Abstract—Fault tuples represent a defect modeling mechanism capable of capturing the logical misbehavior of arbitrary defects in digital circuits. To justify this claim, this paper describes two types of logic faults (state transition and signal line faults) and formally shows how fault tuples can be used to precisely represent any number of faults of this kind. The capability of fault tuples to capture misbehaviors beyond logic faults is then illustrated using many examples of varying degree of complexity. In particular, the ability of fault tuples to modulate fault controllability and observability is examined. Finally, it is described how fault tuples can and have been used to enhance testing tasks such as fault simulation, test generation, and diagnosis, and enable new capabilities such as interfault collapsing and application-based quality metrics.

Index Terms—Defect and fault modeling, defect characterization, diagnosis, fault simulation, test generation.

I. INTRODUCTION

The continued improvement in semiconductor technologies has many challenges. Many of these challenges stem from the complexity increases anticipated in nanoscale fabrication processes. One consequence of this complexity is the yield degradation expected from design–fabrication interactions [1] and yet-to-be-encountered defect types. Because our ability to “see” these failures is quickly diminishing, test will become the main source of insight. Since the test structures typically used to understand the reasons for failure will be limited, product test will be a major source of feedback. In other words, in addition to its normal sorting function, test will be an essential feedback loop for understanding the failure mechanisms inherent in complex fabrication processes. Specifically, test will be the key enabler for characterizing defects, unacceptable parametric variations, and design–process interactions essential for yielding reliable working products in both a timely and cost-effective manner.

To meet this challenge, new efficient approaches to test, testability analysis, design-for-test, built-in self-test, and diagnosis must be developed that can cope with the variety of failure types that will undoubtedly occur in future-generation products. Current test research is restricted due to the fact that both academic (e.g., [2]–[7]) and commercial tools (e.g., [8]–[10]), for the most part, are limited to just a few fault models. This means that existing tools cannot precisely analyze the effects of current and anticipated manufacturing imperfections, thus preventing the accurate evaluation and validation of new approaches to test. We believe fault tuples [11] may be one approach to remedy this situation. The fault-tuple mechanism is a general and effective methodology for analyzing failures in digital integrated circuits. Its key characteristic is its ability to precisely model the effects of any arbitrary failure source on the logical behavior of a digital circuit. Fault simulation, test generation, diagnosis, and other testing tasks can all be enhanced by using fault tuples [12]–[17].

In the rest of this paper, we define and illustrate the use of fault tuples in various test analysis tasks. Specifically, in Section II, we describe other work that has focussed on generalized fault models. In Section III, we introduce two types of logic faults and describe how fault tuples can be used to precisely and efficiently represent these faults in Section IV. Also, in Section IV, we illustrate the defect modeling capability of fault tuples using various examples. Section V describes how fault tuples: 1) enhance various test applications including fault simulation, test generation, and diagnosis and 2) enable new applications such as interfault collapsing. Finally, in Section VI, we summarize our contributions.

II. STATE OF DEFECT MODELING

Over 40 years ago, Eldred elegantly described a test set for a collapsed set of single stuck-line (SSL) faults for an electronic system consisting of diodes and vacuum tubes [18]. From that time to now, the SSL fault model [19]–[21] has become the default standard for simulating, analyzing, detecting, and diagnosing failures in integrated digital systems. Although the SSL fault will most likely continue to be utilized, there has been a long recognized need for more precise approaches [22]. The acuteness of this need has increased significantly as stated in the 2003 International Technology Roadmap for Semiconductors (ITRS) [23]. Hence, many researchers have focussed on developing new fault models for particular types of failure mechanisms. For example, [3], [24], and [25] have examined the behavior of transistor stuck-open (TSO) defects, [26]–[28] analyzed signal line bridges, and a great deal of work has been done in modeling failures due to changes in circuit delay (e.g., [29]–[32]).

One significant consequence of adopting new fault models is the rendering of existing test tools (fault simulators, test generators, etc.) obsolete. To avoid this situation, netlist modification techniques or work-around circuits have been developed to “trick” SSL-based tools into performing the fault analysis desired. For example, a four-gate circuit is introduced into a netlist to model two-line bridge faults [33]; in [34]–[37], various circuit transformations are utilized to analyze delay faults;
and in [38], researchers replace gates with multiplexers to analyze pattern faults [39].

To alleviate this tradeoff, it is desirable to have a general fault modeling mechanism and corresponding test tools that can effectively analyze arbitrary fault types. Some work toward this objective has been done. The oldest example is the D-calculation introduced by Roth [40]. The D-calculation is a modeling mechanism that uses D-cubes (i.e., collapsed truth-table entries) to model any arbitrary change in the logic function of a circuit block. Similar to the D-calculation, the input pattern (IP) fault model [38] can model any arbitrary change in the logic function of a circuit block, where a block is defined to be any combinational subcircuit described at any level of the design hierarchy. In [41], the authors describe tools that can analyze a variety of faults that include SSL, transition, TSO, and bridge faults. In addition, their test tools can handle what they call “conditional” faults. A conditional fault allows additional signal line objectives to be combined with the detection requirements of a particular fault. Most commercial tools can handle just one or two additional fault models beyond the SSL fault model. One notable exception is the test analysis tool from Cadence (formerly TestBench from IBM) that can analyze pattern faults [39]. A pattern fault allows an arbitrary set of signal lines to be grouped into activation conditions for a single fault. Most commercial tools can handle what they call “conditional” faults. A conditional fault allows additional signal line objectives to be combined with the detection requirements of a particular fault. Most commercial tools can handle just one or two additional fault models beyond the SSL fault model. One notable exception is the test analysis tool from Cadence (formerly TestBench from IBM) that can analyze pattern faults [39]. A pattern fault allows an arbitrary set of signal lines to be grouped into activation conditions for a single fault site, allowing a variety of fault types to be modeled. Pattern faults can be either static or dynamic. In a dynamic fault, an additional set of initial values is specified for the required signal line values. Finally, pattern faults themselves can be “ANDed” or “ORed” together to further guide analysis.

Other related work has been done after the introduction of fault tuples. In [42], the authors describe a fault simulator that can directly simulate any single gate-level IP fault [38]. Their simulator is demonstrated through grading experiments that involve both SSL and bridge faults involving inputs of a single gate. The general fault model from Intel [43] has capabilities that are similar to fault tuples. Specifically, they have constructs called atoms, excitation conditions, and impact conditions that are analogous to fault-tuple products, condition tuples, and error tuples, respectively. The excitation conditions, which can be either mandatory or optional, specify the requirements for circuit lines. An impact condition specifies the voltage, logic value effect, or amount of transition delay on a signal line when mandatory excitation conditions are met. Unlike fault tuples, however, the timing properties of faults are limited to two cycles and specific clock cycles cannot be referenced. The priority, optionality, and delay parameter features of the general fault model are not intrinsic to fault tuples but can be easily accommodated. Specifically, in Section V-C, we show how priority and optionality can be accomplished using dominance, while delay is handled by identifying the affected partial paths.

The aforementioned approaches offer flexibility in defect modeling beyond SSL faults. However, none of them can effectively cope with multiple faults or capture sequential misbehavior due to, for example, a bridge that introduces unwanted feedback into the circuit. Fault tuples solve this shortcoming by providing the capability to model any number and any type of change in logical behavior. Also, the precision enabled by fault tuples allows the controllability, observability, and diagnosability of defects to be optimized in a very cost-effective manner.

III. LOGIC-LEVEL MODELS

We begin this section by describing our logic-level model for the digital circuit under test. Using this model, we define and illustrate two types of logic faults. We then define fault tuples and show how combinations of fault tuples (called macrofaults) can precisely represent logic faults.

A. Circuit Representation

A logical or gate-level model of a digital circuit has been the preferred representation for a design by the test community. For many reasons, it still continues to be the most widely used representation of a design.

The logical structure of a synchronous digital circuit $C$ can be described using a netlist of interconnected gates (e.g., AND, OR, and XOR) and memory elements (latches or flip-flops). The logical behavior of a synchronous digital circuit can be described using a finite-state machine (FSM) model. An FSM can be characterized [44] by:

1) a set of primary input patterns $X$;
2) a set of primary output patterns $Y$;
3) a set of states $S$;
4) state transition and output functions $\delta : X \times S \rightarrow S$ and $\lambda : X \times S \rightarrow Y$, respectively.

For example, the sequential adder circuit in Fig. 1(a) is composed of six gates and a single D-type flip-flop. The state transition table, which is a tabulation of the state transition function $\delta$ and output function $\lambda$, for the sequential adder circuit is shown in Fig. 1(b). Each entry in the transition table is the next state $s_{i+1}$ and output $y$ produced for the current state $s_i$ and input $x$, that is, $s_i, x \rightarrow s_{i+1}, y$, where $x \in X$, $y \in Y$, and $s_i, s_{i+1} \in S$. Without loss of generality, the gates and signal interconnects are assumed to have zero delay, implying that all signal lines become instantaneously stable at their final values immediately after the clock edge for a duration equal to the clock period. The memory elements of a circuit (i.e., the flip-flops or latches) are assumed to be ideal in that there are no setup and hold time requirements. In addition, the current state consisting of all memory outputs is assumed to persist for the entire duration of the clock cycle and instantaneously change at the start of the next clock cycle. For example, Fig. 1(c) illustrates the timing of the adder circuit in Fig. 1(a) for some sample input values, assuming the initial state of the signal $Cout$ is unknown.

In addition to a state transition table, sequential Boolean expressions that describe the logic behavior of each signal line can be used to characterize the overall behavior of a logical circuit. In general, a sequential Boolean expression is a Boolean expression involving current and past values of circuit signal lines. More formally, a sequential Boolean expression can be defined recursively as follows.

1) Logic 0 and logic 1 are sequential Boolean expressions.
2) If $\alpha$ is the current or past value of a signal line, then $\alpha$ is a sequential Boolean expression. (Note that future values
of signal lines are not allowed since this could lead to
undecidable behavior.)

3) If \( \alpha \) is a sequential Boolean expression, then \( (\alpha) \) is a sequential Boolean expression.

4) If \( \alpha \) is a sequential Boolean expression, then \( \text{NOT}(\alpha) \) is a sequential Boolean expression.

5) If \( \alpha \) and \( \beta \) are sequential Boolean expressions, then the disjunction \( \alpha + \beta \) is a sequential Boolean expression.

6) If \( \alpha \) and \( \beta \) are sequential Boolean expressions, then the conjunction \( \alpha \cdot \beta \) is a sequential Boolean expression.

As already mentioned, the behavior of a signal line can be represented using a sequential Boolean expression. For example, the equation \( 9(i) = A(i) \cdot B(i) \) is an example of a sequential Boolean equation that indicates that the logic value of signal line \( 9 \) in any clock cycle \( i \) is the logical AND of the values of lines \( A \) and \( B \) in the same clock cycle \( i \), where \( i \geq 1 \) is an integer. Similarly, the logic value of \( \text{Cout}(i) \) in clock cycle \( i \) is the value of signal line \( \text{Cin} \) of the previous clock cycle, that is, \( \text{Cin}(i-1) \). The sequential Boolean equations describing the logical behavior of other signals are listed in Fig. 1(d).

To prevent undefined behavior due to self-referencing, a sequential Boolean expression representing the behavior of a signal line \( l \) cannot reference the current value of \( l \). For example, the equation \( l(i) = k(i)l(i) \) is invalid.

**B. Defect Modeling**

A defect can affect a circuit in multiple dimensions. For example, the unintended connection between two lines (i.e., a bridge defect) can change the logic function of a circuit, slowdown or even speedup the circuit [28], affect the amount of current that it draws, or have no perceptible impact whatsoever. Some effects of a defect are typically abstracted or altogether ignored when representing it above the physical level as a fault. At the logical level, only the logical effects of a defect can be precisely captured. In this paper, we focus on the abstraction of a defect to the logical level. We define two types of logic faults for a circuit \( C \) with an input set \( X \), state set \( S \), output set \( Y \), state transiting function \( \delta \), and output function \( \lambda \).

**Definition 1:** A state transition fault (STF) \( s_i, x \rightarrow (s_i+1, y)/(s_i', y') \) affecting a circuit \( C \) causes \( C \) to produce a next state \( s_i+1 \in S \) and an output \( y' \in Y \) for the present state and input combination \( s_i, x \), where either or both \( s_i' \neq s_i+1 \in S \) and \( y' \neq y \in Y \) are true.

**Definition 2:** A signal line fault (SLF) affecting a signal line \( l \in C \) changes the sequential Boolean equation from its specified behavior \( l(i) \) to an incorrect behavior represented by another equation \( q(i) \).

Many traditional fault types can be exactly represented using STFs. For instance, Fig. 2(a) shows the state transition table resulting from the multiple stuck-at (MSL) fault \{12-sa0, 5-sa1\} for the adder circuit in Fig. 1(a). The underlined entries show the specific changes in the state transition function \( \delta \) and output function \( \lambda \). Specifically, it causes six simultaneously
existing STFs, namely, 1, 00 → (0,1)/(0,0), 0, 01 → (0,1)/(1,1), 1, 01 → (1,0)/(1,1), 0, 10 → (0,1)/(1,1), 1, 10 → (1,0)/(1,1) and 0, 11 → (1,0)/(1,1). One obvious significant shortage of representing this MSL fault and other types of faults as one or more STFs is the complexity involved in mapping the fault to a set of STFs.

The problem of fault mapping is eliminated when using SLFs. For example, a line j stuck-at-v is easily represented by replacing the sequential Boolean equation for line j with \( j(i) = v \). Representing an MSL fault simply involves replacing all lines involved with the appropriate constant equation. Other fault types can also be easily represented. For example, Fig. 2(b) shows various faults represented as changes in one or more sequential Boolean equations. Moreover, the types of misbehavior allowed by SLFs goes beyond what can be expressed by a set of STFs. For example, STFs cannot be used to represent a fault that increases the number of circuit states. However, since an SLF is a sequential Boolean equation of current and past values of any number of circuit lines, an arbitrary amount of “new state” introduced by a fault can be modeled using SLFs. For example, a line \( j \) stuck-at-\( v \) can be modeled by replacing the sequential Boolean equation for line j by \( j(i) = v \). The value \( v \) may be any line in the circuit under test. The value \( v \) is used in the defective circuit. Note that two separate symbols are used for the unknown value, that is, \( U = X = \{0, 1\} \). This is necessary so that the unknown value \( U \), which is attributed to a modeled defect, can be distinguished from (the fault-free) unknown value \( X \). In addition, we adopt a conservative approach for \( U \) propagation, that is, \( AND(U, X) = OR(U, X) = U \).

For generality, we assume that a test sequence \( TS = \{t_1, t_2, \ldots, t_q\} \) applied to a circuit under test is done so in a non-scan manner, that is, each test \( t_i \) is applied during operational clock cycle \( i \). The clock cycle constraint \( T \in \{N, i + N, i < N, i, i + N, i < i, i + N, i < N, i < i, i + N \} \) of a fault tuple specifies when the value \( v \) of signal line \( l \) is relevant. For example, \( T = N \), where \( N \) is some positive integer, designates the \( N \)th clock cycle where test pattern \( t_N \in TS \) is applied. \( T = i \) is used to represent an arbitrary clock cycle, while \( T = i + N \) is used to designate the clock cycle that is \( N \) cycles in the future from some clock cycle \( i \). Element \( i > i \) is used to designate a clock cycle \( j \) that is greater than some clock cycle \( i \). The remaining elements \( (i < i) \) are similarly defined.

The fault modeling capability of fault tuples stems from their combination. A product is a conjunctive combination of \( n \) fault tuples, namely, \( P = f_1 \cdot f_2 \cdot \cdots \cdot f_n \), where \( n \geq 1 \) is an integer. Products can also be combined. A macrofault is a disjunctive combination of \( m \) fault-tuple products, namely, \( M = P_1 + P_2 + \cdots + P_m \), where \( m \geq 1 \) is an integer. Macrofaults are used to alter the behavior of a circuit \( C \) to represent all the ways that a modeled defect deviates, in a logical manner, from the defect-free circuit \( C \). Detection of a macrofault means that the underlying constraints defined by the fault-tuple products must be satisfied.

Definition 3: A condition fault tuple \( f_i = \{l, v, T\}^c \) for a circuit \( C \ni l \) affected by a macrofault \( M = P_1 + P_2 + \cdots + P_m \), where \( f_i \in P_j \in M \), is satisfied by a test sequence \( TS \) if \( l = v \) in the defective circuit during a clock cycle described by \( T \).

Definition 4: A value fault tuple \( f_i = \{l, v, T\}^v \) for a circuit \( C \ni l \) affected by a macrofault \( M = P_1 + P_2 + \cdots + P_m \), where \( f_i \in P_j \in M \), is satisfied by a test sequence \( TS \) if \( l = v \) in the defect-free circuit \( C \) during a clock cycle described by \( T \).

Definition 5: An error fault tuple \( f_i = \{l, v, T\}^e \) for a circuit \( C \ni l \) affected by a macrofault \( M = P_1 + P_2 + \cdots + P_m \), where \( f_i \in P_j \in M \), is satisfied by a test sequence \( TS \) if \( l = v \) in the defect-free circuit \( C \) during a clock cycle described by \( T \).

In general, determining if a fault is satisfied means checking that the signal line is a particular logic value at a particular clock cycle within the defective or defect-free circuit. Similar to an individual fault tuple, a product of fault tuples can be satisfied.

Definition 6: A product \( P_j = f_1 \cdot f_2 \cdot \cdots \cdot f_n \) for a circuit \( C \) affected by a macrofault \( M = P_1 + P_2 + \cdots + P_m \), where \( f_i \in P_j \in M \), is satisfied by a test sequence \( TS \) if each fault tuple \( f_i \in P_j \in M \) is satisfied.
2) product has one or more error fault tuples. In the first case, product satisfaction simply means that all the tuples have to be satisfied by the applied test sequence $TS$, that is, there are no errors. On the other hand, when $P$ contains one or more error fault tuples, satisfaction of all the condition tuples must be achieved before lines become “faulty.” In other words, satisfaction of all the condition tuples within a product means that each signal line $l$ of $f_i = \{l, v_f, T\}^e$ is set to $v_f$ in the defective circuit. Note, however, that satisfaction of the condition tuples does not necessarily mean that error tuple lines are erroneous. Whether an error exists on an error tuple line depends on the value of the error tuple. Therefore, for a product to be satisfied, all the condition tuples must first be satisfied, followed by satisfaction of the error and value tuples. Because the order in which tuples are satisfied matters means that product satisfaction can depend on the sequence of signal line changes within a circuit.

To guide the creation of well-formed products of fault tuples, certain construction rules should be followed. In general, the rules are meant to ensure that: 1) incompatible requirements are not imposed on a signal line and 2) incompatible faulty values are not imposed on a signal line.

1) If there exists a fault tuple $f_x \in P$ with clock-cycle constraint $T_x \in \{i + N, > i, < i, \geq i\}$, then there must be at least one other fault tuple $f_y \in P$ with $T_y = i$.

2) For each error fault tuple $f_x \in P$, there cannot be a condition fault tuple $f_y \in P$ such that $T_y > T_x$ always.

3) For each fault tuple $f_x \in P$ with $l_x = a$ and $T_x = b$, there cannot be a fault tuple $f_y \in P$ such that $l_y = a$ and $T_y = b$. In other words, there cannot be two fault tuples in the same product that refers to the same signal line potentially during the same clock cycle.

Rule 1 enforces the existence of a reference fault tuple for other fault tuples that depend on the instantiability of an arbitrary clock cycle $i$. Rule 2 is needed to ensure that creation of a faulty value in the circuit under test is not dependent upon some event in the future. Rule 3 prevents redundancy and incompatibility.

**Example 1:** Fault-tuple products $P_1 = (a, 0, i + 1)\{b, 1, i + 2\}^e$ and $P_2 = (a, 0, i)\{b, 1, i + 2\}^e$ are not well-formed products. Specifically, $P_1$ violates Rule 1 since there does not exist a fault tuple $f_x \in P_1$ with $T_x = i$. The intent of $P_1$ can be easily captured by the well-formed product $P_3 = (a, 0, i)\{b, 1, i + 1\}^e$. $P_2$ does not violate Rule 1 but does violate Rule 2 since the time element of the condition fault tuple $(b, 1, i + 2)^e$ is greater than the error fault tuple $(a, 0, i)^e$, that is, $i + 2 > i$ always. Rule 2 simply enforces intuition about how faulty behavior manifests. For $P_2$, it simply does not make sense to have a faulty value appear on signal line $a$ based on what happens with line $b$ two clock cycles into the future. Examples of well-formed products include $P_4 = (a, 0, 23)^e\{b, 1, 25\}^e\{a, 0, 25\}^e$, $P_5 = (a, 0, i)^e\{b, 1, i + 2\}^e\{c, 1, > i\}^e$, and $P_6 = (a, 0, 23)^e\{c, 1, i\}^e\{x, 0, i + 1\}^e\{x, 0, 1 + i\}^e$. Note that $P_7$ does not violate Rule 2 since it is possible for the condition tuple to be satisfied before the error tuple and it is not required that the value tuples $(x, 1, i + 1)^e$ and $(x, 0, 1 + i)^e$ (which require signal line $x$ to have an error 1/0 in clock cycle $i + 1$) be satisfied before the error tuple $(c, 1, i)^e$.

Product satisfaction of a macrofault by a test sequence may cause signal lines in a circuit under test to become erroneous. For the macrofault to be detected, an error resulting from a satisfied product must be observed.

**Definition 7:** A macrofault $M = P_1 + P_2 + \cdots + P_m$ affecting a circuit $C$ is detected by a test sequence $TS$ if there is at least one $P_j \in M$ that is satisfied and at least one error (if any) is propagated to an output of $C$.

Satisfaction of one or more products means that multiple faulty sites can exist over both time and space. This means that errors resulting from satisfied products can interact to cause error masking, oscillation, product unsatisfaction, and product satisfaction depending on: 1) the macrofault itself; 2) the structure of the circuit $C$ under test; 3) the test sequence $TS$ applied; and 4) the sequence of signal line changes.

**Example 2:** Fig. 3 shows a three-gate combinational circuit affected by a macrofault. Specifically, line $C$ is stuck-at-0 only when line $Z_1$ is a logic 0. The macrofault representing the

---

Fig. 3. Simple gate-level circuit affected by the macrofault $M = (Z_1, 0, i)^e\{C, 0, i\}^e$. (a) Test sequence applied satisfies the one condition tuple. (b) The fault value of the error tuple is then injected ($v_f(C) = 0$), which means signal line $C$ is erroneous since $v_g(C) = 1$. (c) However, error propagation unsatisfies $(C, 0, i)^e$, causing reversion back to the initial circuit state shown in (a).
conditional stuck-at fault is $M = (Z_t, 0, i)^e(C, 0, i)^e$. Notice that the circuit input values shown cause oscillation. However, application of the inputs $A = 1, B = 1, D = 1, \text{and} \ G = 1$ will detect the macrofault $M$ without oscillation.

Similar to products, there are rules for ensuring that macrofaults are well formed. Specifically, there cannot be two products $P_i \in M$ and $P_j \in M$ that can establish the same or conflicting faulty values on the same signal line during the same clock cycle. This rule ensures that macrofault products are compatible and irredundant. Determining whether two products $P_i \in M$ and $P_j \in M$ adhere to this rule is a complex task, so test tools (e.g., a fault simulator) must be able to identify and report this incompatibility.

Macrofaults can also be combined based on their detection statuses due to the application of a TS to the circuit under test. To indicate the detection status of a macrofault $M_i$, we use $D(M_i) = 1$ to indicate that $M_i$ is detected by TS while $D(M_i) = 0$ indicates that $M_i$ is not detected by TS. A compound macrofault $CM = F(D(M_1), D(M_2), \ldots, D(M_k))$ is defined as a sum-of-products expression of $k$ macrofault detection statuses, where $k \geq 1$ is an integer.

Definition 8: A compound macrofault $CM = F(D(M_1), D(M_2), \ldots, D(M_k))$ affecting a circuit $C$ is detected by a test sequence $TS$ if $F$ evaluates to a one.

Compound macrofaults, although more general, are similar to ANDed and ORed pattern faults [39]. Specifically, they both enable one to investigate the ability of a test sequence $TS$ to detect modeled faults simultaneously or alternatively. It should be noted, however, that each fault within a compound macrofault or an ANDed or ORed pattern fault is separate faulty machines, that is, they cannot interact to cause error masking or error reinforcing.

B. Modeling Capability

To illustrate the modeling capability of fault tuples, we show that any STF or SLF can be mapped to an equivalent macrofault.

Definition 9: Consider two faults $F_1$ and $F_2$ that separately affect a circuit $C$ with faulty state transition tables $ST_1$ and $ST_2$, respectively. $F_1$ and $F_2$ are defined to be equivalent if $ST_1$ and $ST_2$ are identical.

Theorem 1: An STF $s_{i,1}, x \rightarrow (s_{i+1,1}, y)/(s_{i+1,1}, y')$ that affects a circuit $C$ is equivalent to the fault-tuple macrofault $M = f_{1,1} \cdot \cdots \cdot f_{1,n_1} \cdot f_{2,1} \cdot \cdots \cdot f_{2,n_2} \cdot f_{3,1} \cdot \cdots \cdot f_{3,n_3} \cdot f_{4,1} \cdot \cdots \cdot f_{4,n_4}$ where

1) $f_{1,j} = (l_j, v, i)^e$ is a condition fault tuple where $v$ is the logic value corresponding to a memory output signal $l_j$ for $1 \leq j \leq n_1$ and $n_1$ is the number of memory elements of the circuit $C$;

2) $f_{2,j} = (l_j, v, i)^e$ is a condition fault tuple where $v$ is the logic value corresponding to a circuit input $l_j$ for $1 \leq j \leq n_2$ and $n_2$ is the number of primary inputs of the circuit $C$;

3) $f_{3,j} = (l_j, v, i+1)^e$ is an error fault tuple where $v$ is the logic value corresponding to an erroneous memory element $l_j$ for $1 \leq j \leq n_3$ and $n_3$ is the number of memory elements that are in error for $s_{i+1,1}$;

4) $f_{4,j} = (l_j, v, i+1)^e$ is an error fault tuple where $v$ is the logic value corresponding to an erroneous primary output $l_j$ for $1 \leq j \leq n_4$ and $n_4$ is the number of circuit outputs that are in error for $y'$.

Proof: Consider any input sequence $TS$ that places the circuit $C$ into state $s_{i}$ and applies input combination $x$. By definition, fault tuples $f_{1,1}, \ldots, f_{1,n_1}, f_{2,1}, \ldots, f_{2,n_2}$ are satisfied by $TS$. During the next clock cycle, each memory output line $l_j$ corresponding to each fault tuple $f_{i,j}$ is erroneous. Similarly, each primary output line $l_j$ corresponding to each fault tuple $f_{i,j}$ is also in error. Thus, the change in the behavior of $C$ represented by $s_{i,1}, x \rightarrow (s_{i+1,1}, y)/(s_{i+1,1}, y')$ is equivalent to the change modeled using the fault-tuple macrofault $M = f_{1,1} \cdot \cdots \cdot f_{1,n_1} \cdot f_{2,1} \cdot \cdots \cdot f_{2,n_2} \cdot f_{3,1} \cdot \cdots \cdot f_{3,n_3} \cdot f_{4,1} \cdot \cdots \cdot f_{4,n_4}$.

Corollary 1: A multiple STF $\{ST_{F_1}, ST_{F_2}, \ldots, ST_{F_N}\}$ that affects $C$ is equivalent to $M = P_1 + P_2 + \cdots + P_N$, where each $P_j$ corresponds to STF $f_j$, as described in Theorem 1, for $1 \leq j \leq N$.

Theorem 2: Given an SLF $l(i) = p_{1,1} \cdot \cdots \cdot p_{1,n_1} + p_{2,1} \cdot \cdots \cdot p_{2,n_2} + \cdots$ that affects $C$ is equivalent to $M = f_{1,1} \cdot \cdots \cdot f_{1,n_1} \cdot f_{2,1} \cdot \cdots \cdot f_{2,n_2} \cdot f_{3,1} \cdot \cdots \cdot f_{3,n_3} \cdot f_{4,1} \cdot \cdots \cdot f_{4,n_4}$.

Proof: First, consider the special case where $l(i) = 1$. For $l(i) = 1$, the SLF is equivalent to the stuck-at-1 fault on line $l$, which means $M$ can be reduced to $f_{1,1} \cdot \cdots \cdot f_{1,n_1} + f_{2,1} \cdot \cdots \cdot f_{2,n_2} + \cdots$. Thus, the change in the behavior of $C$ represented by $s_{i,1}, x \rightarrow (s_{i+1,1}, y)/(s_{i+1,1}, y')$ is equivalent to the change modeled using the fault-tuple macrofault $M = f_{1,1} \cdot \cdots \cdot f_{1,n_1} \cdot f_{2,1} \cdot \cdots \cdot f_{2,n_2} \cdot f_{3,1} \cdot \cdots \cdot f_{3,n_3} \cdot f_{4,1} \cdot \cdots \cdot f_{4,n_4}$.

Now consider the more general case where $l(i)$ is not the constant function zero or one. Consider the first term in $l(i), p_{1,1} \cdot \cdots \cdot p_{1,n_1}$. Recall that each literal $p_{i,j} \in p_{1,1} \cdot \cdots \cdot p_{1,n_1}$ denotes the current or past value of some signal line $l \in C$. Let $p_{a,b} \in p_{1,1} \cdot \cdots \cdot p_{1,n_1}$ be a literal that references the value of some signal line $q_{max} \geq 0$ cycles in the past while all other literals $p_{x,y} \neq p_{a,b}$ reference a signal line $w$ cycles in the past, where $0 \leq w \leq q_{max}$. In other words, $p_{a,b}$ is a literal that references some signal line value furthest in the past. (Note, $q_{max} \geq 0$ means that each literal $p_{i,j} \in p_{1,1} \cdot \cdots \cdot p_{1,n_1}$ only references current values of circuit signal lines.) Next, we partition $p_{1,1} \cdot \cdots \cdot p_{1,n_1}$ into $k$ subproducts $\{\{s_{1,1}, \cdots, s_{1,n_1}\}, \{s_{2,1}, \cdots, s_{2,n_2}\}, \cdots, \{s_{k,1}, \cdots, s_{k,n_k}\}\}$ based on the $k$ different signal lines that are in $p_{1,1} \cdot \cdots \cdot p_{1,n_1}$, where $n_1^2 + n_2^2 + \cdots + n_k^2 = n_1$. Now consider the first subproduct $s_{1,1} \cdot \cdots \cdot s_{1,n_1}$ and let the signal line associated with $s_{1,1} \cdot \cdots \cdot s_{1,n_1}$ be termed $l_{n_1}$. Each literal $s_{1,j} \in s_{1,1} \cdot \cdots \cdot s_{1,n_1}$ is represented using the
condition fault tuple \((l_{n_1}, v, i + q_{\text{max}} - w)^c\), where \(v = 0\) if \(s_{i,j}\) refers to the complement of \(l_{n_1}\) or \(v = 1\) if \(s_{i,j}\) refers to the true value of \(l_{n_1}\), and \(w\) is the number of cycles in the past that \(s_{i,j}\) references \(l_{n_1}\). Similarly, each remaining subproduct \((s_{1,1} \cdot \ldots \cdot s_{n_1,n_1}) \in p_{1,1} \cdot \ldots \cdot p_{1,n_1}\) is represented using condition fault tuples that reference their corresponding signal line \(l_{n_1}'\).

At this point, each literal \(p_{i,j} \in p_{1,n_1} \cdot \ldots \cdot p_{1,n_1}\) has been mapped to a condition fault tuple \(f_{1,j}\) as described earlier. Now consider the fault-tuple product \(P = f_{1,1} \cdot \ldots \cdot f_{1,n_1}\) of all the condition fault tuples. Note that product \(P\) is satisfied for any input sequence \(TS\) applied to \(C\) affected by \(l(i) = p_{1,1} \cdot \ldots \cdot p_{1,n_1}\) as described earlier. For the clock cycle that \(p_{1,1} \cdot \ldots \cdot p_{1,n_1}\) is true, signal line \(l\) is a logical 1. Adding the error fault tuple \(f_{i,n_1+1} = (i, 1, i + q_{\text{max}})^c\) to \(P\) results in a new product \(P' = f_{1,1} \cdot \ldots \cdot f_{1,n_1+1}\) that ensures signal line \(l\) is a logic 1 whenever \(p_{1,1} \cdot \ldots \cdot p_{1,n_1}\) is true. In a similar fashion, each remaining term of \(l(i) = p_{1,1} \cdot \ldots \cdot p_{1,n_1}\) can be mapped to a product of fault tuples. Thus, the sum of fault-tuple products \(f_{1,1} \cdot \ldots \cdot f_{1,n_1}\cdot f_{1,n_1+1} + f_{2,1} \cdot \ldots \cdot f_{2,n_2}\cdot f_{2,n_2+1} + \ldots\) guarantees that \(l = 1\) whenever \(l(i) = p_{1,1} \cdot \ldots \cdot p_{1,n_1}\) is true. Using the same approach for the complement of \(l(i)\), \(l'(i) = q_{1,1} \cdot \ldots \cdot q_{1,m_1}\cdot q_{2,m_2} + \ldots\) can be mapped to the sum of products \(f_{1,1} \cdot \ldots \cdot f_{1,m_1}\cdot f_{1,m_1+1} + f_{2,1} \cdot \ldots \cdot f_{2,m_2}\cdot f_{2,m_2+1} + \ldots\) which ensures that \(l = 0\) whenever \(l'(i) = q_{1,1} \cdot \ldots \cdot q_{1,m_1}\cdot q_{2,m_2} + \ldots\) is true. Thus

\[
M = f_{1,1} \cdot \ldots \cdot f_{1,n_1}\cdot f_{1,n_1+1} + f_{2,1} \cdot \ldots \cdot f_{2,n_2} \\
+ f_{2,n_2+1} + \ldots \cdot f_{1,1} \cdot \ldots \cdot f_{1,m_1}\cdot f_{1,m_1+1} \\
+ f_{2,1} \cdot \ldots \cdot f_{2,m_2}\cdot f_{2,m_2+1} + \ldots
\]

is equivalent to the SLF \(l(i) = p_{1,1} \cdot \ldots \cdot p_{1,n_1} + p_{2,1} \cdot \ldots \cdot p_{2,n_2} + \ldots\).

**Example 3:** Reconsider the SLF \(11(i) = 4(i-1)4(i-2)4(i-3)\) from Section III-B. The complement of this SLF expression is \(11'(i) = 4'(i) + 4'(i-1) + 4'(i-2) + 4'(i-3)\). Therefore, the macrofault that is equivalent to \(11'(i) = 4(i-1)4(i-2)4(i-3)\) is given by

\[
M = (4, 1, i)^c \cdot (4, 1, i+1)^c \cdot (4, 1, i+2)^c \cdot (4, 1, i+3)^c \\
\cdot (11, 1, i+3)^c + (4, 0, i)^c \cdot (11, 0, i)^c + (4, 0, i)^c \\
\cdot (11, 0, i+1)^c + (4, 0, i)^c \cdot (11, 0, i+2)^c \\
+ (4, 0, i)^c \cdot (11, 0, i+3)^c.
\]

**Corollary 2:** A multiple SLF \(\{SLF_1, SLF_2, \ldots, SLF_N\}\) that affects \(C\) is equivalent to \(M = M_1 + M_2 + \cdots + M_N\), where each \(M_j\) corresponds to \(SLF_j\) as described in Theorem 2 for \(1 \leq j \leq N\).

**C. Defect Modeling**

A common characteristic of many faults is that much of their behavior matches that of the fault-free circuit. It is therefore both efficient and sufficient to model the difference between a fault and the correct behavior of the circuit. SLFs, however, require all the behavior of a fault to be completely characterized. Fault-tuple-based macrofaults, on the other hand, are much more efficient because they allow the difference between a faulty and fault-free circuit to be efficiently modeled.

1) **Basic Faults:** Using the small combinational circuit in Fig. 4(a), we give macrofault descriptions for various types of well-known faults [Fig. 4(b)]. This list is obviously not comprehensive but is simply meant to illustrate the diversity of classical faults that can be represented using fault tuples.

   a) **Stuck-at:** An SSL fault is represented using a macrofault consisting of a single product that consists of a single error fault tuple. For instance, 7-sa0 is modeled using \(M = (7, 0, i)^c\). The use of \(i\) in the clock cycle constraint indicates that the product of \(M\) can be satisfied during any one clock cycle. Multiple stuck-line (MSL) faults can be easily modeled as well. An N-line MSL fault is modeled by a macrofault consisting of \(N\) error fault tuples ORed together. For example, the MSL fault \(\{8-sa1, 7-sa0, 4-sa0\}\) is represented using \(M = (8, 1, i)^c + (7, 0, i)^c + (4, 0, i)^c\).

   b) **Pseudo stuck-at:** A pseudo SSL fault is used in \(I_{DDQ}\) testing. Specifically, it is used to activate bridges between nodes within a gate (including the power supply nodes) by applying opposite logic values to the nodes involved in the bridge. In addition, the gate should be sensitized to ensure elevated current flow. For example, pseudo 9-sa1 is modeled using \(M = (9, 0, i)^c(10, 1, i)^c\).

   c) **Transition:** In our timing model, a timing fault localized to a gate or a path segment is assumed to violate the slack constraint along every path that emanates from the fault site(s). For timing faults that increase delay, this means the faulty value (i.e., the initial logic value) can propagate (non-robustly) along any path. For timing faults that decrease delay, this means the faulty value (i.e., the final logic value) can propagate (non-robustly) along any path and arrive too early. Note that faults that reduce delay can cause failure via a hold-time violation. Fig. 4(b) shows macrofaults for a transition fault, a slow-to-rise-only transition fault, and a slow-to-fall path delay fault. Later, we show how more complicated delay faults can be modeled using fault tuples.

   d) **Two-line bridge:** In a zero-resistance bridge fault involving two signal lines, one line imposes its value on the other. Fig. 4(b) shows five examples of two-line bridge faults. In an AND-type bridge fault, a logic 0 value is presumed to override a logic 1. Hence, for any clock cycle \(i\) where one line is driven to a logic 0 and the other line is supposed to be driven to a logic 1, then the latter line becomes erroneous. An OR-type bridge, where a logic 1 is presumed to be stronger, is modeled in a similar way, and for a dominating bridge, one line

---

Here we present many macrofault models of defects. However, different misbehaviors may be attributable to the defects discussed. The point we want to stress is the modeling capability of fault tuples and not the particular defect models themselves.
is presumed to always impose its value on the other line. The four-way bridge model, which subsumes the behavior of the AND-type, OR-type, and the dominating bridge, is also shown.

Other types of voting models based on the relative strengths of the driving transistors can also be modeled [26], [27]. The last macrofault example describes behavior of a bridge between lines 11 and 13 [see Fig. 4(b)] based on the input values applied to the gates that drive these affected lines. For this example, we (arbitrarily) assume that an equal or greater number of n-type driving transistors always dominate an equal or lesser number of p-type driving transistors. Fig. 5 enumerates the behavior of this bridge for all 16 possible values of drivers 1, 8, 4, and 5. The last entry in Fig. 4(b) shows a macrofault that captures the behavior of rows 4 and 5 only; a complete macrofault would also include products corresponding to rows 6, 7, 9, 10, 11, and 13–15.

Finally, fault tuples can also capture the role of gates downstream from the bridge site. For instance, it is possible that some combination of contending transistors produces an intermediate voltage that is interpreted differently by downstream gates. For example, row 5 in Fig. 5 involves the contention of one n-type transistor from the NOR gate and two parallel p-type
transistors from the NAND gate. A product that represents downstream gates interpreting an intermediate value produced by this contention differently is \( P = (0, 1, i)^c \cdot (4, 0, i)^c \cdot (5, 0, i)^c \cdot (1, 0, i)^c \cdot (12, 0, i)^c \), where the OR gate interprets the intermediate voltage as a faulty logic 1 and the AND and XNOR gates interpret the voltage as a faulty logic 0.

2) Complex Faults: The last section demonstrated that fault tuples can be used to represent many well-known fault types. Here, we present more complicated examples to further demonstrate the modeling capability of fault tuples.

a) Unknown behavior: The exact logical behavior of many defect types cannot be predicted a priori. Hence, many researchers have used the unknown or don’t care logic value \{0, 1\} to represent a defect’s potential effect on a signal line [45]–[47]. In [45], Nanya and Goosen introduce the Byzantine fault model where they state: “A fault that causes a signal line to take on X will be called a Byzantine fault...”. Moreover, they state that the Byzantine fault model can be used to represent many types of faults including multiple stuck-at and open faults. This statement has been investigated and validated by many others (e.g., [46], [47]).

Using the faulty unknown value \( U \), fault tuples can model various types of Byzantine faults used to represent, for example, resistive bridges, interconnect and transistor opens, and “X-lists” [47]. Fig. 6 shows examples of these four types of faults.

Fig. 6(a) shows a resistive bridge between signal lines 1 and 2. Each time the fault is activated with different logic values on the two affected signal lines, a faulty unknown value is assumed to result on all of the fanout lines. Fig. 6(b) shows an interconnect open. It is assumed that the receiving line \( B \) is a faulty unknown value unless transitions on both capacitively coupled aggressors establish an erroneous value on line \( B \). In addition, it is assumed that line \( B \) is initially an unknown faulty value, modeled by the error tuple \((B, U, 1)^e\). Fig. 6(c) shows a TSO fault for a two-input NAND gate. Like the interconnect open, it can begin with an unknown faulty value [modeled by the product \((A, 0, 1)^c \cdot (B, 1, 1)^c \cdot (OUT, U, 1)^e\). Similar to other researchers, we model this fault as a stuck-at-0 fault when a 0-to-1 transition takes place with \( B = 1 \) [3]. In addition, however, we use a faulty unknown value to model a floating output when the NAND gate is suppose to produce consecutive logical ones through the open transistor. Finally, Fig. 6(d) shows an X-list [47] consisting of lines 3–5 and 8–12. An unknown fault that can potentially affect these lines is modeled using the one-product macrofault consisting of the three error fault tuples shown.

b) Multiple faults: Fault tuples can model multiple faults of the same type or of different types. For example, in [15], a spot defect consisting of polysilicon [referred to as the “nasty polysilicon spot defect” (NPSD)] that affects the gate of a transistor is shown to simultaneously cause both a bridge and an open defect in the affected gate. Using Spice simulations, the logical behavior of the NPSD is characterized and modeled using fault tuples.

Multiple faults do not have to be localized to the same gate or region of the circuit however. For example, macro-fault \( M \) represents three-line and two-line bridge faults that affect lines \{9, 10, 11\} and \{12, Cout\}, respectively, in the sequential adder circuit in Fig. 7. The behavior of the two-line bridge is arbitrarily chosen to be zero dominating, and for the three-line bridge we assume a majority function between the lines. Obviously, other misbehaviors including delay behavior (see Section IV-C2d) can also be used to model the behavior of each constituent bridge site.

c) Feedback faults: One of the most complicated fault types to model is one that causes feedback. In Fig. 8(a), a bridge
that creates feedback between signal lines $Z$ and $G$ is assumed to convert the AND gate and its driving logic into a latch. The set, reset, and hold behaviors are defined by the circuit signal line values shown in the table in Fig. 8(b). A macrofault that captures this behavior is $M = (G, 0, i)^c (E, 1, i + 1)^c (G, 0, i + 1)^c$ and $P_5 = (G, 0, i)^c (E, 1, i + 1)^c (D, 1, i + 1)^c (G, 0, i + 1)^c$ capture the hold behavior when the current latch state is logic 0 while $P_4 = (G, 1, i)^c (E, 1, i + 1)^c (C, 1, i + 1)^c (G, 1, i + 1)^c$ and $P_3 = (G, 0, i)^c (E, 1, i + 1)^c (D, 1, i + 1)^c (G, 0, i + 1)^c$ represent the hold operation when the current latch state is logic 1. Products $P_6 = (G, U, i)^c (E, 1, i + 1)^c (C, 1, i + 1)^c (G, U, i + 1)^c$ and $P_7 = (G, U, i)^c (E, 1, i + 1)^c (D, 1, i + 1)^c (G, U, i + 1)^c$ are also needed when the hold operation is performed when the current latch state is unknown due to lack of a reset or set. Product $P_8 = (G, U, 1)^c$ is used to initialize the latch to an unknown state. Finally, note that the set behavior is not included in the macrofault $M$ since no error results from this operation, that is, all signal line values match the good circuit when a set operation is performed. It could be included, however, without altering the fidelity of the model.

d) Delay faults: Previously (Section III-A), several delay faults were described that included a transition, slow-to-rise gate, and slow-to-fall path delay faults. However, fault tuples are not limited to these types of delay faults. Segment delay faults [48] and multiple delay faults of various types can be modeled as well. Moreover, test effectiveness and the level of faulty behavior for a delay fault can be controlled.

Consider a multiple delay fault consisting of a slow-to-rise segment delay fault [48] comprising signal lines 4, 8, and 10, and a slow-to-rise gate delay fault affecting $Z_2$ in Fig. 9. Macrofault $M_1$ listed in Fig. 9 specifies the low-quality detection of this multiple delay fault since the only requirements specified include: 1) a transition along the targeted segment with error propagation to some observable point or 2) a transition at the targeted gate with error propagation to some observable point. $M_1$ is considered to be low quality since it is possible for glitches (e.g., due to hazards or capacitive coupling) to invalidate the activation of either fault or prevent path sensitization.

However, detection quality can be controlled using fault tuples. For example, consider the path delay fault from the flip-flop output to the flip-flop input, namely, $\{Q, Z_2\}$. Macrofault $M_2$ in Fig. 9 sensitizes this path for a slow-to-rise fault using non-robust [29], [31] criteria. Macrofault $M_3$ increases the quality of detection by placing “strong non-robust” [49] conditions on the path’s side inputs. Finally, by using the hazard-free logic values $hf_0$ and $hf_1$ described in [50], $M_4$ and $M_5$ sensitize the path using robust and hazard-free criteria, respectively.

Test quality can be precisely controlled even more. For example, reconsider macrofault $M_5$ that requires each side input to be hazard free. The fault tuples of $M_5$ ensure that the side inputs do not experience glitches due to logic hazards. However, it is possible that glitches could result from crosstalk generated by capacitive coupling. $M_6$ places “super hazard-free” conditions on the path’s side inputs to prevent test invalidation due to crosstalk by requiring signal $a_1$, which is coupled to side-input 19, to be steady at logic 1.

The severity of misbehavior can also be controlled using fault tuples. This characteristic is particularly important for delay defects since detection is more likely if the conditions for worst case misbehavior can be established. For example, the delay of the slow-to-rise path delay fault on $\{Q, Z_2\}$ can be potentially increased by causing transitions of the opposite polarity on signals $a_2$ and $a_3$. (See macrofault $M_7$ in Fig. 9.) Another example of enabling worst case behavior is the “pulse-stretching” effect exhibited by circuits implemented in silicon-on-insulator (SOI) technology [51]. If signal lines are held at a steady value for some duration of time, then some transistors become more conductive while others become less conductive. Subsequent circuit inputs that cause a sequence of alternating transitions will be delayed. Such delays can accumulate along a path of gates to either cause or exacerbate a delay fault. Fault tuples can be easily used to establish the pulse-stretching effect created by SOI. For example, macrofault $M_8$ non-robustly sensitizes the slow-to-rise path delay fault $\{Q, Z_2\}$ with a pulse-stretching effect established over three clock cycles.

Fault tuples are not limited to modeling delay faults that cause excessive delay. It is possible for a failure to result from delay reduction or “speedup.” For example, a hold-time failure can occur when the delay of a $Q$-to-$D$ flip-flop path is reduced.
Fig. 9. Examples of various types of delay faults modeled using fault-tuple macrofaults.

<table>
<thead>
<tr>
<th>Delay fault description</th>
<th>Macrofault representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Segment delay fault affecting lines {4, 8, 10} and a slow-to-rise gate delay fault at $Z_2$ both with non-robust sensitization.</td>
<td>$M_1 = (4,0,i)^{(8,1,i)}(10,0,i)^{(4,1,i+1)}(8,0,i+1)^{(3,1,i+1)^{(9,1,i,1)}} + (18,0,i)^{(18,1,i+1)}(9,1,i+1)^{(Z_2,0,i+1)^{(19,0,i)}}(18,1,i+1)^{(Z_2,0,i+1)^{(19,1,i,1)}}$</td>
</tr>
<tr>
<td>Slow-to-rise path delay fault {Q, Z_2} with non-robust sensitization.</td>
<td>$M_2 = (Q,0,i)^{(Q,1,i,1)}(11,1,i,1)^{(19,1,i,1)}$</td>
</tr>
<tr>
<td>Slow-to-rise path delay fault {Q, Z_2} with strong non-robust sensitization.</td>
<td>$M_3 = (Q,0,i)^{(Q,1,i,1)}(11,1,i,1)^{(19,1,i,1)}$</td>
</tr>
<tr>
<td>Slow-to-rise path delay fault {Q, Z_2} with robust sensitization.</td>
<td>$M_4 = (Q,0,i)^{(Q,1,i,1)}(11,1,i,1)^{(19,1,i,1)}(19,hi,f,i,1)^{(1)}$</td>
</tr>
<tr>
<td>Slow-to-rise path delay fault {Q, Z_2} with hazard-free sensitization.</td>
<td>$M_5 = (Q,0,i)^{(Q,1,i,1)}(11,1,i,1)(19,hi,f,1,i,1)^{(1)}(19,hi,f,1,i,1)^{(1)}$</td>
</tr>
<tr>
<td>Slow-to-rise path delay fault {Q, Z_2} with “super hazard-free” sensitization.</td>
<td>$M_6 = (Q,0,i)^{(Q,1,i,1)}(11,1,i,1)(19,hi,f,1,i,1)^{(1)}(19,hi,f,1,i,1)^{(1)}(a_1,1)^{(a_1,hi,f,1,i,1)^{(1)}}$</td>
</tr>
<tr>
<td>Slow-to-rise path delay fault {Q, Z_2} aggravated by noise via aggressors $a_2$ and $a_1$.</td>
<td>$M_7 = (Q,0,i)^{(Q,1,i,1)}(11,1,i,1)^{(19,1,i,1)}(a_2,1,i)^{(a_2,0,i,1)^{(a_2,0,i,1)}}(a_2,1,i,1)^{(1)}$</td>
</tr>
<tr>
<td>Slow-to-rise path delay fault {Q, Z_2} aggravated by SOI pulse-stretching effect.</td>
<td>$M_8 = (Q,1,i)^{(Q,1,i,1)}(15,1,i,1)^{(15,1,i,1)}(15,1,i,2)^{(1)Z_2,1,i,1}(Z_2,1,i,1)^{(1)}(11,1,i,4)^{(1)}$</td>
</tr>
<tr>
<td>Fast-to-rise path delay fault {Q, Z_2} with non-robust sensitization aggravated by noise via $a_2$ and $a_3$, and the SOI pulse-reduction effect.</td>
<td>$M_9 = (Q,1,i)^{(Q,0,i,1)}(15,0,i,1)^{(15,0,i,1)}(15,0,i,2)^{(1)Z_2,0,i,1}(Z_2,1,i,1)^{(1)}(11,1,i,4)^{(1)}(a_2,0,i,3)^{(a_2,1,i,3)^{(1)}}(a_2,0,i,4)^{(1)}(a_2,1,i,4)^{(1)}$</td>
</tr>
<tr>
<td>Two-detect of a slow-to-rise gate delay fault at 9 aggravated by noise via aggressor $a_4$.</td>
<td>$M_{10} = (6,1,i)^{(7,1,i)}(6,0,i,1)^{(9,0,i,1)^{(a_4,0,i,1)}}(a_4,0,i,1)^{(1)}(x_1,0,i,1)^{(1)} + (6,1,i)^{(7,1,i)}(7,0,i,1)^{(9,0,i,1)}(a_4,0,i,1)^{(1)}(x_1,0,i,1)^{(1)}$</td>
</tr>
<tr>
<td>Segment delay fault affecting lines {4, 8, 10} that prevents sensitization to $Z_2$.</td>
<td>$M_{12} = (4,0,i)^{(8,1,i)}(10,0,i)^{(4,1,i,1)^{(8,0,i,1)}}(10,0,i,1)^{(3,1,i,1)^{(Z_2,1,i,1)^{(1)}}(Z_2,1,i,1)^{(1)}} + (4,0,i)^{(8,1,i)}(10,0,i)^{(4,1,i,1)}(8,0,i,1)^{(10,0,i,1)^{(3,1,i,1)^{(Z_2,0,i,1)^{(1)}}(Z_2,0,i,1)^{(1)}}}$</td>
</tr>
</tbody>
</table>

by a bridge defect [28], SOI pulse reduction [51], and/or capacitive coupling [52]. Consider again the path {Q, Z_2} in the circuit in Fig. 9. Assume that a fast-to-rise transition along this path, exacerbated by bridging and coupling to lines $a_2$ and $a_3$, respectively, and subjected to SOI pulse reduction over three cycles leads to a hold-time violation. Macrofault $M_9$ describes
the non-robust conditions [32] for detecting this fault under these circumstances.

Importantly, it should be noted that fault tuples cannot precisely specify the timing of signal line transitions. The relative timing of signal line transitions is necessary, however, to establish worst case behavior. For example, the transition on signal \( a_2 \) specified by \( M_{10} \) will maximize delay reduction along path \( \{ Q, Z_2 \} \) if it occurs at the same time as the transition on \( Q \). But attempting to control or analyze signal transitions in such a precise manner is virtually impossible given the timing variation caused by circuit operating conditions (temperature, noise, etc.) and fabrication variations. Instead, a statistical approach must be adopted where faults are detected multiple times with varying conditions [53]–[55]. Detecting a given fault multiple times increases the likelihood that the worst case behavior of the defect is established. Fault tuples can be used to detect a fault multiple times [56]. For example, macrofaults \( M_{10} \) and \( M_{11} \) specify that a slow-to-rise gate delay fault affecting signal line 9, aggravated by coupling on line \( a_4 \), should be detected but with different conditions on the driver of \( a_4 \).

Earlier, we mentioned that delay faults were assumed to violate the slack constraint on every path that emanates from the fault site. This assumption, however, can be relaxed by using value tuples. For example, macrofault \( M_{12} \) shows how the segment delay fault of \( M_1 \) can be modified to prevent detection through \( Z_2 \). Specifically, \( M_{12} \) requires that the segment delay fault be detected but requires \( Z_2 \) to be error free. In a similar way, other delay fault requirements (e.g., TARO [57]) can be handled using fault tuples.

The examples described here illustrate that fault tuples can capture the logical misbehavior of a variety of defects. Moreover, fault tuples can also be used to modulate the level of misbehavior and the quality of test, thus enabling a significant amount of flexibility for various testing tasks.

V. FAULT-TUPLE APPLICATIONS

Fault tuples can enhance the testing process as well as enable new approaches to testing. In the following sections, we highlight some of the applications of fault tuples.

A. Test Generation and Fault Simulation

The obvious application for fault tuples is to improve automatic test pattern generation (ATPG) and fault simulation. To illustrate this point, Fig. 10 shows the results of “grading” a 100% stuck-at test set against a variety of faults that include: 1) layout-extracted bridge faults; 2) TSO faults; and 3) transition delay faults. The results were generated during a single run of the fault-tuple simulator FATSIM [12], a capability not offered by any other fault simulator and obviously not limited to these fault types. For this small example, it is clear that the SSL test set consisting of 22 test patterns is unable to detect all the targeted faults. The undetected faults, however, can then be passed to a fault-tuple ATPG tool to generate tests or prove them redundant. Using the fault-tuple test generator FATGEN [13], the 39 undetected faults from the FATSIM simulation were all found to be detectable with an additional 26 test patterns. Finally, using FATGEN and FATSIM together to directly target all the faults leads to a test set of only 38 tests, a reduction of more than 20% over fault grading followed by ATPG on the undetected faults. Although small, this example illustrates one of the most significant applications of fault tuples.

B. Diagnosis

Reliance on a finite number of fault models hinders the diagnosis of circuit failures. The ability to accurately model and distinguish arbitrary misbehavior [14] means that the accuracy of a diagnosis outcome can be precisely assessed [15], [16]. Moreover, it is possible to increase the confidence of diagnosis by generating new tests that confirm, invalidate, and/or refine hypotheses concerning the source of failure [17].

Fig. 11 shows one possible approach to diagnosis using fault tuples. Similar to current diagnosis approaches, the starting point is the failing circuit and the production test set used to test the circuit. Every failing circuit will produce an erroneous test response for one or more of the production tests. Comparing and analyzing fault-tuple simulations with actual circuit test responses will lead to an initial set of “candidates,” that is, possible failure sites and associated misbehaviors. The candidates, along with the test responses, are fed to the diagnostic macrofault generator. Using various known and yet-to-be-developed circuit and process analysis techniques, the generator will refine the candidates by forming hypotheses about the type of fault or misbehavior within the failing circuit. The postulated
misbehaviors can be fed back to the simulator in an attempt to either confirm or disprove the hypotheses. In addition, new diagnostic tests can be generated for application to the failing circuit. The diagnostic tests will create new test responses that enable the diagnosis flow to be repeated, further refining the list of candidates. The diagnosis loop in Fig. 11 can continue until the desired level of confidence in the diagnosis outcome is achieved. Our work in [17] uses a flow very similar to the one just described.

Current diagnostic tools terminate after producing the initial set of possible failure locations. This can be followed by an expensive time-consuming analysis that involves physical examination of the failing circuit. A fault-tuple based approach is appealing in that failure can be both localized and characterized, possibly eliminating the need for physical failure analysis. Moreover, the proposed framework is completely automatable on the tester, which means the time required for diagnosis can be significantly reduced while simultaneously improving accuracy and confidence.

C. Interfault Collapsing

Local dominance and equivalence relationships for a single fault type have been exploited to reduce test set size and test generation time. However, these relationships have not been explored across fault types. Using fault tuples, local dominance, and equivalence relationships across various fault types can be derived [14]. Experiment results using four different fault types (SSL, transition, TSO, and path delay) for ISCAS85 and ITC99 benchmark circuits revealed that test set size can be reduced by as much as 19% through interfault collapsing [14].

Interfault dominance and equivalence relationships are also useful for diagnosis. For example, if two different faults (possibly of different types) are candidates for a given diagnosis, we know it may be possible to distinguish them if there exists a local dominance relationship between them. On the other hand, if the two faults are functionally equivalent, then other approaches must be used to distinguish them [17], [58].

Example 4: Fig. 12(a) shows a dominance graph for a TSO fault (\(ts_1\)) and a slow-to-fall transition fault (\(tf_1\)) affecting a two-input NAND gate. In the dominance graph, a directed edge from fault \(f_1\) to a fault \(f_2\) means that \(f_2\) dominates \(f_1\), that is, all the test sequences that detect \(f_1\) will also detect \(f_2\). The double arrow in Fig. 12(a) means that \(tf_1\) and \(ts_1\) have the same tests, which, for these two faults, means they are functionally equivalent. The equivalence of \(tf_1\) and \(ts_1\) means that ATPG resources solely directed at distinguishing these two faults would be wasted for this two-input NAND gate.

On the other hand, Fig. 12(b) shows that the slow-to-rise transition fault \(tf_1\) and the stuck-at-0 fault \(ts_1\) have different tests and therefore can possibly be distinguished at the circuit outputs. For example, a test sequence that detects the macrofault \(M = (z, 1, i)^c(b, 0, i + 1)^c\) means \(tf_1\) can be distinguished from three equivalent SSL faults.

Dominance relationships can also be effectively used to optimize the quality of test. Consider again testing the path \(\{Q, Z_2\}\) in the circuit in Fig. 9. Macrofaults \(M_2 - M_6\) specify testing \(\{Q, Z_2\}\) with increasing levels of quality. To ensure that the highest quality of testing is achieved, one can specify a dominance relationship among \(M_2 - M_6\), namely, \(M_6 \rightarrow M_5 \rightarrow M_4 \rightarrow M_3 \rightarrow M_2\). A fault-tuple ATPG tool that exploits dominance would first target \(M_6\). If a test sequence is found for \(M_6\), then by the dominance relation macrofaults \(M_2 - M_5\) could be dropped without performing fault simulation. If, however, \(M_6\) was found to be redundant, then \(M_5\) could be targeted. Following this strategy would ensure that the targeted path would be tested with the highest quality test. Exploiting dominance in this way means that the priority and optionality features of GFM [43] can also be accomplished using fault tuples.

D. Other Applications

- Design-and-technology-specific quality metrics: With the capability of fault tuples to model arbitrary defects, one can create customized fault lists for fault simulation and test generation that are derived from the characteristics of the design and the underlying fabrication technology. For instance, one can imagine a scenario where SSL testing is deemed to be sufficient for one portion of the IC under test (e.g., sparse control logic generated by synthesis tools) and a separate area where bridge and open defect testing is believed necessary (e.g., hand-drawn data path circuitry). Using defect statistics and defective parts per million (DPM) objectives, it is conceivable that a customized fault list that satisfies desired quality objectives over the lifetime of the product can be derived.

- Fault model generation: It is typically the case that a small fraction of a circuit’s applied test set detects a large percentage of the defective parts. Analyzing this subset of tests and their corresponding responses may lead to the discovery of new and improved fault models. Fault tuples enable such an analysis since they make it possible to analyze a variety of misbehaviors.

- Gate-level verification: Because fault tuples can model any change in the logical behavior of a circuit, they can be used for gate-level verification tasks. For example, any of the gate-level design errors described in [59] can be modeled as macrofaults.
VI. SUMMARY

The fault-tuple mechanism is a general and effective methodology for modeling failures in digital integrated circuits. Although many characteristics of defects (changes in current, voltage, timing, etc.) cannot be explicitly represented, fault tuples can represent any effects these attributes may have on the logical behavior of a digital circuit. In this paper, we have formally shown that fault tuples can precisely and efficiently model arbitrary changes in the logical behavior of a digital circuit. In addition, we have illustrated the modeling capability with a plethora of examples. Finally, we have described how many test tasks (fault simulation, ATPG, diagnosis, etc.) can all be enhanced with fault tuples.

REFERENCES


---

R. D. (Shawn) Blanton (S’93–M’95–SM’03) received the B.S. degree in engineering from Calvin College, Grand Rapids, MI, in 1987, the M.S. degree in electrical engineering from the University of Arizona, Tucson, in 1989, and the Ph.D. degree in computer science and engineering from the University of Michigan, Ann Arbor, in 1995.

He is currently a Professor of electrical and computer engineering at Carnegie Mellon University, Pittsburgh, PA, where he also serves as an Associate Director at the Center for Silicon System Implementation. His current research interests include test and diagnosis of integrated heterogeneous systems.

Kumar N. Dwarakanath (S’97–M’93) received the B.S. degree from Birla Institute of Technology and Science, Pilani, India, in 1996, the M.Eng. degree from Cornell University, Ithaca, NY, in 1997, and the Ph.D. degree in electrical and computer engineering from Carnegie Mellon University, Pittsburgh, PA, in 2004.

He is currently with DAFCA Inc., Framingham, MA, where he focuses on post-silicon debug infrastructure and associated algorithms for enabling efficient system-on-chip debug. His research interests are in the areas of CAD for manufacturing test, defect diagnosis, yield modeling, and design debug.

Rao Desineni (S’02–M’04) received the B.S. degree in electronics engineering from the National Institute of Technology, Trichy, India, in 1998, the M.S. and Ph.D. degrees in electrical and computer engineering from Carnegie Mellon University, Pittsburgh, PA, in 2001 and 2006, respectively, where his research was focused on defect-based testing, fault modeling, automatic test pattern generation, and, most recently, physically aware fault diagnosis.

Since October 2005, he has been with IBM Fishkill, working in the 300-mm Diagnostics Integration and Characterization Department, where his responsibilities include developing new diagnosis solutions to enable rapid identification of key yield detractors in a volume manufacturing environment.