A Theory of Mutations with Applications to Vacuity, Coverage, and Fault Tolerance

Orna Kupferman
Hebrew University
orna@cs.huji.ac.il

Wenchao Li
UC Berkeley
wenchao@berkeley.edu

Sanjit A. Seshia
UC Berkeley
sseshia@eecs.berkeley.edu

***Abstract*— The quality of formal specifications and the circuits they are written for can be evaluated through checks such as vacuity and coverage. Both checks involve mutations to the specification or the circuit implementation. In this context, we study and prove properties of mutations to finite-state systems. Since faults can be viewed as mutations, our theory of mutations can also be used in a formal approach to fault injection. We demonstrate theoretically and with experimental results how relations and orders amongst mutations can be used to improve specifications and reason about coverage of fault tolerant circuits.**

I. Introduction

Model checking [9] has proved successful in verifying the correctness of finite-state systems with respect to their specifications. In recent years, however, there has been growing awareness of the importance of challenging positive answers of model-checking tools. The main reasons include the possibility of errors or imprecision in the modeling of the system or the specification. In order to detect such errors, one needs to pose and answer questions about the quality of the modeling and the exhaustiveness of the specification.

Vacuity and *coverage* are two checks proposed to answer these questions. In vacuity, the goal is to detect cases where the system satisfies the specification in some unintended trivial way. For example, the temporal logic specification $\varphi = \mathbf{G}(req \rightarrow \mathbf{F}grant)$ ("every request is eventually followed by a grant") is satisfied in a system in which requests are never sent, but that clearly points to an error in the model or the specification. In coverage, the goal is to increase the exhaustiveness of the specification by detecting components of system behavior that do not play a role in the verification process (i.e., are "not covered by the specification"). For example, a system in which a request is followed by two grants satisfies the specification φ above, but only one of the grants plays a role in the satisfaction.

Vacuity and coverage have a lot in common; in particular, both checks involve iterating the verification procedure on a *mutated input*. In vacuity, mutations are introduced in the specification, whereas in coverage, the system is mutated. If verification succeeds even with a mutated implementation (specification), it indicates to the user that some component of the specification (implementation) is redundant. The user can then inspect the implementation and specification, possibly with a trace illustrating the redundancy, and decide how to tighten that component of the design. However, there is currently little guidance to users, in the form of automated tool support, on how the specification can be tightened to improve coverage or eliminate vacuity.

A particular challenge is posed in the verification of fault-tolerant designs. The reason is that faults can be viewed as mutations to the design. Thus, an ability to mutate the implementation and still satisfy the specification is somehow inherent in a fault-tolerant design, and need not indicate low coverage [19]. How then can we check coverage of fault-tolerant designs?

In this paper, we present a new theory of mutations, with applications to certifying the correctness of fault-tolerant circuits, checking coverage and vacuity of specifications, and potentially also to the synthesis of environment assumptions. We build upon our previous work [15], which formally proved that vacuity and coverage are *dual notions* using a basic set of mutations. Roughly speaking, a mutation μ is the dual of a mutation ν if proving lack of coverage using μ implies showing vacuity using ν, or vice-versa. Duality gives us a systematic approach to modifying the specification when low coverage is detected.

This paper makes the following novel contributions:

- *New useful mutations:* We extend the set of mutations defined in [15] and show how this extended set allows for better improvement of specifications for our target applications.
- *Properties of mutations:* We define properties of mutations such as *monotonicity* and *invertability* and explore relationships amongst them. A mutation μ is monotonic if its application on both the implementation and the specification preserves satisfaction. Mutations are inverse of each other if composing them results in the identity mutation. We prove, for example, that monotonic mutations that are inverse of each other are dual. These properties and results enable us to find more dual mutations, and to obtain simple proofs for duality of mutations.
- *Aggressiveness order between mutations:* We introduce the notion of an *aggressiveness order* between mutations. Intuitively, mutation μ is more aggressive than mutation ν, if applying μ to the implementation or specification makes satisfaction harder than applying ν. Aggressiveness orders amongst several mutations are presented. We demonstrate how the aggressiveness order can be used to improve the coverage of specifications.
- *Coverage for fault tolerant circuits:* It has been observed that some circuits are inherently fault-tolerant, in the sense that the specification is satisfied even when a fault is injected in some circuit component [19]. The impact of a fault is often only a small performance penalty which does

not make the circuit incorrect.

Using the notion of an aggressiveness order in this setting enables us to define coverage for fault-tolerant circuits. In essence, there is low coverage if we can apply to the implementation a mutation that is more aggressive than the one it is designated to tolerate, and still satisfy the specification. The least aggressive mutation that causes the implementation to fail the specification can then be returned to the designer as a formal indicator of low coverage of the specification.

We demonstrate our approach to checking coverage and improving specifications on RTL implementations of circuits, including VIS benchmarks [20], and the Verilog design of a chip multiprocessor router [18].

Related Work. There is much previous work on vacuity and coverage, which we do not have space to survey here; we instead point the reader to some relevant papers ([1], [2], [4], [5], [7], [8], [14], [16]).

Our work is the first to formalize the connection between vacuity, coverage, and fault tolerance. It does so with a new theory of mutations, many of which are inspired by physical faults. Some past work on coverage can be expressed concisely in terms of the theory we develop here. For example, Große et al. [12] describe a method for estimating coverage for bounded model checking which can be viewed as applying the flip mutation (defined in Section III-B) at a specific cycle, and then asking the model checker for a witness. Similarly, Fummi et al. [11] use the single stuck-at fault (mutation) to estimate coverage of specifications, which is an instance of coverage checking using the stuck-at mutations we define here. The ordering amongst mutations that we derive can potentially be used to formally compare these different proposals for coverage checking. Our theory of mutations can also be used with implementation-independent techniques for estimating coverage (e.g., [10], which uses single stuck-at faults).

Outline of Paper. We begin, in Section II, with an overview of our approach, applied to a simple fault-tolerant design. Section III gives the formal model and definitions for developing our theory of mutations. In Section IV, we describe some key properties of mutations. We introduce the notion of an aggressiveness order in Section V. Section VI discusses some applications of our theory and presents experimental results.

II. Overview

We give an overview of how our mutation-based approach can be used to certify fault tolerance of the arbiter module of a chip multiprocessor (CMP) router design [18].

In the general case, the CMP router has N input channels and N output channels, where N is a parameter. We focus our discussion on the arbiter module of the router (see Figure 1), which implements a round-robin arbitration for each output channel. The vector $\vec{r}$ represents an input N-bit request vector from the input channels (r_i is high whenever channel i issues a request), and the vector $\vec{g}$ represents the corresponding output N-bit grant vector (g_i is high whenever channel i is granted access). The signals p_{ij}, for $0 \leq i < j \leq N-1$, maintain the priorities of the channels, with p_{ij} being high indicating that channel i has higher priority over input channel j The policy of the arbiter is to grant access to a channel i that requests an access only if every other channel that issues a request has lower priority than i. For example, in Figure 1, both r_0 and r_1 are high, but since p_{01} is high and no other requests are high, g_0 is high. The arbiter then inverts all p_{0j}'s in the next cycle.

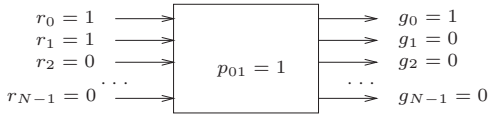

Fig. 1. The arbiter in the CMP router.

Let us consider the case $N = 2$, and the specification $\mathcal{S}$ asserting that whenever Channel 0 issues a request, then access should be granted within 12 cycles. Thus, in temporal logic, $\mathcal{S}$ is given by

$$\mathcal{S} = \mathbf{G}\,(r_0 \rightarrow \mathbf{X}^{\leq 12} g_0)$$

The implementation $\mathcal{I}$ of the CMP router was model checked against $\mathcal{S}$ and it was found that $\mathcal{I}$ satisfies $\mathcal{S}$.

We then analyzed the design for resilience to *single event upsets* (also called soft errors). A single event upset (formally defined in Section III-B) is a fault where, in any run of the system, a single state-holding Boolean element can take the opposite value of what it is supposed to take, at a single but arbitrary cycle of operation. In particular, we checked whether $\mathcal{I}$ satisfies $\mathcal{S}$ even when a soft error is injected into the arbiter priority bit p_{01} of $\mathcal{I}$. (This experiment was performed using the methodology of verification-guided error resilience [19].)

We found that the arbiter bit p_{01} is resilient to a single soft error with respect to $\mathcal{S}$. Does this mean that $\mathcal{S}$ is not a good specification?

To analyze this, we applied a more aggressive mutation, introducing up to two soft errors in bit p_{01}. It turned out that $\mathcal{S}$ is still satisfied! The designer must decide if this additional level of fault tolerance is really necessary. Let us assume that it is not – that the circuit need not be resilient to two soft errors in p_{01}. The question is: how can we improve $\mathcal{S}$?

The soft error mutation does not yet have an associated dual mutation, so in order to improve $\mathcal{S}$, we analyzed the impact of other mutations on $\mathcal{I}$ for which duality results are available. One such mutation, indicated by the form of the LTL property, is to delay the output g_0 of the arbiter module by k cycles, for increasing values of k.

It turned out that the specification $\mathcal{S}$ is satisfied even after applying these delay mutations to $\mathcal{I}$ for $k = 1, 2, 3, 4$, but not $k = 5$. The dual mutation for the delay of 4 cycles is to make the output g_0 in the specification change prematurely by 4 cycles. Applying this prematureness mutation to $\mathcal{S}$ gives us the new mutant specification $\mathcal{S}' = \mathbf{G}\,(r_0 \rightarrow \mathbf{X}^{\leq 8} g_0)$,

We then checked $\mathcal{I}$ mutated with one and two soft errors against $\mathcal{S}'$: the former satisfied $\mathcal{S}'$ but the latter did not. Thus, the new specification $\mathcal{S}'$ is a good specification for analyzing fault tolerance: it certifies the resilience of p_{01} against a single soft error, but catches the occurrence of more than one soft error.

III. Formal Model and Definitions

We present a unified formal notation to model implementations and specifications as *circuits* (Section III-A). We then

present several mutations including those proposed earlier as well as newer mutations (Section III-B).

A. Circuits

A sequential circuit (*circuit*, for short)[1] is a tuple $\mathcal{C} = \langle I, O, C, \theta, \delta, \rho \rangle$, where I is a set of input signals, O is a set of output signals, and C is a set of control signals that induce the state space 2^C. Accordingly, the three other components of the tuple are defined as:

- $\theta : 2^I \rightarrow 2^{2^C} \setminus \emptyset$ is a nondeterministic initialization function that maps every input assignment (that is, assignment to the input signals) to a nonempty set of initial states
- $\delta : 2^C \times 2^I \rightarrow 2^{2^C} \setminus \emptyset$ is a nondeterministic transition function that maps every state and input assignment to a nonempty set of possible successor states
- $\rho : O \rightarrow \mathcal{B}(C)$ is an output function that maps every output signal in O to a Boolean formula over the set of control signals; a signal $x \in O$ holds (is 1) at exactly the states $t \in 2^C$ that satisfy $\rho(x)$.

It is required that $I \cap C = I \cap O = \emptyset$. Possibly $O \cap C \neq \emptyset$, in which case for all $x \in O \cap C$, we have $\rho(x) = c$, and x is termed an observable control signal. Note that the interaction between the circuit and its environment is initiated by the environment. Once the environment generates an input assignment $i \in 2^I$, the circuit starts reacting with it from one of the states in $\theta(i)$. Note also that $\theta(i)$ and $\delta(s, i)$ are not empty for all $i \in 2^I$ and $s \in 2^C$. Thus, $\mathcal{C}$ is *receptive*, in the sense it never gets stuck.

We will interpret a state s (or an input i or output o) as a set comprising exactly those signals that evaluate to 1 in that state. Thus, $s \cup \{x\}$ denotes the state in which signal x is 1 and which agrees with s on all other signal values. Similarly, $s \setminus \{x\}$ denotes the state with signal $x = 0$ and agreeing with s on all other signal values.

Given an input sequence $\xi = i_0, i_1, \ldots \in (2^I)^\omega$, a *computation* of $\mathcal{C}$ on ξ is a word $w = w_0, w_1, \ldots \in (2^{I \cup O})^\omega$ such that there is a *path* $s = s_0, s_1, \ldots \in (2^C)^\omega$ in $\mathcal{C}$ that can be traversed while reading ξ, and w describes the input and output along this traversal. Formally, $s_0 \in \theta(i_0)$ and for all $j \geq 0$, we have $s_{j+1} \in \delta(s_j, i_j)$ and $w_j = i_j \cup \rho(s_j)$. The *language* of $\mathcal{C}$, denoted $L(\mathcal{C})$, is the union of all its computations.

Consider an implementation $\mathcal{I} = \langle I, O, C, \theta, \delta, \rho \rangle$ and a specification $\mathcal{S} = \langle I', O', C', \theta', \delta', \rho' \rangle$. We assume that $I' \subseteq I$, $O' \subseteq O$, and $C' \subseteq C$. In applications such as hierarchical refinement, the implementation may still not be precise, so we allow nondeterminism in both $\mathcal{I}$ and $\mathcal{S}$. Satisfaction of $\mathcal{S}$ in $\mathcal{I}$ can be formalized in both the linear and the branching frameworks for specification. In the linear framework, we say that $\mathcal{I}$ is *contained* in $\mathcal{S}$, denoted $\mathcal{I} \subseteq \mathcal{S}$, if $L(\mathcal{I}) \subseteq L(\mathcal{S})$. In the branching framework, we define satisfaction by means of *simulation*. A binary relation $H \subseteq 2^C \times 2^{C'}$ is a *simulation* from $\mathcal{I}$ to $\mathcal{S}$ if for all $\langle s, s' \rangle \in H$, the following conditions hold: (1) $\rho(s) \cap O' = \rho'(s')$, and (2) For each $i \in 2^I$, and $t \in \delta(s, i)$ there is $t' \in \delta'(s', i \cap I'))$ such that $H(t, t')$. We say that H is *initial with respect to $\mathcal{I}$ and $\mathcal{S}$* if for every input assignment $i \in 2^I$ and state $s \in \theta(i)$, there is a state $s' \in \theta'(i \cap I')$ such that $H(s, s')$. We say that $\mathcal{S}$ *simulates* $\mathcal{I}$, denoted $\mathcal{I} \leq \mathcal{S}$, if there is an initial simulation from $\mathcal{I}$ to $\mathcal{S}$. Intuitively, it means that $\mathcal{S}$ has more observable behaviors than $\mathcal{I}$. Formally, every universal property over the observable signals $I' \cup O'$ that is satisfied in $\mathcal{S}$ is also satisfied in $\mathcal{I}$ [3], [13]. The branching approach is stronger, in the sense that $\mathcal{I} \leq \mathcal{S}$ implies that $\mathcal{I} \subseteq \mathcal{S}$, but not vice versa. For a recent survey comparing the linear and branching approaches see [17]. We say that $\mathcal{I}$ satisfies $\mathcal{S}$, denoted $\mathcal{I} \models \mathcal{S}$ if $\mathcal{I} \subseteq \mathcal{S}$ (in the linear approach) or $\mathcal{I} \leq \mathcal{S}$ (in the branching approach).

B. Mutations on Circuits

In previous work [15], various circuit mutations have been defined. The mutations are partitioned into three classes: those that remove behaviors, modify behaviors, and add behaviors. It is possible to control the temporal pattern of mutations, specifying the cycles at which mutations occur. It is also possible to apply mutations on top of each other (like function composition).[2]

Below, we briefly describe the previously proposed mutations along with some new mutations we have devised.

Mutations that remove behaviors. Consider a specification $\mathcal{S}$ and an implementation $\mathcal{I}$ such that $\mathcal{I} \models \mathcal{S}$. If there is a mutant $\mathcal{S}'$ of $\mathcal{S}$ such that $\mathcal{S}'$ has fewer behaviors than $\mathcal{S}$ and still $\mathcal{I} \models \mathcal{S}'$, we say that $\mathcal{I}$ *vacuously satisfies* $\mathcal{S}$. Vacuous satisfaction suggests that behaviors expected by the specifier have not been exhibited in the implementation. We list some typical mutations below.

- *Arbitrary removal of behaviors*. This mutation restricts the functions θ and δ according to a given set of transitions to be removed. Mutations in this class are useful for modeling temporal-logic vacuity, for finding maximal vacuity, vacuity along computations, and arbitrary delays that can be avoided.
- *Removing behaviors that depend on a signal*. This mutation is parameterized by a signal $x \in I \cup C \setminus O$ and is obtained from the original circuit by removing transitions that depend on x. Thus, a transition stays only if it exists with both $t \setminus \{x\}$ and $t \cup \{x\}$, for either input assignments $t \in 2^I$ or states $t \in 2^C$. We term the mutation DEP_x.
- *Restricting a signal to a value*. A mutation in this class is parameterized by a signal $x \in C \cup O$ and it restricts the value of x to 0 (or 1) by disabling transitions to states in which the value of x is 1 (resp. 0). We term the mutations REST_x_TO_0 and REST_x_TO_1.

Mutations that modify behaviors. Consider a specification $\mathcal{S}$ and an implementation $\mathcal{I}$ such that $\mathcal{I} \models \mathcal{S}$. If there is a mutant $\mathcal{S}'$ of $\mathcal{S}$ such that $\mathcal{S}'$ has different behaviors than $\mathcal{S}$ and still $\mathcal{I} \models \mathcal{S}'$, or there is a mutant $\mathcal{I}'$ of $\mathcal{I}$ such that $\mathcal{I}'$ has different behaviors than $\mathcal{I}$ and still $\mathcal{I}' \models \mathcal{S}$, we say that $\mathcal{I}$ *diversely satisfies* $\mathcal{S}$. Diverse satisfaction suggests that some components in the implementation or the specification do not play a role in the satisfaction.

[1] Note that, although we focus mainly on applications to hardware verification in this paper, the formalism is a *finite-state transducer*, and the results also apply to finite-state models of software.

[2] Recall that we assume that both the implementation and its specification are given by means of circuits. As demonstrated in [15], many mutations on circuits have analogous mutations on temporal-logic formulas, which can be applied when the specification is given by means of a formula rather than a circuit.

- *Forcing a signal to be flipped*. This mutation is parameterized by a signal x, and it consistently flips the value of x; i.e., x always takes the opposite value of what it is supposed to take. We term the mutation FLIP_x.
- *Forcing a signal to get stuck*. A mutation in this class is parameterized by a signal $x \in I \cup O \cup C$ and it forces x to get stuck at 0 (or 1) by acting as if $x = 0$ (resp. $x = 1$) regardless of its actual value. For example, if $x \in C$, then the valuation of the initialization, transition, and output functions may change, as θ, δ and ρ are now applied to the evaluation containing the possibly-modified value of x. We term the mutations STICK_x_AT_0 and STICK_x_AT_1.
- *Forcing a delayed or a premature output*. A mutation in this class causes the output of the circuit to be delayed or prematured (by a fixed number of cycles, specified by the user). We term the mutations DELAY_k and PREMATURE_k, where k is the number of cycles.
- *Inserting perturbation*. A mutation in this class inserts small local perturbation to the circuit. The mutations include permuting the output sequence, stuttering the output, or applying other noise on it [15].

Mutations that add behaviors. Consider a specification $\mathcal{S}$ and an implementation $\mathcal{I}$ such that $\mathcal{I} \models \mathcal{S}$. If there is a mutant $\mathcal{I}'$ of $\mathcal{S}$ such that $\mathcal{I}'$ has more behaviors than $\mathcal{S}$ and still $\mathcal{I}' \models \mathcal{S}$, we say that $\mathcal{I}$ *loosely satisfies* $\mathcal{S}$. Loose satisfaction suggests that some components of the implementation are not covered by the specification.

- *Adding an arbitrary set of behaviors*. A mutation in this class extends the functions θ and δ according to a given set of transitions to be added. Mutations in this class are useful for detecting computations and delays that are not covered by the specifications.
- *Freeing a signal*. A mutation in this class is parameterized by a signal x and it adds to the original circuit behaviors that agree with existing behaviors of it on everything but x. We term this mutation FREE_x.

Additional mutations. The formalism used in [15] is slightly different, and the output function of a circuit there is $\rho : 2^C \rightarrow 2^O$. We have changed the formalism in order to extend the set of mutations. In our formalism, $\rho : O \rightarrow \mathcal{B}(C)$ associates with each output signal a formula over the control signals. This enables us to define a *conditional stuck-at* mutation, which, as we later show in Example 4.3, has a natural dual mutation. We also define mutations for soft errors, those that remove or add transitions according to guards in $\mathcal{B}(C)$, and variants to the DELAY_k and PREMATURE_k mutations.

- *Single event upset (SEU).* (modifies behaviors) Recall from the start of this section that we can qualify a mutation with a pattern that specifies the cycle(s) at which that mutation occurs. This pattern can reflect non-deterministic behavior. One prominent example of such a pattern-mutation is the single event upset (SEU), also called a *soft error*. A k-SEU_x is obtained by combining FLIP_x with a pattern that non-deterministically chooses to perform FLIP_x at k non-deterministically chosen cycles of operation. A 1-SEU_x is also simply called an SEU.
- *Conditional stuck-at.* (modifies behaviors) Consider an output signal $x \in O$. For a formula $\beta \in \mathcal{B}(C)$, let β_STICK_x_AT_1 be the mutation that replaces ρ_x by $\rho_x \vee \beta$, and let β_STICK_x_AT_0 be the mutation that replaces ρ_x by $\rho_x \wedge \neg\beta$. Intuitively, β_STICK_x_AT_1 sticks x to 1 in states that satisfy the condition β, whereas β_STICK_x_AT_0 sticks x to 0 in such states.
- *Conditional addition/removal of transitions.* (adds/removes behaviors) For formulas $\beta, \gamma \in \mathcal{B}(C)$, let (β, γ)_ADD be the mutation that adds transitions from all states that satisfy β to all states that satisfy γ. In the mutated circuit $\mathcal{C}_\mu$, we have $t \in \delta_\mu(s, i)$ iff $t \in \delta(s, i)$ or both $\beta(s)$ and $\gamma(t)$. Likewise, the mutation (β, γ)_REMOVE removes all transitions from states that satisfy β to states that satisfy γ.
- *Bounded delay/prematureness in output.* (modifies behaviors) DELAY_LEQ_k and PREMATURE_LEQ_k are variants of the DELAY_k and PREMATURE_k mutations. Here, the parameter k does not specify the exact delay or prematureness, but an upper bound on it. Thus, the output is delayed or prematured by *at most* k cycles.

IV. Properties of Mutations

Let M_a, M_m, and M_r be the sets of mutations that respectively add, modify, and remove behaviors. Then, $M_{imp} = M_a \cup M_m$ is the set of mutations to apply to implementations, and $M_{spec} = M_r \cup M_m$ is the set of mutations to apply to specifications. Let $M = M_a \cup M_m \cup M_r$. For a circuit $\mathcal{C}$ and a mutation μ, let $\mathcal{C}_\mu$ denote the mutant circuit obtained from $\mathcal{C}$ by applying μ.

We study and classify mutations to circuits with respect to the following properties.

1. **Duality:** Mutations $\mu \in M_{imp}$ and $\nu \in M_{spec}$ are *dual*, denoted $dual(\mu, \nu)$, if for all implementations $\mathcal{I}$ and specifications $\mathcal{S}$, we have that $\mathcal{I}_\mu \models \mathcal{S}$ iff $\mathcal{I} \models \mathcal{S}_\nu$.
   We also consider single-sided versions of the definition: If $\mathcal{I}_\mu \models \mathcal{S}$ implies $\mathcal{I} \models \mathcal{S}_\nu$, then ν is *specification dual* to μ. Also, if $\mathcal{I} \models \mathcal{S}_\nu$ implies $\mathcal{I}_\mu \models \mathcal{S}$, then μ is *implementation dual* to ν.
2. **Invertability:** Mutation $\nu \in M$ is the *inverse* of mutation $\mu \in M$ if for all circuits $\mathcal{C}$, we have that $(\mathcal{C}_\mu)_\nu = \mathcal{C}$.
   We also consider single-sided versions of the definition: Consider mutations $\nu \in M_{spec}$ and $\mu \in M_{imp}$. If for all circuits $\mathcal{C}$, we have that $\mathcal{C} \models (\mathcal{C}_\mu)_\nu$, then ν *decreases* μ. Also, if for all circuits $\mathcal{C}$, we have that $(\mathcal{C}_\nu)_\mu \models \mathcal{C}$, then μ *increases* ν.
3. **Monotonicity:** Mutation $\mu \in M$ is *monotone* if for all circuits $\mathcal{I}$ and $\mathcal{S}$, if $\mathcal{I} \models \mathcal{S}$ then $\mathcal{I}_\mu \models \mathcal{S}_\mu$.

The following theorem states useful relations amongst the above properties.

Theorem 4.1: Consider two mutations $\mu \in M_{imp}$ and $\nu \in M_{spec}$.

1. **[Specification duality]** If ν is monotone, and ν decreases μ, then ν is specification dual to μ.
2. **[Implementation duality]** If μ is monotone, and μ increases ν, then μ is implementation dual to ν.
3. **[Duality]** If μ and ν are monotone and inverse of each other, then they are dual.

Proof: We start with the first claim. We have to prove that for all implementations $\mathcal{I}$ and specifications $\mathcal{S}$, we have

that $\mathcal{I}_\mu \models \mathcal{S}$ implies $\mathcal{I} \models \mathcal{S}_\nu$. Assume that $\mathcal{I}_\mu \models \mathcal{S}$. By monotonicity, $(\mathcal{I}_\mu)_\nu \models \mathcal{S}_\nu$. Since ν decreases μ, we have $\mathcal{I} \models (\mathcal{I}_\mu)_\nu$. Hence, by transitivity of $\models$, also $\mathcal{I} \models \mathcal{S}_\nu$.

The proof of the second claim is similar. We have to prove that for all implementations $\mathcal{I}$ and specifications $\mathcal{S}$, we have that $\mathcal{I} \models \mathcal{S}_\nu$ implies $\mathcal{I}_\mu \models \mathcal{S}$. Assume that $\mathcal{I} \models \mathcal{S}_\nu$. By monotonicity, $\mathcal{I}_\mu \models (\mathcal{S}_\nu)_\mu$. Since μ increases ν, we have that $(\mathcal{S}_\nu)_\mu \models \mathcal{S}$. Hence, by transitivity of $\models$, also $\mathcal{I}_\mu \models \mathcal{S}$.

By definition, if μ is the inverse of ν, then μ both increases and decreases ν. Thus, the third claim follows from the first two claims. ■

The conditions in the above theorem are sufficient for duality, but they are not necessary. To see this, let $\mu \in M_{imp}$ be the mutation that adds behaviors that dualize the value of a signal x. Also, let $\nu \in M_{spec}$ be the mutation DEP_x that removes behaviors that depend on signal x. In [15], the authors show that ν is specification dual to μ. On the other hand, while μ is monotone, it does not increase ν. Indeed $(\mathcal{C}_\nu)_\mu = \mathcal{C}_\nu$, since if we remove all behaviors dependent on x, there is nothing to add back.

We now illustrate the use of the above theorem in proving duality results about mutations introduced in Section III-B.

For lack of space, we omit the full proofs of both propositions. The proofs are based on showing that the studied mutation μ is monotone and dual to itself (Proposition 4.2) or that the studied mutations μ and ν are monotone and increase or decrease each other (Proposition 4.3).

Proposition 4.2: **[Flipping the value of a signal is self-dual]** Let μ be FLIP_x, for $x \in I$. The mutation μ is dual to itself.

Proof sketch: Let μ be FLIP_x, for $x \in I$. The mutation μ is clearly the inverse of itself. Also, it is easy to see that μ is monotone, because applying a flip to both the implementation and specification changes their runs in exactly the same way, so that trace containment or simulation is maintained.

Thus μ is dual to itself. □

In earlier work [15], it was shown that FLIP_x is also self-dual for $x \in O$ and $x \in C$ under certain reasonable conditions. The above proposition is just an illustration of the use of the newly defined properties of mutations.

Recall that it is possible to control the temporal pattern in which mutations are applied. For example, we can flip the value of x at exactly all even positions or at specific cycles. When the pattern is deterministic (as in "flip in all even positions" but not as in "flip exactly once at non-deterministically chosen cycle"), then monotonicity and invertability are maintained, and so is the duality.

Proposition 4.3: **[Conditional stuck-at is dual]** Consider a signal $x \in O$ and a formula $\beta \in \mathcal{B}(C)$. Let μ be β_STICK_x_AT_1 and ν be β_STICK_x_AT_0. If $\rho_x \rightarrow \neg\beta$ and β is over observable control signals, then ν is specification dual to μ. Similarly, if $\beta \rightarrow \rho_x$ and β is over observable control signals, then μ is implementation dual to ν.

Proof sketch: It is easy to see that ν is the inverse of μ when $\rho_x \rightarrow \neg\beta$ and μ is the inverse of ν when $\beta \rightarrow \rho_x$: this is obtained by syntactically evaluating $((\rho_x)_\mu)_\nu$ and $((\rho_x)_\nu)_\mu$ and verifying that they simplify to ρ_x.

The monotonicity follows from the crucial observation that the implementation and specification agree on the values of the observable control signals, and hence on the values of β at each step. Hence, they will continue to agree on the values of observable signals even after mutation by ν (for specification duality) or μ (for implementation duality). □

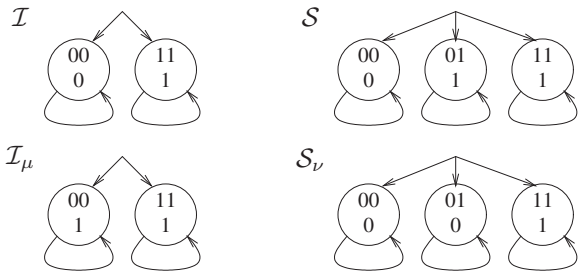

Fig. 2. Duality for conditional stuck-at mutation.

Note that, in the first case, if a designer has in mind a condition γ that is not disjoint from ρ_x, he can take $\beta = \gamma \wedge \neg\rho_x$ as the condition that would satisfy $\rho_x \rightarrow \neg\beta$. Likewise, in the second case, if a designer has in mind a condition γ that does not imply ρ_x, he can take $\beta = \gamma \wedge \rho_x$ as the condition that satisfies $\beta \rightarrow \rho_x$.

Example 1: Consider the implementation $\mathcal{I}$ and specification $\mathcal{S}$ in Figure 2. In both, $I = \emptyset$, $C = \{y, z\}$, and $O = \{x, y\}$. In the figure, the upper line describe the values of y and z, and the bottom line describes the value of x. In the specification, the labeling function $\rho_x^{\mathcal{S}}$ of x is $y \vee z$, whereas in the implementation, $\rho_x^{\mathcal{I}}$ is $y \wedge z$. It is easy to see that $\mathcal{I}$ satisfies $\mathcal{S}$.

Assume we apply to $\mathcal{I}$ a mutation μ that sticks x to 1 whenever $y = 0$. Thus, $\beta = \neg y$ and in the mutated implementation $\mathcal{I}_\mu$, we have that $(\rho_x^{\mathcal{I}})_\mu$ is $(y \wedge z) \vee \neg y$. Note that $\mathcal{I}_\mu \models \mathcal{S}$ and, as required, $\rho_x^{\mathcal{I}} \rightarrow \neg\beta$ and β is over observable control signals. Thus, the mutation ν that replaces $\rho_x^{\mathcal{S}}$ by $\rho_x^{\mathcal{S}} \wedge \neg\beta$ is specification-dual to μ. In the mutated specification $\mathcal{S}_\nu$, we have that $(\rho_x^{\mathcal{S}})_\nu$ is $(y \vee z) \wedge \neg\neg y = y$, and, as guaranteed from the duality, $\mathcal{I} \models \mathcal{S}_\nu$.

Remark 1: Let μ and ν be dual mutations. Suppose that $\mathcal{I}_\mu \models \mathcal{S}$. Then, we know that $\mathcal{I} \models \mathcal{S}_\nu$. What do we know about the satisfaction of $\mathcal{S}_\nu$ in $\mathcal{I}_\mu$? It turns out that for some dual mutations, $\mathcal{I}_\mu \models \mathcal{S}_\nu$ is equivalent to $\mathcal{I} \models \mathcal{S}$ (for example, when $\mu = \nu =$FLIP_x), for some it "doubles the effect" of the mutation (for example, when μ is DELAY_k and ν is PREMATURE_k), and for some it is equivalent to checking $\mathcal{I}_\mu \models \mathcal{S}$ or $\mathcal{I} \models \mathcal{S}_\nu$ (for example, when μ is FREE_x and ν is DEP_x), and for some, it cannot be that $\mathcal{I}_\mu \models \mathcal{S}_\nu$ (for example, in conditional stuck at). Thus, it is impossible to come up with a general recommendation about such a check. □

V. Aggressiveness Order Between Mutations

We now present a set of aggressiveness orders between many of the mutations we have presented in this paper. These orders help to organize the space of mutations, and are especially useful in reasoning about fault-tolerant circuits, as we will demonstrate in Section VI.

The aggressiveness order between a pair of mutations is a partial order parameterized by a *polarity*. The polarity indicates whether the mutations are applied to the implementation

(in which case, the mutations are in M_{imp}, and the more behaviors the mutation adds, the more aggressive it is) or to the specification (in which case, the mutations are in M_{spec} and the more behaviors the mutation removes, the more aggressive it is). We first define the aggressiveness order and study its properties, and then describe some natural orders.

Definition 1: **[Aggressiveness order for implementations]** Let μ and ν be mutations in M_{imp}. We say that μ is *more implementation-aggressive than* ν, denoted $\mu \geq_{imp} \nu$, if for every implementation $\mathcal{I}$ and specification $\mathcal{S}$, we have that $\mathcal{I}_\mu \models \mathcal{S}$ implies $\mathcal{I}_\nu \models \mathcal{S}$.

Definition 2: **[Aggressiveness order for specifications]** Let μ and ν be mutations in M_{spec}. We say that ν is *more specification-aggressive than* μ, denoted $\nu \geq_{spec} \mu$, if for every implementation $\mathcal{I}$ and specification $\mathcal{S}$, we have that $\mathcal{I} \models \mathcal{S}_\nu$ implies $\mathcal{I} \models \mathcal{S}_\mu$.

Intuitively, $\mu \geq_{imp} \nu$ means that the mutation μ makes it harder for the implementation to satisfy the specification. Dually, $\nu \geq_{spec} \mu$ means that the mutation ν makes it harder for the specification to be satisfied by the implementation. Formally, we have the following.

Theorem 5.1: Consider mutations μ and ν. The following are equivalent: (1) For every circuit $\mathcal{C}$, we have $C_\nu \models \mathcal{C}_\mu$ (2) $\mu \geq_{imp} \nu$ (3) $\nu \geq_{spec} \mu$.

Proof: We first prove that (1) implies (2) and (3). Consider an implementation $\mathcal{I}$ and specification $\mathcal{S}$. Assume that $\mathcal{I}_\mu \models \mathcal{S}$. By the assumption (1), we have that $\mathcal{I}_\nu \models \mathcal{I}_\mu$. Now, transitivity of $\models$ implies that $\mathcal{I}_\nu \models \mathcal{S}$, thus $\mu \geq_{imp} \nu$. Assume now that $\mathcal{I} \models \mathcal{S}_\nu$. By (1), we have $\mathcal{S}_\nu \models \mathcal{S}_\mu$. The transitivity of $\models$ implies that $\mathcal{I} \models \mathcal{S}_\mu$, thus $\nu \geq_{spec} \mu$.

We now prove that (2) implies (1). For a circuit $\mathcal{C}$, we refer to $\mathcal{C}$ as an implementation and to $\mathcal{C}_\mu$ as its specification. By the assumption (2), if $\mathcal{C}_\mu \models \mathcal{C}_\mu$, then $\mathcal{C}_\nu \models \mathcal{C}_\mu$. Hence, by the reflexivity of $\models$, we have $\mathcal{C}_\nu \models \mathcal{C}_\mu$.

It is left to prove that (3) implies (1). For a circuit $\mathcal{C}$, we refer to $\mathcal{C}_\nu$ as an implementation and to $\mathcal{C}$ as its specification. By the assumption (3), if $\mathcal{C}_\nu \models \mathcal{C}_\nu$, then $\mathcal{C}_\nu \models \mathcal{C}_\mu$. Hence, by the reflexivity of $\models$, we have $\mathcal{C}_\nu \models \mathcal{C}_\mu$. ■

Aggressiveness orders arise in a few different ways. We explore each of these ways in a separate subsection below.

A. Syntactic Aggressiveness

Syntactic aggressiveness involves mutations that change the syntactic structure of a circuit, such as deleting transitions, or changing conditionals.

We say that a mutation μ is *clean* if it only adds and/or removes transitions of the circuit. That is, for every circuit $\mathcal{C}$, there is a pair $\langle a_\delta, r_\delta \rangle$ of functions $a_\delta, r_\delta : 2^C \times 2^I \to 2^{2^C}$ such that the circuit obtained form $\mathcal{C}$ by applying μ is $\mathcal{C}' = \langle I, O, C, \theta, \delta', \rho \rangle$, where for all $i \in 2^I$ and $s \in 2^C$, we have that $\delta'(s,i) = \delta(s,i) \cup a_\delta(s,i) \setminus r_\delta(s,i)$. Thus, $\mathcal{C}'$ is obtained from $\mathcal{C}$ by increasing the nondeterminism in δ according to a_δ and them decreasing it according to r_δ. Note that the application of μ may result in a circuit that is not receptive.[3]

Intuitively, the more transitions a mutation adds or removes, the more aggressive it is. Formally, we have to consider also the polarity of the aggressiveness, and we have the following. Consider two clean mutations $\mu = \langle a_\delta, r_\delta \rangle$ and $\nu = \langle a'_\delta, r'_\delta \rangle$. If $a'_\delta \subseteq a_\delta$ and $r_\delta \subseteq r'_\delta$, then we say that μ is *syntactically more implementation-aggressive than* ν. Dually, if $a_\delta \subseteq a'_\delta$ and $r'_\delta \subseteq r_\delta$, then we say that ν is *syntactically more specification-aggressive than* μ.

Proposition 5.2: **[Syntactic aggressiveness implies aggressiveness]** If μ is syntactically more implementation-aggressive than ν, or ν is syntactically more specification-aggressive than μ, then $\mu \geq_{imp} \nu$ and $\nu \geq_{spec} \mu$.

Proof: Assume that μ is syntactically more implementation-aggressive than ν, or that ν is syntactically more specification-aggressive than μ. Thus, $a'_\delta \subseteq a_\delta$ and $r_\delta \subseteq r'_\delta$, or $a_\delta \subseteq a'_\delta$ and $r'_\delta \subseteq r_\delta$. It is easy to see that in both cases, for every circuit $\mathcal{C}$, we have $\mathcal{C}_\nu \models \mathcal{C}_\mu$. Indeed, in the branching approach, we can restrict the identity relation to an initial simulation, and in the linear approach all the computations of $\mathcal{C}_\nu$ have matching computations of $\mathcal{C}_\mu$. Hence, by Theorem 5.1, $\mu \geq_{imp} \nu$ and $\nu \geq_{spec} \mu$. ■

Since adding and removing transitions is a clean mutation, and the more we add/remove the more syntactically aggressive we are, Proposition 5.2 implies the following.

Proposition 5.3: If $\beta_1 \to \beta_2$ and $\gamma_1 \to \gamma_2$, then (β_2, γ_2)_ADD $\geq_{imp}$ (β_1, γ_1)_ADD and (β_2, γ_2)_REMOVE $\geq_{spec}$ (β_1, γ_1)_REMOVE.

B. Orders for Mutations on Signals

We exhibit aggressiveness orders for different kinds of mutations involving signals of the circuit.

The first set of orders are given in the following proposition.

Proposition 5.4:

- DEP_x $\geq_{spec}$ REST_x_TO_0 $\geq_{spec}$ STICK_x_AT_0,
- DEP_x $\geq_{spec}$ REST_x_TO_1 $\geq_{spec}$ STICK_x_AT_1.
- DEP_x $\geq_{spec}$ FLIP_x.
- For all $k \geq 1$, DELAY_LEQ_$k+1$ $\geq_{imp}$ DELAY_LEQ_k.

Proof sketch: Recall that in REST_x_TO_0, we disable transitions to states in which the value of x is not 0, whereas in STICK_x_AT_0, we leave such states reachable but behave as if the value of x in them is 0. Also, in DEP_x, we disable all transitions that depend on the value of x. Hence, it is not hard to prove that for every circuit $\mathcal{C}$, we have $\mathcal{C}_{\text{DEP\_}x} \models \mathcal{C}_{\text{REST\_}x\text{\_TO\_0}} \models \mathcal{C}_{\text{STICK\_}x\text{\_AT\_0}}$ in both the linear and branching approaches (and similarly with REST_x_TO_1 and STICK_x_AT_1). Hence, by Theorem 5.1, the aggressiveness orders follow.

It is also clear that, for all $k \geq 1$, $\mathcal{C}_{\text{DELAY\_LEQ\_k+1}}$ exhibits a superset of behaviors of $\mathcal{C}_{\text{DELAY\_LEQ\_k}}$. Therefore, DELAY_LEQ_$k+1$ $\geq_{imp}$ DELAY_LEQ_k. □

Remark 2: Note that it is not the case that DELAY_$k+1$ $\geq_{imp}$ DELAY_k. Indeed, when the mutation specifies an exact bound, a larger delay may lead to satisfaction of properties that are not satisfied in a smaller delay. A related phenomenon in temporal-logic vacuity is the fact that there is no order between vacuity detection that is done by mutating a single occurrence of a sub-formula and multiple occurrences of it. □

The implementation-aggressiveness order is dual to the specification-aggressiveness order:

Theorem 5.5: If $dual(\mu, \tilde{\mu})$ and $dual(\nu, \tilde{\nu})$, then $\mu \geq_{imp} \nu$ iff $\tilde{\mu} \geq_{spec} \tilde{\nu}$.

[3] In the definition above, we ignored initial transitions described by θ. Extending the definition with $a_\theta, r_\theta : 2^I \to 2^{2^C}$ is straightforward.

Proof: Assume that $\mu \geq_{imp} \nu$. Consider an implementation $\mathcal{I}$ and specification $\mathcal{S}$. Assume that $\mathcal{I} \models \mathcal{S}_{\tilde{\mu}}$. By the duality of μ and $\tilde{\mu}$, we have that $\mathcal{I}_{\mu} \models \mathcal{S}$. Since $\mu \geq_{imp} \nu$, it follows that $\mathcal{I}_{\nu} \models \mathcal{S}$. The duality of ν and $\tilde{\nu}$ implies that $\mathcal{I} \models \mathcal{S}_{\tilde{\nu}}$, and we are done. The other direction is dual. ■

It is shown in [15] that FREE_x and DEP_x are dual, and FLIP_x is self-dual. Hence, by Proposition 5.4 and Theorem 5.5, FREE_$x \geq_{imp}$ FLIP_x. Likewise, since DELAY_LEQ_k and PREMATURE_LEQ_k are dual, Theorem 5.5 implies that PREMATURE_LEQ_$k+1 \geq_{spec}$ PREMATURE_LEQ_k.

Note that FREE_$x \geq_{imp}$ STICK_x_AT_1 and FREE_$x \geq_{imp}$ STICK_x_AT_0, because the FREE_x mutation adds behaviors that stick x to 1 as well as those that stick x to 0.

Further, note that the FREE_x mutation is equivalent to the mutation that non-deterministically, at each step, either flips x or doesn't. In other words, FREE_x is identical to allowing an infinite number of SEUs to occur in x. This yields that FREE_$x \geq_{imp}$ k-SEU_x for all $k \geq 1$.

C. Duality and Aggressiveness

For mutations that do not have duals that are independent of the circuit $\mathcal{I}$, it is useful to define an *implementation-specific duality*. This notion has an associated *circuit-dependent aggressiveness order*.

Definition 3: **[Dual aggressiveness]** Consider a mutation $\mu \in M_{imp}$. For a circuit $\mathcal{I}$ and a mutation $\nu_{\mathcal{I}} \in M_{spec}$, we say that $\nu_{\mathcal{I}}$ *dualizes* μ *for* $\mathcal{I}$ if, for all specifications $\mathcal{S}$, we have that $\mathcal{I}_{\mu} \models \mathcal{S}$ implies that $\mathcal{I} \models \mathcal{S}_{\nu_{\mathcal{I}}}$. Then, we say that a mutation $\nu \in M_{spec}$ is *more dual-implementation-aggressive* than μ if for all implementations $\mathcal{I}$ and mutations $\nu_{\mathcal{I}}$ that dualize μ for $\mathcal{I}$, we have $\nu \geq_{spec} \nu_{\mathcal{I}}$.

Example 2: Consider the mutation μ = (β, γ)_ADD. Assume that β and γ are over observable control signals. For a given implementation $\mathcal{I}$, let $\nu_{\mathcal{I}}$ be the mutation that removes exactly all the transitions added by (β, γ)_ADD. Note that since $\mathcal{I}$ may contain transitions from states that satisfy β to states that satisfy γ, the mutation $\nu_{\mathcal{I}}$ does not coincide with the mutation (β, γ)_REMOVE. Indeed, the latter removes all transitions between states that satisfy β to states that satisfy γ, and not only these added by μ. It is easy to see, however, that (β, γ)_REMOVE is syntactically more specification aggressive than $\nu_{\mathcal{I}}$, and hence, by Proposition 5.2, (β, γ)_REMOVE is more specification aggressive than $\nu_{\mathcal{I}}$.

Since β and γ are over observable control signal, the mutation $\nu_{\mathcal{I}}$ is monotonic. Also, since $(\mathcal{I}_{\mu})_{\nu_{\mathcal{I}}} = \mathcal{I}$, it follows that for every specification $\mathcal{S}$, if $\mathcal{I}_{\mu} \models \mathcal{S}$ then $\mathcal{I} \models \mathcal{S}_{\nu_{\mathcal{I}}}$. Thus, $\nu_{\mathcal{I}}$ dualizes μ for $\mathcal{I}$. Hence, as (β, γ)_REMOVE is more specification aggressive than $\nu_{\mathcal{I}}$ for all implementations $\mathcal{I}$ and mutations $\nu_{\mathcal{I}}$, we conclude that (β, γ)_REMOVE is more dual-implementation-aggressive than μ.

For easy reference, we summarize some useful aggressiveness orders in Table I. Each row of the table states an aggressiveness order along with any conditions under which it holds.

D. Coverage for Fault-Tolerant Systems

We now describe how coverage of specifications can be computed for fault-tolerant circuits.

Faults mutate the implementation. Thus, an implementation $\mathcal{I}$ is said to tolerate faults modeled by $\mu_1, \mu_2, \ldots, \mu_k$ in M_{imp} if $\mathcal{I}$ continues to satisfy its specification even in the presence of these faults.

Aggressiveness order	Conditions
DEP_$x \geq_{spec}$ REST_x_TO_0 $\geq_{spec}$ STICK_x_AT_0	none
DEP_$x \geq_{spec}$ REST_x_TO_1 $\geq_{spec}$ STICK_x_AT_1	none
DEP_$x \geq_{spec}$ FLIP_x	none
PREMATURE_LEQ_$k+1 \geq_{spec}$ PREMATURE_LEQ_k	$k \geq 1$
DELAY_LEQ_$k+1 \geq_{imp}$ DELAY_LEQ_k	$k \geq 1$
FREE_$x \geq_{imp}$ STICK_x_AT_1	none
FREE_$x \geq_{imp}$ STICK_x_AT_0	none
FREE_$x \geq_{imp}$ FLIP_x	none
FREE_$x \geq_{imp}$ k-SEU_x	$k \geq 1$
(β_2, γ_2)_ADD $\geq_{imp}$ (β_1, γ_1)_ADD	$\beta_1 \rightarrow \beta_2$
(β_2, γ_2)_REMOVE $\geq_{spec}$ (β_1, γ_1)_REMOVE	and $\gamma_1 \rightarrow \gamma_2$

TABLE I
SUMMARY OF AGGRESSIVENESS ORDERS

Note that by Theorem 5.1, the above definition also means that $\mathcal{I}$ continues to satisfy its specification when less aggressive faults are applied to it.

Formally, we model a *fault-tolerant* system by an implementation $\mathcal{I}$ along with a set of mutations $\mu_1, \mu_2, \ldots, \mu_k$, such that for all specifications $\mathcal{S}$, if $\mathcal{I} \models \mathcal{S}$, then $\mathcal{I}_{\mu_j} \models \mathcal{S}$ for for all $1 \leq j \leq k$.

Measuring the coverage of a fault-tolerance system is challenging, as satisfaction of a specification under mutations (loose satisfaction) need not imply low coverage. Thus, in the context of fault-tolerant systems, we have to redefine loose satisfaction to take the tolerance into account and apply to the system mutations that are more implementation-aggressive than those it tolerates. Formally, we say that a fault tolerant system $\langle \mathcal{I}, \{\mu_1, \mu_2, \ldots, \mu_k\} \rangle$, *loosely satisfies* a specification $\mathcal{S}$, if there is a mutation μ such that $\mu_j \not\geq_{imp} \mu$ for all $1 \leq j \leq k$, and $\mathcal{I}_{\mu} \models \mathcal{S}$.

In particular, if $\mathcal{I}_{\mu}$ satisfies $\mathcal{S}$ and $\mathcal{I}$ is designed to be tolerant only to μ, and further if $\mathcal{I}_{\nu}$ satisfies $\mathcal{S}$ for $\nu \geq_{imp} \mu$, then we say that $\mathcal{S}$ has low coverage.

We will see in the next section how the theory of this section can be applied in practice.

VI. EXPERIMENTAL RESULTS

We now present experimental results demonstrating the application of our theory of mutations in the context of fault-tolerant circuits. We demonstrate how analysis of fault tolerance can also provide vacuity and coverage information. Due to lack of space, we describe specifications using LTL rather than draw the finite-state transducers – the correspondence with LTL is described in [15].[4]

Experiments were performed using the Cadence SMV model checker with option "-absref3" (BDD-based counterexample guided abstraction refinement) on an Intel Core2Duo 2 GHz machine. Each model checking run involved checking whether a possibly-mutated implementation satisfied a possibly-mutated specification. For each experiment on a benchmark, we have an associated set M comprising mutations of interest. For example, to check coverage of a circuit

[4]We note that in order to handle full LTL, one has to consider circuits with fairness, and, accordingly, extend the containment and simulation relations to ones that account for fairness. In our experiments, the only LTL formulas we mutated were safety LTL formulas, for which fairness is not needed.

Benchmark Benchmark	Number of latches	Number of LTL properties	Coverage (%)
Ibuf	6	8	83.3
Lock	9	2	55.6
Am2910	99	4	99.0
SyncArb	48	48	33.3

TABLE II
FLIP_x MUTATION ON VIS BENCHMARKS

with respect to the SEU mutation, the set M comprises an 1-SEU_x mutation for every latch x in the design. Thus, for each benchmark, a coverage experiment would involve the following two steps for each element μ of the set M of mutations: (a) automatically create a new SMV model of the circuit mutated with μ, and then (b) use SMV to check whether the mutated model satisfies its specification.

The scalability of our approach is naturally dependent on the scalability of model checking for the circuits of interest. In our experiments, for any benchmark and set of mutations M, the total run-time was on the order of a few minutes; in fact, for each benchmark listed in Tables II-IV, the total run-time of all experiments performed on that benchmark was less than 500 seconds. We believe that incremental approaches to model checking that re-use work performed in different runs can be quite effective at further reducing run-time, since mutations tend to be local and mutated models tend to be similar.

A. Vacuity from Coverage

The first experiment illustrates how useful vacuity information can be obtained for free once coverage has been checked. A number of benchmarks are selected from the VIS benchmark suite [20]. We use the mutation FLIP_x to investigate the resilience of latches in the circuits with the given specifications. Hence, a latch x is *covered* by a specification $\mathcal{S}$ with respect to FLIP_x if, after mutation by FLIP_x, the circuit fails $\mathcal{S}$.

Table II summarizes our results. The coverage numbers in the table give the percentage of latches which, when mutated, caused the circuit to fail its specification. We note that none of the four circuits have 100% coverage[5]. This suggests that at least some of the latches are resilient to FLIP_x in these circuits. A verification engineer might wonder: why are they resilient? At this stage, the engineer is usually left with two choices. Either she can try to write more specifications and hope they will cover those latches, or she can go back to the circuit designer and investigate whether they were intended to be fault-tolerant. These two approaches are both time-consuming. We propose that we can use *duality* to help answer this question.

Recall from Section IV that the FLIP_x mutation is self-dual. Consider the following LTL specification from the benchmark "Am2910".

$$spec: \ assert\ \mathbf{G}(sp[2..0] = 110 \rightarrow \mathbf{X}(sp[2..0] = 111));$$

In our experiment, applying the FLIP_x mutation to $x = sp[0]$ in the circuit still satisfies the above specification.

[5]The LTL specifications used here are the ones that verified to be true for the original circuit in SMV. Hence the number of specifications may be smaller than the one found in the benchmark suite.

Benchmark	1-SEU_x coverage	20-SEU_x coverage	FREE_x coverage
Ibuf	83.3%	83.3%	100%
Lock	0%	0%	100%
Am2910	99.0%	99.0%	99.0%
SyncArb	33.3%	33.3%	66.7%

TABLE III
COVERAGE OF k-SEU_x VS. FREE_x

Benchmark	STICK_x_AT_0 Coverage	FREE_x Coverage
Ibuf	83.3%	100%
Lock	100%	100%
Am2910	97.0%	99.0%
SyncArb	66.7%	66.7%

TABLE IV
COVERAGE OF STICK_x_AT_0 VS. FREE_x

Thus, applying the dual mutation (FLIP_x again) to the above LTL specification gives us the property below, because the transition from the state 110 goes to 110 after the flip in the 0th bit of sp instead of 111.

$$spec': \ assert\ \mathbf{G}(sp[2..0] = 110 \rightarrow \mathbf{X}(sp[2..0] = 110));$$

The only way that both specifications are satisfied is that they are *vacuously* true, i.e. $sp[2..0]$ never reaches 6. We verified that indeed $\mathbf{G}(\neg sp[2..0] = 110)$ was satisfied.

B. Coverage for Fault-Tolerant Circuits

The second experiment illustrates how the *aggressiveness orders* can be used to analyze coverage in fault-tolerant circuits. We use the same VIS benchmarks from the previous experiment, but apply a 1-SEU_x mutation to each latch x in each benchmark. We also tried applying k-SEU_x for increasing k, as well as the FREE_x mutation. Table III shows our coverage results for three mutations on the VIS benchmarks (we only show $k = 1$ and $k = 20$). As argued in the preceding section, FREE_x $\geq_{imp}$ 20-SEU_x $\geq_{imp}$ 1-SEU_x

We considered specifications of the form $\mathbf{G}(\alpha \rightarrow F\ \beta)$, and found cases where the circuits are tolerant to a large number of of SEUs but not to an infinite number of them (the FREE_x). Hence, we can certify these specifications as being covered by the FREE_x mutation, and the circuit is tolerant to k-SEU_x for any k, but not to the FREE_x mutation. Thus, the FREE_x mutation defines a "lower bound" for the specification.

Observe that for benchmark "Am2910", coverage is not 100% even if we make a latch totally nondeterministic (the FREE_x mutation). This result is due to the vacuity bug found in the previous experiment. A similar result holds for the "SyncArb" benchmark: here the reason for less than 100% coverage of FREE_x is that one-third of the latches involve latched inputs (request lines) which can be changed arbitrarily by the environment in any case.

Similarly, we analyzed the coverage of fault-tolerance to STICK_x_AT_0 using FREE_x since we know that FREE_x $\geq_{imp}$ STICK_x_AT_0. Table IV shows the results for the same benchmarks. For benchmarks "Ibuf" and "Am2910", there are latches resilient to STICK_x_AT_0 but not FREE_x.

In general, the aggressiveness order helps pinpoint the *fault-tolerance boundary* of a system by finding faults that the system tolerates as well as more aggressive faults that the

system does not tolerate. Designers and verification engineers can use information about this boundary to either optimize the circuit or evaluate the quality of their specifications.

C. Improving Specifications

We next show how our methodology can lead to improved specifications. Consider once again, the CMP router mentioned in Section II. For simplicity, again assume that $N = 2$. We started with a specification $\mathcal{S}$ asserting that if both channels issue a request and Channel 1 has priority over Channel 0, then it is not the case that both channels are granted access in the next cycle. Thus, $\mathcal{S} = \mathbf{G}\left(((\vec{r} = 11) \wedge \neg p_{01}) \rightarrow \mathbf{X}\left(\neg(\vec{g} = 11)\right)\right)$.

The form of this LTL property ($\mathbf{G}(\alpha \rightarrow \mathbf{X}\alpha')$) indicates that modifying transitions might be a reasonable form of mutation. We start by trying (β, γ)-REMOVE where $\beta = ((\vec{r} = 11) \wedge \neg p_{01})$ and $\gamma = (\vec{g} = 00)$. Denote this mutation by ν_1. The resulting specification $\mathcal{S}_1 = \mathbf{G}\left(((\vec{r} = 11) \wedge \neg p_{01}) \rightarrow \mathbf{X}\left((\vec{g} = 01) \vee (\vec{g} = 10)\right)\right)$ continues to be satisfied. Continuing, we can apply the same kind of mutation, with the same β but $\gamma = (\vec{g} = 01)$ and find that the resulting property $\mathcal{S}_2$ is also satisfied. Let the mutation we applied in this step be denoted by ν_2; we note that $\mathcal{S}_2$ is obtained by mutating $\mathcal{S}$ with the composition $\nu_1 \circ \nu_2$. (However, if we attempt to use $\gamma = (\vec{g} = 10)$, this is not satisfied.) We now obtain $\mathcal{S}_2 = \mathbf{G}\left(((\vec{r} = 11) \wedge \neg p_{01}) \rightarrow \mathbf{X}\left(\vec{g} = 10\right)\right)$ which looks like what we want. However, we can tighten it still further: by again using the same kind of mutation, but with $\beta = (r_1 = 1)$ (i.e., the second channel need not make a request). Denote this mutation by ν_3. The resulting mutated specification is still satisfied, giving us our final result: $\mathcal{S}_3 = \mathbf{G}\left(((r_1 = 1) \wedge \neg p_{01}) \rightarrow \mathbf{X}\left(\vec{g} = 10\right)\right)$.

It is easy to see that the mutations we applied satisfy the aggressiveness order $\nu_1 \circ \nu_2 \circ \nu_3 \geq_{spec} \nu_1 \circ \nu_2 \geq_{spec} \nu_1$.

VII. Discussion

We have given a novel theory of mutations in this paper, and demonstrated its usefulness in reasoning about vacuity, coverage, and fault-tolerance in a unified way. We conclude with a brief discussion of other potential applications of our theory.

The results of this paper are also relevant for *tightening specifications to catch bugs*. Consider an implementation $\mathcal{I}$ that satisfies its specification $\mathcal{S}$. The ultimate goal of checking coverage of $\mathcal{I}$ by $\mathcal{S}$ is to find errors in $\mathcal{I}$. We suggest two ways to use the aggressiveness order in order to automatically generate a specification that reveals errors. The first is to apply to $\mathcal{S}$ a mutation ν' that is more specification aggressive than ν. If, for example, μ introduces some delay, ν' introduces a bigger prematureness. This may be done automatically, or with user guidance, especially in mutations that involve guards. For example, if ν is (β, γ)-REMOVE, then ν' can be (β', γ')-REMOVE, for $\beta' \rightarrow \beta$ and/or $\gamma' \rightarrow \gamma$. The user is aware of the fact that $\mathcal{I}$ loosely satisfies $\mathcal{S}$ with the mutation (β, γ)-ADD, and can use this awareness in order to come up with the stronger β' and γ'. The second approach is to use dual aggressiveness and apply a mutation that is more dually-aggressive than the one with which loose satisfaction has been detected. Note that in this case, as demonstrated in Example 2, the mutation with which loose satisfaction has been detected need not have a a dual mutation.

We also note that tightening of specifications *can be applied also in the context of assume-guarantee reasoning*. There, a specification for a component is of the form $\langle \varphi, \psi \rangle$, and the component has to satisfy a guarantee ψ under the assumption that its environment satisfies φ. Clearly, the tighter φ is, the more likely it is to imply ψ. Our mutations offer an automatic tightening: if $\langle \varphi, \psi \rangle$ does not hold, we can use mutations on the environment in order to automatically generate a tighter assumption φ_μ for which $\langle \varphi_\mu, \psi \rangle$ does hold. Note that this application is to prove properties, rather than to detect errors.

Acknowledgments. The authors gratefully acknowledge the support of the Gigascale Systems Research Focus Center, one of five research centers funded under the Focus Center Research Program, a Semiconductor Research Corporation program. This work was done while the first author was at UC Berkeley.

References

[1] R. Armoni, L. Fix, A. Flaisher, O. Grumberg, N. Piterman, A. Tiemeyer, and M. Vardi. Enhanced vacuity detection for linear temporal logic. In *Proc 15th CAV*, 2003.

[2] I. Beer, S. Ben-David, C. Eisner, and Y. Rodeh. Efficient detection of vacuity in ACTL formulas. *FMSD*, 18(2):141–162, 2001.

[3] M. Browne, E. Clarke, and O. Grumberg. Characterizing finite Kripke structures in propositional temporal logic. *TCS*, 59:115–131, 1988.

[4] D. Bustan, A. Flaisher, O. Grumberg, O. Kupferman, and M. Vardi. Regular vacuity. In *CHARME*, LNCS 3725, pages 191–206, 2005.

[5] M. Chechik and A. Gurfinkel. Extending extended vacuity. In *Proc. 5th FMCAD*, LNCS 3312, pages 306–321, 2004.

[6] H. Chockler, J. Halpern, and O. Kupferman. What causes a system to satisfy a specification? CoRR cs.LO/0312036, 2003.

[7] H. Chockler, O. Kupferman, R. Kurshan, and M. Vardi. A practical approach to coverage in model checking. In *13th CAV*, LNCS 2102, pages 66–78, 2001.

[8] H. Chockler, O. Kupferman, and M. Vardi. Coverage metrics for temporal logic model checking. In *Proc. 7th TACAS*, LNCS 2031, pages 528 – 542, 2001.

[9] E. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 1999.

[10] S. Das, A. Banerjee, P. Basu, P. Dasgupta, P. P. Chakrabarti, C. R. Mohan, and L. Fix. Formal methods for analyzing the completeness of an assertion suite against a high-level fault model. VLSI Design 2005, pages 201–206.

[11] F. Fummi, G. Pravadelli, A. Fedeli, U. Rossi, and F. Toto. On the use of a high-level fault model to check properties incompleteness. In MEMOCODE 2003, pages 145–152.

[12] D. Große, U. Kühne, and R. Drechsler. Estimating functional coverage in bounded model checking. In *Design Automation and Test in Europe* (DATE), 2007, pages 1176–1181.

[13] O. Grumberg and D. Long. Model checking and modular verification. *ACM TOPLAS*, 16(3):843–871, 1994.

[14] Y. Hoskote, T. Kam, P.-H. Ho, and X. Zhao. Coverage estimation for symbolic model checking. In *Proc. 36th DAC*, pages 300–305, 1999.

[15] O. Kupferman, W. Li, and S. A. Seshia. On the duality between vacuity and coverage. Technical report UCB/EECS-2008-26, EECS Department, UC Berkeley, March 2008.

[16] O. Kupferman and M. Vardi. Vacuity detection in temporal model checking. *Software Tools for Technology Transfer*, 4(2):224–233, 2003.

[17] S. Nain and M. Vardi. Branching vs. linear time: Semantical perspective. In *5th ATVA*, LNCS 4762, 2007.

[18] L.-S. Peh. *Flow Control and Micro-Architectural Mechanisms for Extending the Performance of Interconnection Networks*. PhD thesis, Stanford University, August 2001.

[19] S. A. Seshia, W. Li, and S. Mitra. Verification-guided soft error resilience. In *Design Automation and Test in Europe* (DATE), pages 1442–1447. ACM Press, 2007.

[20] VIS verification benchmarks. Available at `ftp://vlsi.colorado.edu/pub/vis/vis-verilog-models-1.0.tar.gz`.