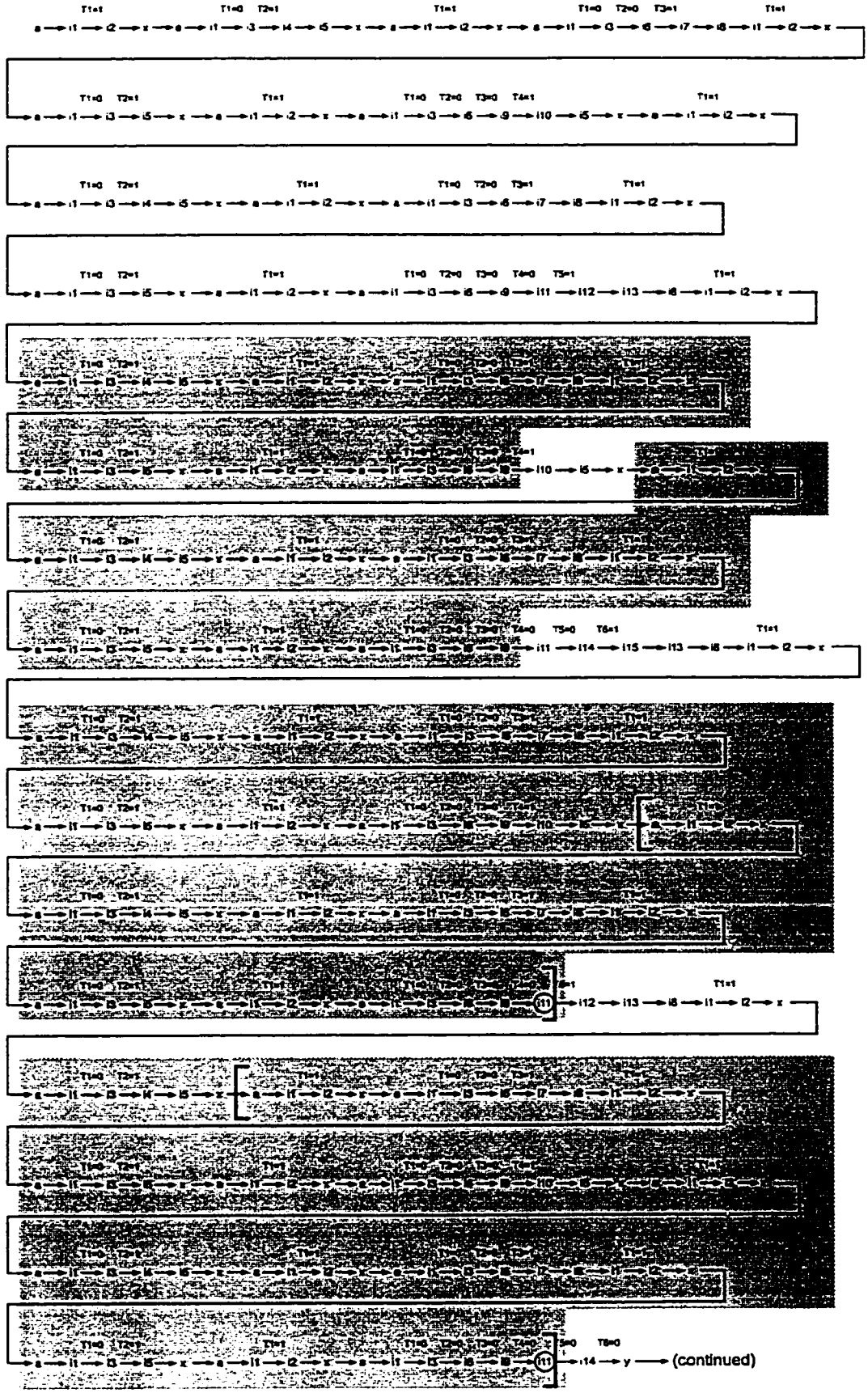


Fig. 7.7: Mod-53 counter behavior (continued).

(continued)

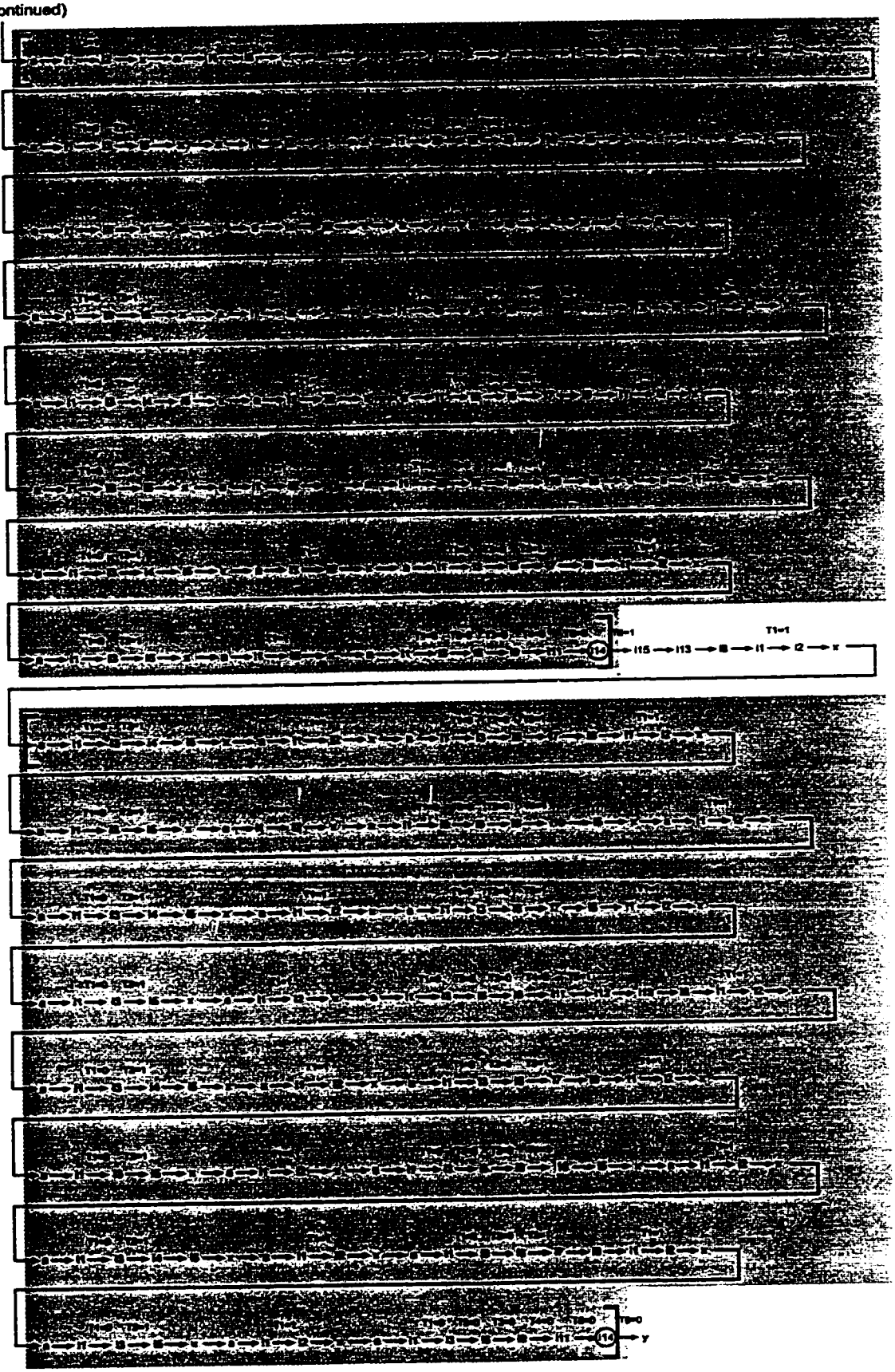


Fig. 7.7: Mod-53 counter behavior.

matching of states more difficult. The ideal case can just send transitions to the second half of the counter and ignore the switching of components following the feedback, since we know that the transition must eventually reach the output due to the non-existence of terminals. But for more general circuits with feedback, it would not be obvious how to obtain a correct protocol without using the algorithms.

7.4 Example #4 (sensitization)

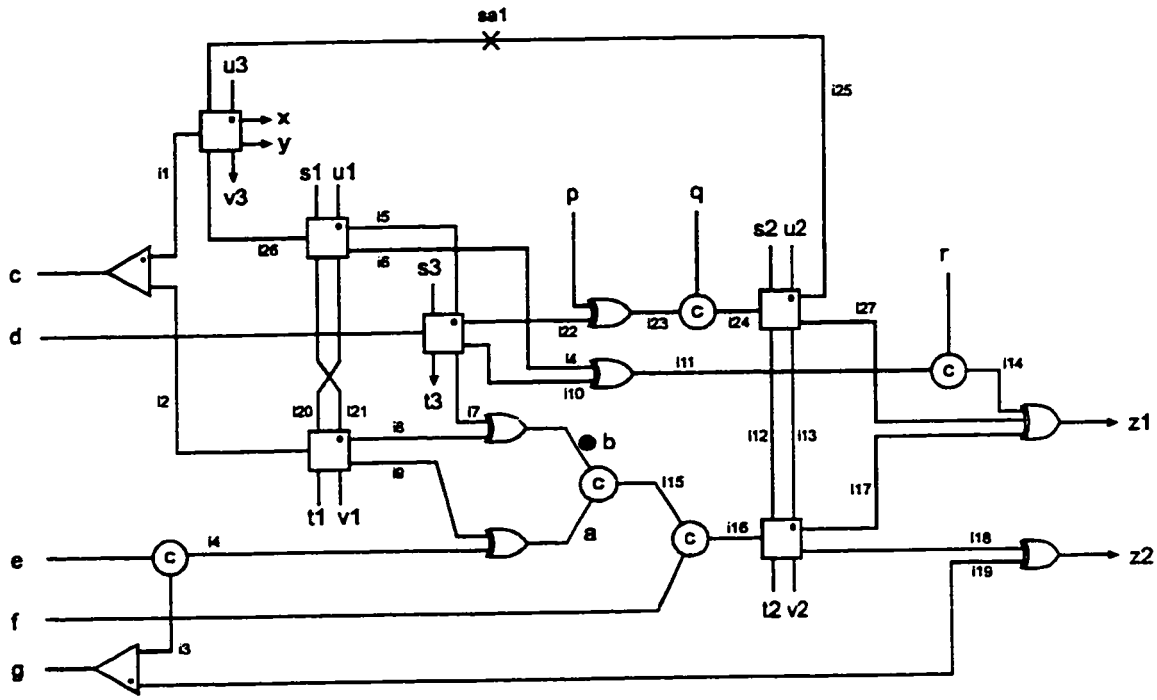


Fig. 7.8: Circuit to be sensitized.

Fig. 7.8 shows a circuit in which the at-least-2 test of Fig. 7.9 is not sufficient to detect the saf. The spurious token at initialization traverses through some demultiplexers and stops at input *b* of the c-element. The demultiplexer with input *d* is the one which is incorrectly set and which causes redirection of tokens around the sa1.

When the sensitization algorithm is run on the pomset behavior, the prefix *p*

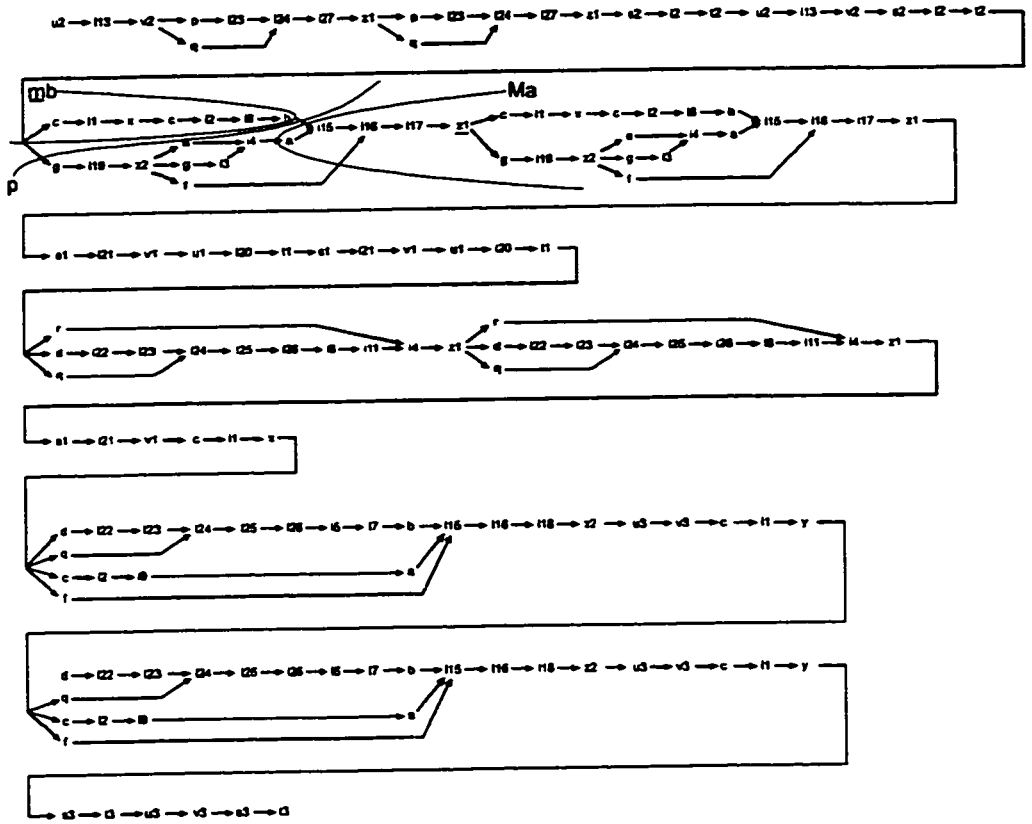


Fig. 7.9: Sensitization behavior.

to be searched for is advanced to the first join (C-element) containing event *b* since the spurious token to be sensitized is located at *b*. The prefix *p* is then set to the minimum prefix **mb** and then it searches the region from **mb** to **Ma**. The sensitization algorithm will detect the first partial state which can lead to the production of some interface output, which will be the second occurrence of output *z1* as underlined in the figure. In this example, it is the prefix *p* as shown in the figure. The sensitization region is the pomset segment from *p* to *z1*. It can be checked that the partial state at *p* with respect to the actions in the sensitization region matches with the state at circuit initialization. The actual test would be {*g*}; {*z2*}; {*e,f,g*}; {*z1*}.

Chapter 8

Conclusion

8.1 Summary

This thesis introduces a clear fault model for transition signaling circuits and reveals the problems caused by races/hazards during testing. Unlike previous work, we address the testing of a more general class of circuit which include multiple output components. We show that the common notion of "exercising every node twice" is not always sufficient to detect every saf in a DI circuit. An impossibility result is proved for fault tolerant DI circuits. A design for test method using observation points is shown which guarantees 100% fault coverage. Structural theorems are introduced which can reduce the need for observation points, or completely eliminate them for a large class of circuit consisting of CRF components. Sensitization techniques are shown for improving the fault coverage of circuits without design for test. A new use of control points is presented for test length reduction. To avoid the extremely difficult problem of obtaining a race/hazard-free test on an altered circuit, a new technique based on protocol extraction and partial

states is used.

To summarize, a test can involve the following sequences:

Without design for test:

(i) Use structural (behavioral) theorems to check whether every node is covered by an at-least-2 test.

(ii) Use sensitization test to check for spurious tokens at terminal components for remaining nodes which are not covered.

(iii) The remaining nodes are not guaranteed to be detected.

Alternatively, behavioral race simulation can be used to check the coverage of a node, but this would be more expensive. It would not replace sensitization which can detect saf's which are not detectable with the given at-least-2 test.

With design for test:

(i) Use structural (behavioral) theorems to check whether every node is covered by an at-least-2 test.

(ii) Optional: use sensitization test to reduce number of observation points required.

(iii) Observation points are inserted at terminal components which are reachable by the uncovered nodes to guarantee 100% fault coverage.

Race simulation can be used to determine the exact requirement of observation points but at high cost in time.

8.2 Future Research

The control point insertion algorithms can be refined and improved. The given gap finding algorithm only searches for the largest size gaps starting from the beginning (or end) of the pomset. Since the present gap finding algorithm is relatively cheap to perform, it is possible to explore other gap finding strategies, such as extracting the largest size gaps starting from some middle location or some

random location. The resulting insertion points obtained can be quite different and a larger search space for optimization is possible. The selection of control point to insert can be made more intelligently by sharing/reusing control points for different gap matchings. The present gap matching algorithm only makes use of local information within a gap and inserts the control point at the farthest right of the gap to maximize gap size. Selecting an internal node preceding this farthest right location can increase the reusability of control points, such as by keeping a record of control points already used by other gaps, and so the total number of control points can be minimized. When the size of a gap is reduced and fixed in gap matching, it is possible to increase the size of the other uncommitted gaps as a result of the increased events traversed during gap fixing. The tour finding algorithm can also be improved using branch and bound or some randomizing technique.

It is possible to use invasive control points in addition to the XOR control points to improve the results. XOR control points require that a transition sent into a XOR propagate through the circuit until stable state. Insertion of an invasive control point (a disconnect) following the XOR control point can decrease the distance that the input transition travels, decreasing the size of the protocol which must be traversed, and increasing the possibility of matching states. This allows the state of a single component (or a few components) to change states by inserting a XOR control point preceding a component and an invasive control point immediately following the component. The requirement that the start of a gap be a stable state in the pomset is relaxed, since the invasive control point is effectively converting an internal node into a primary output and creating a new stable state.

It is possible to perform the searches in sensitization and control point insertion using the pomset tree directly instead of using a particular linear behavior. There is a potential for improved results, but at the expense of a more complicated

search due to the choice of behaviors to execute. The linear at-least-2 test contains much of the behavior in the pomset tree and the out of order tours simulate this behavior to some extent. The additional information of knowing exactly where the choice points are may lead to better algorithms.

Deserialization is a concept which can (i) compact a test, and (ii) permit sensitization which otherwise could not be used. It eliminates artificial serialization in an at-least-2 test. For example, we are given two parallel wires with input-output of a-x and b-y. If the protocol specifies the usage as only $(a;x;b;y)^*$, then the at-least-2 test would be serial. However, we know that the test can be concurrent, $(\{a,b\};\ \{x,y\})^2$. By increasing concurrency, the total length of the test can be shortened. Deserialization can be useful in sensitization by allowing certain events to be eliminated from the sensitization region.

Additional structural/behavioral theorems can also be explored, which can further reduce the number of observation points required. The present theorems determine when a saf at a node is guaranteed to be covered by an at-least-2 test. The converse, theorems guaranteeing when a node is ***not*** covered by an at-least-2 test is much more difficult, although maybe not impossible. This is because the theorems must take into account possible critical races involving CR components, rather than trying to partition the circuit to avoid these CR components. Checking non-coverage may well require race simulation.

Future work on SI circuits is possible using the models in this thesis. The Petri net already models an isochronic fork because the firing of a fork transition produces output tokens at the same time. The difference between a DI and an SI Petri net is that DI nets allow wire transitions to follow the fork transition, which allows an arbitrary difference in delay between fork branches. It will be worthwhile to explore how the results in this thesis will apply when this subtle but significant

change is made to the model. The results may also be dependent on the type of SI circuit used. For example, AND/OR gates are terminal components because certain input transitions may not produce an output. If the SI circuit contains many AND/OR gates, then the insertion of observation points at the inputs of these gates would not be practical. The results of this thesis can be applicable to SI circuits which use transition signaling, since DI is a subset of SI.

References

[1] V.K. Agarwal and A.S.F. Fung, "Multiple Fault Testing of Large Circuits by Single Fault Test Sets", *IEEE Trans. Computers*, vol.C-30, no.11, 1981, pp. 855.

[2] J.D. Allen, P.T. Gaughan, D.E. Schimmel, and S. Yalamanchili, "Ariadne - An Adaptive Router for Fault-Tolerant Multicomputers", *Proc. Int'l Symposium on Computer Architecture*, 1994, pp. 278-288.

[3] P.A. Beerel and T.H.-Y. Meng, "Semi-Modularity and Testability of Speed-Independent Circuits", *Integration, VLSI Journal*, 1992, pp. 301-322.

[4] P.A. Beerel and T.H.-Y. Meng, "Semi-Modularity and Self-Diagnostic Asynchronous Control Circuits", *Advanced Research in VLSI*, UC Santa Cruz, 1991, pp. 103-117.

[5] P.A. Beerel and T.H.-Y. Meng, "Testability of Asynchronous Timed Control Circuits with Delay Assumptions", *Design Automation Conference*, 1991, pp. 446-451.

[6] K. van Berkel, R. Burgess, J.L.W. Kessels, A. Peeters, M. Roncken, and F. Schalij, "A Fully Asynchronous Low-Power Error Corrector for the DCC Player", *IEEE J. of Solid-State Circuits*, vol.29, no. 12, 1994, pp. 1429-1439.

[7] K. van Berkel, R. Burgess, J. Kessels, M. Roncken, F. Schalij, and A. Peeters, "Asynchronous Circuits for Low Power: A DCC Error Corrector", *IEEE Design & Test of Computers*, Summer 1994, pp. 22-32.

[8] C.H. van Berkel, and R. Saeijs, "Compilation of Communicating Processes into Delay-Insensitive Circuits", *Proc. Int'l Conf. Computer Design*, 1988, pp. 157-162.

[9] E. Best, "COSY: Its Relation to Nets and to CSP", *Petri Nets: Applications and Relationships to Other Models of Concurrency*, LNCS 255, 1987, pp. 416-440.

[10] G. Boudol and I. Castellani, "Concurrency and Atomicity", *Theoretical Computer Science*, 59, 1988, pp. 25-84.

[11] E. Brunvand and R.F. Sproull, "Translating Concurrent Programs into Delay-Insensitive Circuits", *Proc. Int'l Conf. Computer Aided Design*, 1989, pp. 262-265.

[12] J.A. Brzozowski and K. Raahemifar, "Testing C Elements is not Elementary", Report, Dept. Comp. Sci. and ECE, Univ. Waterloo, 1995.

[13] J.A. Brzozowski and J.C. Ebergen, "On the Delay-Sensitivity of Gate Networks", *IEEE Trans. Computers*, Vol. 41, No. 11, 1992.

[14] G. Carson and G. Borriello, "A Testable CMOS Asynchronous Counter", *J. Solid-State Circuits*, vol.25, no.4, 1990, pp. 952.

[15] A.P. Chandrakasan and R.W. Brodersen, "Minimizing Power Consumption in Digital CMOS Circuits", *Proc. of the IEEE*, Vol. 83, No. 4, April 1995, pp. 498-523.

[16] K.-T. Cheng, V.D. Agrawal, and E.S. Kuh, "A Simulation-Based Method for Generating Tests for Sequential Circuits", *IEEE Trans. Computers*, vol.39, no.12, 1990, pp.1456.

[17] T.-A. Chu, "Automatic Synthesis and Verification of Hazard-Free Control Circuits from Asynchronous Finite State Machine Specifications", *Proc. Int'l Conf. Computer Design*, 1992, pp. 407-413.

[18] T.-A. Chu, "Synthesis of Self-timed VLSI Circuits from Graph-theoretic Specifications", *Proc. Int'l Conf. Computer Design*, 1987, pp. 220-223.

[19] I. David, R. Ginosar, and M. Yoeli, "Implementing Sequential Machines as Self-Timed Circuits", *IEEE Trans. Computers*, Vol. 41, No. 1, 1992, pp. 12-17.

[20] P. Degano and S. Marchetti, "Partial Ordering Models for Concurrency Can Be Defined Operationally", *Int'l Journal of Parallel Programming*, Vol. 16, No. 6, 1987, pp. 451-478.

[21] S. Devadas, et. al., "A Synthesis and Optimization Procedure for Fully and Easily Testable Sequential Machines", *IEEE Trans. Computers*, vol.8, no.10, 1989, pp. 1100-1107.

[22] J.C. Ebergen, "A Formal Approach to Designing Delay-Insensitive Circuits", *J. Distributed Computing*, Vol. 5, 1991, pp. 107-119.

[23] J.C. Ebergen, *Translating Programs into Delay-Insensitive Circuits*, Ph.D. Thesis, Eindhoven University of Technology, 1987.

[24] J.A. Fifield and C.H. Stapper, "High-Speed On-Chip ECC for Synergistic Fault-Tolerance Memory Chips", *IEEE J. of Solid-State Circuits*, vol.26, no. 10, 1991, pp. 1449-1452.

[25] M.A. Franklin and T. Pan, "Performance Comparison of Asynchronous Adders", *Proc. Int'l Symposium on Advanced Research in Asynchronous Circuits and Systems*, Salt Lake City, Utah, 1994, pp. 117-125.

[26] H. Fujiwara and S. Toida, "The Complexity of Fault Detection Problems for Combinational Logic Circuits", *IEEE Trans. Computers*, vol.C-31, no.6, 1982, pp. 555.

[27] J.L. Gischer, "The Equational Theory of Pomsets", *Theoretical Computer Science*, 61, 1988, pp. 199-224.

[28] R. van Glabbeek and U. Goltz, "Equivalence Notions for Concurrent Systems and Refinement of Actions", *Mathematical Foundations of Computer Science*, LNCS 379, 1989, pp. 237-248.

[29] S.S. Guillory, D.G. Saab, and A. Yang, "Fault Modeling and Testing of Self-timed Circuits", *Proc. VLSI Test Symposium*, 1991, pp. 62-66.

[30] S. Hauck, "Asynchronous Design Methodologies: An Overview", *Proc. of the IEEE*, vol.83, no.1, 1995, pp. 69-93.

[31] P.J. Hazewindus, *Testing Delay-Insensitive Circuits*, PhD Thesis, California Institute of Technology, 1992.

[32] B.J. Heard, et.al. "Automatic Test Pattern Generation for Asynchronous Networks", *Proc. Int'l Test Conf.*, 1984, pp. 63-69.

[33] H. Hulgaard, S.M. Burns, and G. Borriello, "Testing Asynchronous Circuits: A Survey", Technical Report FR-35, Dept. Comp. Sci. and Eng., Univ. Washington, 1994.

[34] O.H. Ibarra and S.K. Sahni, "Polynomially Complete Fault Detection Problems", *IEEE Trans. Computers*, vol.C-24, no.3, 1975, pp. 242.

[35] J. Jacob and N.N. Biswas, "GTBD Faults and Lower Bounds on Multiple Fault Coverage of Single Fault Test Sets", *Proc. Int'l Test Conference*, 1987, pp. 849.

[36] K. Keutzer, L. Lavagno, and A. Sangiovanni-Vincentelli, "Synthesis for Testability Techniques for Asynchronous Circuits", *IEEE Trans. Computer-Aided Design*, Vol. 14, No. 12, Dec. 1995, pp. 1569-1577.

[37] K. Keutzer, L. Lavagno, and A. Sangiovanni-Vincentelli, "Synthesis for Testability Techniques for Asynchronous Circuits", *Proc. Int'l Conf. Computer Aided Design*, 1991, pp. 326-329.

[38] A. Khoche and E. Brunvand, "Testing Micropipelines", *Proc. Int'l Symposium on Advanced Research in Asynchronous Circuits and Systems*, Salt Lake City, Utah, 1994, pp. 239-246.

[39] L. Kleeman and A. Cantoni, "Metastable Behavior in Digital Systems", *IEEE Design & Test of Computers*, Dec. 1987, pp. 5-19.

[40] I. Kohavi, "Fault Diagnosis of Logical Circuits", *Tenth Annual Symposium on Switching and Automata Theory*, 1969, pp. 166.

[41] P.N. Lam, H.F. Li, and S.C. Leung, "Optimization of State Encoding in Distributed Circuits", *IEEE Trans. Computer-Aided Design*, Vol. 13, No. 5, May 1994, pp. 581-588.

[42] L. Lavagno, M. Kishinevsky, and A. Lioy, "Testing Redundant Asynchronous Circuits by Variable Phase Splitting", *Proc. European Design Automation Conf.*, 1994, pp. 328-333.

[43] L. Lavagno, K. Keutzer, and A.L. Sangiovanni-Vincentelli, "Synthesis of Verifiably Hazard-Free Asynchronous Control Circuits", *Advanced Research in VLSI*, UC Santa Cruz, 1991, pp. 87-102.

[44] J. Leenstra and L. Spaanenburg, "On the Design and Test of Asynchronous Macros Embedded in Synchronous Systems", *Proc. Int'l Test Conference*, 1989, pp. 838-845.

[45] S.C. Leung and H.F. Li, "On the Realizability and Synthesis of Delay-Insensitive Behaviors", *IEEE Trans. Computer-Aided Design*, Vol. 14, No. 7, 1995, pp. 833-848.

[46] S.C. Leung and H.F. Li, "A Syntax-Directed Translation for the Synthesis of Delay-Insensitive Circuits", *IEEE Trans. on VLSI Systems*, Vol. 2, No. 2, June 1994, pp. 196-210.

[47] S.C. Leung, *Synthesis of Delay-Insensitive Circuits from Graph Theoretic Specifications*, Ph.D. Thesis, Concordia University, 1993.

[48] T. Li and L.A. Hollaar, "On the Testability of the Direct Implementation of Asynchronous Circuits", *Advanced Research in VLSI*, M.I.T., 1984, pp.117.

[49] H.F. Li and P.N. Lam, "A Protocol Extraction Strategy for Control Point Insertion in Design for Test of Transition Signaling Circuits", *Great Lakes Symposium on VLSI*, 1995, pp. 178-183.

[50] H.F. Li and P.N. Lam, "Petri Net Fault Model and Testing of Delay-Insensitive Circuits", *Canadian Conf. VLSI*, 1993, pp. 4B-15 to 4B-20.

[51] H.F. Li, S.C. Leung, and P.N. Lam, "Synthesis of Delay-Insensitive Control Circuits by Refinement into Atomic Threads", *Int'l Conf. Computer Design*, 1991, pp. 180-186.

[52] A.J. Martin and P.J. Hazewindus, "Testing Delay-Insensitive Circuits", *Advanced Research in VLSI*, UC Santa Cruz, 1991, pp. 118-131.

[53] A.J. Martin, "The Limitations to Delay-Insensitivity in Asynchronous Circuits", *Advanced Research in VLSI*, MIT, 1990, pp. 263-278.

[54] A.J. Martin, "The Design of a Delay-Insensitive Microprocessor: An Example of Circuit Synthesis by Program Transformation", *Hardware Specification, Verification, and Synthesis: Mathematical Aspects*, LNCS 408, 1990, pp. 244-259.

[55] K.L. McMillan, "Using unfoldings to avoid the state explosion problem in the verification of asynchronous circuits", *Fourth Workshop on Computer Aided Verification*, 1992, pp. 164-174.

[56] T.H.-Y. Meng, R.W. Brodersen, and D.G. Messerschmitt, "Automatic Synthesis of Asynchronous Circuits from High-Level Specifications", *IEEE Trans. Computer-Aided Design*, Vol. 8, No. 11, 1989, pp. 1185-1205.

[57] T. Murata, "Petri Nets: Properties, Analysis and Applications", *Proc. of the IEEE*, vol.77, no.4, April 1989, pp. 541-580.

[58] O.A. Petlin and S.B. Furber, "Scan Testing of Asynchronous Sequential Circuits", *Great Lakes Symposium on VLSI*, 1995, pp. 224-229.

[59] O.A. Petlin, S.B. Furber, A.M. Romankevich, and V.V. Groll, "Designing Asynchronous Sequential Circuits for Random Pattern Testability", *IEE Proceedings, Pt. E.*, vol.142, no.4, July 1995, pp. 299-305.

[60] V.R. Pratt, "Modeling Concurrency with Partial Orders", *Int'l Journal of Parallel Programming*, Vol. 15, No. 1, 1986, pp. 33-71.

[61] D.K. Probst and H.F. Li, "Verifying Timed Behavior Automata with Nonbinary Delay Constraints", *Computer Aided Verification*, LNCS 663, 1992, pp. 121-134.

[62] D.K. Probst and H.F. Li, "Partial-Order Model Checking: A Guide for the Perplexed", *Computer Aided Verification*, LNCS 575, 1991, pp. 322-331.

[63] D.K. Probst and H.F. Li, "Modeling Reactive Hardware Processes using Partial Orders", *Semantics of Concurrency, Workshops in Computing*, July 1990, pp. 324-343.

[64] D.K. Probst and H.F. Li, "Using Partial-Order Semantics to Avoid the State Explosion Problem in Asynchronous Systems", *Computer Aided Verification*, LNCS 531, 1990, pp. 146-155.

[65] G.R. Putzolu and J.P. Roth, "A Heuristic Algorithm for the Testing of Asynchronous Circuits", *IEEE Trans. Computers*, vol.C-20, no.6, 1971, pp.639.

[66] W. Reisig, "On the Semantics of Petri Nets", *Formal Models in Programming*, 1985, pp. 347-372.

[67] M. Rem, J. van de Snepscheut, and J.T. Udding, "Trace Theory and the Definition of Hierarchical Components", *Third Caltech Conference on VLSI*, 1983, pp. 225-239.

[68] D.A. Rennels and H. Kim, "Concurrent Error Detection in Self-Timed VLSI", *Int'l Symposium on Fault-Tolerant Computing*, 1994, pp. 96-105.

[69] M. Roncken, "Partial Scan Test for Asynchronous Circuits Illustrated on a DCC Error Corrector", *Proc. Int'l Symposium on Advanced Research in Asynchronous Circuits and Systems*, Salt Lake City, Utah, 1994, pp. 247-256.

[70] K. Sakashita, T. Hashizume, T. Ohya, I. Takimoto, and S. Kato, "Cell-Based Test Design Method", *Proc. Int'l Test Conference*, 1989, pp. 909-916.

[71] J. Saxena and D.K. Pradhan, "Design for Testability of Asynchronous Sequential Circuits", *Proc. Int'l Test Conference*, 1993, pp. 518-522.

[72] D.R. Schertz and G. Metze, "On the Design of Multiple Fault Diagnosable Networks", *IEEE Trans. Computers*, vol.C-20, Nov. 1971, pp. 1361.

[73] C.-J. Seger, "On the Existence of Speed-Independent Circuits", *Theoretical Computer Science*, vol.86, 1991, pp. 343-364.

[74] C.-J. Seger, "The Complexity of Race Detection in VLSI Circuits", *Decennial Caltech Conference on VLSI*, 1989, pp. 335-350.

[75] C.L. Seitz, "System Timing", in Mead & Conway, *Introduction to VLSI Systems*, Chapter 7, Addison-Wesley, 1980.

[76] M.-D. Shieh, C.-L. Wey, and P.D. Fisher, "A Scan Design for Asynchronous Sequential Logic Circuits Using SR-Latches", *Proc. Midwest Symposium on Circuits and Systems*, 1993, pp. 1300-1303.

[77] L. Spaanenburg, J.Smit, and H.van der Veen, "A Methodology for the Fast and Testable Implementation of State Diagram Specifications", *IEEE J. of Solid-State Circuits*, vol.SC-20, no.2, 1985, pp. 548.

[78] A.K. Susskind, "A Technique for Making Asynchronous Sequential Circuits Readily Testable", *Int'l Test Conf.*, 1984, pp. 842.

[79] I.E. Sutherland, "Micropipelines", *Communications ACM*, Vol. 32, No. 6, June 1989, pp. 720-738.

[80] T. Svedek and V. Ivancic, "ASIC test sequence minimisation by built-in testability in logic surrounding embedded binary asynchronous counters", *IEE Proceedings, Pt. E.*, vol.136, no.5, 1989, pp.450.

[81] J.T. Udding, "A Formal Model for Defining and Classifying Delay-Insensitive Circuits and Systems", *J. Distributed Computing*, 1986, pp. 197-204.

[82] C.-L. Wey, M.-D. Shieh, and P.D. Fisher, "*ASLCScan*: A Scan Design Technique for Asynchronous Sequential Logic Circuits", *Int'l Conf. Computer Design*, 1993, pp. 159-162.

[83] G. Winskel, "Event Structures", *Petri Nets: Applications and Relationships to Other Models of Concurrency*, LNCS 255, 1987, pp. 325-392.

IMAGE EVALUATION TEST TARGET (QA-3)

150mm

6"

APPLIED IMAGE . Inc
1653 East Main Street
Rochester, NY 14609 USA
Phone: 716/482-0300
Fax: 716/288-5989