On the Duality between Vacuity and Coverage

Orna Kupferman
Wenchao Li
Sanjit A. Seshia

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2008-26
http://www.eecs.berkeley.edu/Pubs/TechRpts/2008/EECS-2008-26.html

March 31, 2008

On the Duality between Vacuity and Coverage

Orna Kupferman*† Wenchao Li* Sanjit A. Seshia*

* Department of Electrical Engineering and Computer Sciences
University of California, Berkeley
Berkeley, CA 94720-1770
sseshia@eecs.berkeley.edu, wenchao@berkeley.edu

† School of Computer Science and Engineering
Hebrew University
Jerusalem 91904, Israel
orna@cs.huji.ac.il

March 31, 2008

Abstract

Sanity checks such as vacuity and coverage are used to evaluate the quality of both implementations and specifications. We show formally that vacuity and coverage are *dual* concepts, studying them in a setting in which both the implementation and the specification are given by circuits. To formalize the duality, we present a range of mutations that one can apply to a circuit and partition them into mutations that add, remove, and modify behaviors. Many mutations correspond to physical and design faults, such as ones in which signals are ignored, flipped, delayed, or stuck at a value, and combinations thereof. For most of the mutations, we exhibit corresponding mutations also in the case where the specification is given as a temporal logic formula. We introduce and study the notion of *dual mutations*. A mutation μ that adds or modifies behaviors is dual to a mutation $\tilde{\mu}$ that removes or modifies behaviors if, for all implementations $\mathcal{I}$ and specifications $\mathcal{S}$, satisfaction of $\mathcal{S}$ by a mutant implementation $\mathcal{I}_\mu$, obtained from $\mathcal{I}$ by applying μ, is related to satisfaction by $\mathcal{I}$ of a mutant specification $\mathcal{S}_{\tilde{\mu}}$, obtained from $\mathcal{S}$ by applying $\tilde{\mu}$. Thus, the low coverage of $\mathcal{I}$ by $\mathcal{S}$, which causes $\mathcal{I}_\mu$ to satisfy $\mathcal{S}$, is related to the vacuous satisfaction of $\mathcal{S}$ by $\mathcal{I}$, which causes $\mathcal{I}$ to satisfy $\mathcal{S}_{\tilde{\mu}}$. The notion of dual mutations also applies in a setting in which the specification is a temporal logic formula.

Beyond the clean theoretical picture that the duality suggests, it offers important applications. First, we obtain new coverage metrics and new definitions of vacuity that have so far been used only in one of the sanity checks. Second, when low coverage is detected with a mutation, a tighter specification can be automatically obtained by applying its dual mutation to the original specification. We present experimental results showing the relevance of tightening specifications to self-checking circuits.

1 Introduction

Model checking verifies the correctness of a system with respect to its specification [13]. One of the advantages of model-checking tools is their ability to accompany a negative answer to the correctness query by an erroneous execution of the system. Such counterexamples to the satisfaction of the specification in the system are very important since they can help detect subtle errors in complex designs [12]. On the other hand, when the answer to the correctness query is positive, most model-checking tools provide no additional information. Since a positive answer means that the system is correct with respect to the specification, this seems like a reasonable policy. However, there has been growing awareness to the importance of also challenging positive answers of model-checking tools. One major reason

is the possibility of errors in the modeling of the system or the specification. The goal of *sanity checks* is to detect such errors by further automatic reasoning. Two leading sanity checks are *vacuity* and *coverage*.

In vacuity, the goal is to detect cases where the system satisfies the specification in some unintended trivial way. For example, verifying a system with respect to the specification $\varphi = G(req \rightarrow Fgrant)$ ("every request is eventually followed by a grant"), one should distinguish between satisfaction of φ in systems in which requests are sometimes sent and satisfaction in systems in which requests are never sent. Evidently, the second type of satisfaction suggests some unexpected properties of the system, namely the absence of behaviors in which the precondition is satisfied. Typical definitions of vacuity are based on mutations applied to the specification, where the goal is to identify components of the specification that do not play a role in the verification process. In the above example, the sub-formula $Fgrant$ does not play a role in the satisfaction of φ in a system in which requests are never sent, thus such systems would satisfy even the specification obtained from φ by replacing $Fgrant$ by $false$. Work on vacuity focuses on finding good definitions for vacuous satisfaction for a variety of specification formalisms, and for developing efficient algorithms for detection of vacuous satisfaction [3, 19, 2, 7, 6].

In coverage, the goal is to increase the exhaustiveness of the specification by detecting components of system behavior that do not play a role (i.e., are "not covered") in the verification process. 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. Coverage has roots in simulation-based verification. There, the system is checked with respect to some input vectors [4], and it is crucial to measure the exhaustiveness of the input vectors that are checked. Extensive research in the simulation-based verification community has led to numerous coverage metrics (see the survey by Tasiran and Keutzer [26]). Measuring the exhaustiveness of a specification in model checking ("must more properties be checked?") has a similar flavor as measuring the exhaustiveness of the input vectors in simulation-based verification ("do more vectors have to be checked?"). Nevertheless, while for simulation-based verification it is clear that coverage corresponds to activation during the execution on the input sequence, it is less clear what coverage should correspond to in model checking. Early work on coverage in model checking [16, 10, 9] involved applying mutations to the system, developing efficient algorithms for measuring coverage, and suggesting methods to return useful information to the user, so that he or she might use this information in order to improve the specification. In our example, a mutant system in which one of the two grants is removed still satisfies φ, implying that only one grant is covered. The user can then check that there is a correspondence between the requests and the grants.

Vacuity and coverage have a lot in common, as noted previously [17]. 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. In this paper, we formally relate coverage and vacuity. We show in a precise sense that they are dual, and present useful applications of the duality. The contributions of our paper are further elaborated below.

Formalizing the duality between vacuity and coverage. We make the first formal connection between vacuity and coverage, working in a setting in which both the implementation and the specification are given by circuits (finite-state transducers), at different levels of abstraction. Having the same formalism for the implementation and the specification, we can formally relate mutations that are applied to either of them, and thus formally relate vacuity and coverage.[1]

We consider a wide range of mutations that remove, add, or modify behaviors, many of which are considered for the first time in the context of vacuity and coverage. Many of the mutations correspond to actual physical and design faults (e.g., signals ignored, flipped, delayed, or being stuck at a value), and combinations thereof. Consider an implementation $\mathcal{I}$ and a specification $\mathcal{S}$. We say that $\mathcal{I}$ satisfies $\mathcal{S}$ *vacuously* if $\mathcal{I}$ also satisfies a mutant specification $\mathcal{S}'$ that has fewer behaviors than $\mathcal{S}$. Intuitively, as in temporal-logic-based vacuity, it means that some components of $\mathcal{S}$ could have been tighter. We say that $\mathcal{I}$ satisfies $\mathcal{S}$ *loosely* if $\mathcal{S}$ is also satisfied in a mutant implementation $\mathcal{I}'$ that has more behaviors than $\mathcal{I}$. Intuitively, as in coverage, it means that some components of the implementation could have been more flexible and the specification would still have been satisfied. Finally, $\mathcal{I}$ satisfies $\mathcal{S}$ *diversely* if $\mathcal{S}$ is also satisfied in a mutant implementation $\mathcal{I}'$ in which some behaviors of $\mathcal{I}$ are modified, or if $\mathcal{I}$ satisfies a mutant specification $\mathcal{S}'$ in which some behaviors of $\mathcal{S}$ are modified. Intuitively, again, the verification process was not tight. We analyze vacuous, loose, and diverse satisfaction for each of the presented mutations, and demonstrate that they can all point to errors in the modeling of the system or to an incomplete specification.

[1] Coverage metrics for the case where specifications are given by circuits are studied in [8]. The focus there, however, is on efficient calculation of a particular metric, and the contribution is orthogonal to the one described here.

An important concept we introduce is that of a *dual mutation*, which serves to precisely formalize the duality between coverage and vacuity. A mutation μ that adds or modifies behaviors is dual to a mutation $\tilde{\mu}$ that removes or modifies behaviors if, for all implementations $\mathcal{I}$ and specifications $\mathcal{S}$, satisfaction of $\mathcal{S}$ by a mutant implementation $\mathcal{I}_\mu$ obtained from $\mathcal{I}$ by applying μ is related to satisfaction by $\mathcal{I}$ of a mutant specification $\mathcal{S}_{\tilde{\mu}}$ obtained from $\mathcal{S}$ by applying $\tilde{\mu}$. Thus, loose satisfaction $\mathcal{S}$ by $\mathcal{I}$, which is reflected in the satisfaction of $\mathcal{S}$ by $\mathcal{I}_\mu$, is related to vacuous satisfaction of $\mathcal{S}$ by $\mathcal{I}$, which is reflected in the satisfaction of $\mathcal{S}_{\tilde{\mu}}$ by $\mathcal{I}$. For example, we show that a mutant implementation $\mathcal{I}_\mu$ in which the value of a control signal x is always flipped satisfies a specification $\mathcal{S}$ iff the implementation $\mathcal{I}$ satisfies a mutant specification $\mathcal{S}_{\tilde{\mu}}$ in which the value of x is flipped.

Having the same formalism for the implementation and the specification makes the establishment of the dualities cleaner. In practice, designers often prefer to work with specifications that are given in terms of temporal-logic formulas. We show that the mutations and the notion of notion of duality are carried over also to a setting in which the implementation is given by a circuit and the specification is given by a temporal-logic formula. Technically, this is done by showing how most of the mutations we describe for circuits have corresponding mutations in the temporal-logic setting. For example, rather than always flipping the value of an observable signal x in a circuit, we can negate all its occurences in the formula.

Applications of the duality. Beyond the clean theoretical picture that the duality provides, it enables two important applications. First, the duality leads to improved definitions of vacuity and coverage, solving, for example, the problem of detecting low coverage in universal specifications.

The second application has to do with the challenge of using coverage information in order to automatically generate improved specifications. Existing approaches to coverage metrics return information about components that are not covered. The way from this information to a new and improved specification is still long. Using the duality between vacuity and coverage, we can easily and automatically derive a specification that is tighter than the original specification and that is still satisfied by the implementation. Indeed, whenever satisfaction of $\mathcal{S}$ by a mutant $\mathcal{I}_\mu$ implies satisfaction of a (dual) mutant $\mathcal{S}_{\tilde{\mu}}$ by $\mathcal{I}$, we can let $\mathcal{S}_{\tilde{\mu}}$ serve as a tighter specification. In a similar way, vacuous satisfaction is typically caused by absence of expected behaviors in the implementation. Applying to the implementation a mutation that is dual to the one with which vacuity has been detected adds such behaviors.

We discuss both applications in Section 5. There, we demonstrate how improvement of specifications can be used in the synthesis of circuits that check themselves for the occurrence of physical faults: a fault can be thought of as a mutation μ, and duality can be used to generate a specification $\mathcal{S}_\nu$ that is not satisfied by $\mathcal{I}_\mu$.

2 Definitions

A sequential circuit (*circuit*, for short) 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.[2] Accordingly, $\theta : 2^I \rightarrow 2^{2^C} \setminus \emptyset$ 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, and $\rho : 2^C \rightarrow 2^O$ is an output function that maps every state to the output signals that hold in it. It is required that $I \cap C = I \cap O = \emptyset$. Possibly $O \cap C \neq \emptyset$, in which case for all $x \in O \cap C$ and $s \in 2^C$, we have $x \in s$ iff $x \in \rho(s)$. Thus, $\rho(s)$ agrees with s on signals in C. 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 that it never gets stuck. When $|\theta(s)| = 1$ and $|\delta(s, i)| = 1$ for all $s \in 2^C$ and $i \in 2^I$, we say that $\mathcal{C}$ is deterministic. Nondeterminism in θ and δ reflects abstraction, and would be used when $\mathcal{C}$ models a specification. Then, the set I of inputs may be a strict subset of a richer set of inputs (with respect to which θ and δ are deterministic). For example, if the full set of inputs is $\{i_1, i_2\}$ and the abstraction hides i_2 (that it, $I = \{i_1\}$), we may have $\delta(s, \{i_1\}) = \{s', s''\}$, which abstracts the deterministic transitions $\delta'(s, \{i_1\}) = \{s'\}$ and $\delta'(s, \{i_1, i_2\}) = \{s''\}$, over $\{i_1, i_2\}$.

Given an input sequence $\xi = i_0, i_1, \ldots \in (2^I)^\omega$, a computation of $\mathcal{C}$ on ξ is a word $w = w_0, w_1, \ldots \in (2^{I \cup O})^\omega$

[2] Although we refer to our formalism as a *circuit*, it is not hardware-specific. The formalism is a *finite-state transducer*, and the theory we develop here applies also to analysis of software and other systems where finite-state transducers find application.

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 $s_{j+1} \in \rho(s_j, i_j)$ and $w_j = i_j \cup \rho(s_j)$. The language of $\mathcal{C}$, denoted $L(\mathcal{C})$ is union of all its computations.

The notion of refinement between circuits can be formalized in both the linear and the branching framework. Consider two circuits $\mathcal{I} = \langle I, O, C, \theta, \delta, \rho \rangle$ and $\mathcal{S} = \langle I', O', C', \theta', \delta', \rho' \rangle$. We refer to $\mathcal{S}$ and $\mathcal{I}$ as a specification and its implementation. Accordingly, we assume that $I' \subseteq I$, $O' \subseteq O$, and $C' \subseteq C$. In settings such as hierarchical refinement, the implementation may still not be precise, so we allow nondeterminism in both $\mathcal{I}$ and $\mathcal{S}$. 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 refinement 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 $H(t, t')$. Consider a simulation H from $\mathcal{I}$ to $\mathcal{S}$. 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 $\mathcal{I}$ [5, 14].

It is easy to see that the union of two simulations is a simulation. Hence, the maximal simulation from $\mathcal{I}$ to $\mathcal{S}$, denoted $\mathcal{H}$, is the union of all simulations from $\mathcal{I}$ to $\mathcal{S}$. Note that $\mathcal{S}$ simulates $\mathcal{I}$ iff $\mathcal{H}$ is initial with respect to $\mathcal{I}$ and $\mathcal{S}$.

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 [23].

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).

Temporal-logic formulas can be translated to circuits. An LTL formula ψ is translated to a circuit $\mathcal{C}_\psi$ such that $L(\mathcal{C}_\psi)$ contains exactly all computations that satisfy ψ [27]. A CTL* formula ψ is translated to a circuit $\mathcal{C}_\psi$ that is a *maximal model* for ψ, in the sense it simulates all circuits that satisfy ψ [18]. In Section 3 below, we define mutations on circuits and temporal-logic formulas. For a mutation μ on circuits, we say that a mutation μ_{TL} on temporal-logic formulas *corresponds* to μ if the circuit obtained by applying μ to $\mathcal{C}_\psi$ is equivalent to the circuit of the temporal-logic formula obtained by applying μ_{TL} to ψ. Note that correspondence can be defined in both the linear framework, in which case equivalence corresponds to trace equivalence, or in the branching framework, where equivalence corresponds to two-way simulation.

3 Mutations of Circuits, and their Analysis

In this section, we describe sample mutations to apply to a circuit, and analyse the satisfaction of mutant specifications by mutant implementations. We partition the mutations to three classes: mutations that remove behaviors (Section 3.1), modify behaviors (Section 3.2), and add behaviors (Section 3.3). For most of the mutations, we also describe corresponding mutations for a setting in which the specification is a temporal-logic formula.[3] We then describe a method for controlling the pattern (over time) that faults inducing the mutations have occurred, and for applying mutations on top of each other (Section 3.4).

Existing work suggests several definitions for vacuous satisfaction, based on mutations to temporal-logic formulas, and suggests several coverage metrics, based on mutations to implementations. The mutations described here suggest new helpful definition and metrics. Moreover, the duality between mutations, which we present in Section 4, relates mutations for coverage with mutations for vacuity; it enables, for example, vacuity checks that are geared towards the detection of physical failures.

3.1 Removing behaviors

In this section we describe mutations that remove behaviors of the circuit. 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* $\mathcal{S}$. Note that removal of behaviors from $\mathcal{S}$ may result in a mutant circuit $\mathcal{S}'$ that is

[3] Note that specifications given by temporal-logic formulas refer only to input and output signals. Thus, some of the mutations we describe, and which refer to pure control signals, do not have corresponding mutations in the temporal-logic setting.

not receptive. In this case, no implementation satisfies $\mathcal{S}'$. Note that in order for $\mathcal{S}'$ to be receptive, the original circuit $\mathcal{S}$ has to be nondeterministic, and removal of behaviors only decreases the amount of nondeterminism.

3.1.1 Removing a given set of behaviors

Consider a circuit $\mathcal{C}$. A *restriction* for $\mathcal{C}$ is a pair of functions $r_\theta : 2^I \to 2^{2^C}$ and $r_\delta : 2^C \times 2^I \to 2^{2^C}$. The circuit obtained from $\mathcal{C}$ by applying the restriction $\langle r_\theta, r_\delta \rangle$ 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 $\theta'(i) = \theta(i) \setminus r_\theta(i)$ and $\delta'(s, i) = \delta(s, i) \setminus r_\delta(s, i)$. Thus, $\mathcal{C}'$ is obtained from $\mathcal{C}$ by *decreasing its nondeterminism* according to r_θ and r_δ. We say that a restriction $\langle r_\theta, r_\delta \rangle$ *retains receptiveness* of $\mathcal{C}$ if the circuit obtained from $\mathcal{C}$ by applying the restriction $\langle r_\theta, r_\delta \rangle$ is receptive.

3.1.2 Removing behaviors that depend on a signal

A mutation $\mathcal{C}'$ in this class is parameterized by a signal $x \in I \cup C \setminus O$ and is obtained from $\mathcal{C}$ by removing transitions that depend on x. Formally, $\mathcal{C}' = \langle I, O, C, \theta', \delta', \rho \rangle$, where θ' and δ' depend on the type of x and are defined as follows.

- If $x \in I$, then for all $t \in 2^C$ and $i \in 2^I$, we have, $t \in \theta'(i)$ iff $t \in \theta(i \setminus \{x\}) \cap \theta(i \cup \{x\})$, and for all $s, t \in 2^C$ and $i \in 2^I$, we have $t \in \delta'(s, i)$ iff $t \in \delta(s, i \setminus \{x\}) \cap \delta(s, i \cup \{x\})$.
- If $x \in C \setminus O$, we restrict $\mathcal{C}'$ to states $s \setminus \{x\}$ and $s \cup \{x\}$ that agree on their output and transitions. Formally, for all $t \in 2^C$ and $i \in 2^I$, we have $\{t \setminus \{x\}, t \cup \{x\}\} \subseteq \theta'(i)$ iff $\{t \setminus \{x\}, t \cup \{x\}\} \subseteq \theta(i)$ and $\rho(t \setminus \{x\}) = \rho(t \cup \{x\})$. Also, for all $s, t \in 2^C$ and $i \in 2^I$, we have $\{t \setminus \{x\}, t \cup \{x\}\} \subseteq \delta'(s, i)$ iff $\{t \setminus \{x\}, t \cup \{x\} \subseteq \delta(s \setminus \{x\}, i) \cap \delta(s \cup \{x\}, i)$ and $\rho(t \setminus \{x\}) = \rho(t \cup \{x\})$.

Note that we could have excluded x from the set of signals of $\mathcal{C}'$. For the sake of uniformity among the different mutations, we leave the set of signals to agree with that of $\mathcal{C}$. This convention of leaving the signal set unchanged is going to be the case for other mutations also, and it is convenient when mutations are applied one on top of the other.

Remark 1 Note that the case x is purely an output signal is not interesting, as the behavior of a circuit always depends on the pure output signals. If we want to define a mutant with respect to a signal $x \in C \cap O$, the mutant would be receptive only if we define it with respect to a subset O' of the output signals such that $x \notin O'$. □

Remark 2 For a Kripke structure K with state space $2^C \times 2^I$, one can define an abstraction based on predicates in $C \cup I$. The mutations described above correspond to the case we restrict attention to predicates over $C \cup I \setminus \{x\}$, and define the abstraction so that there is a transition from an abstract state a to an abstract state a' iff all concrete states that correspond to a have transitions to all the concrete states that correspond to a'. □

When the specification is given by a temporal-logic formula, we remove behaviors that depend on a value of an observable signal $x \in I$ by replacing a formula ψ by the formula $\forall x \psi$. This mutation, which coincides with the semantic approach to vacuity of [2], captures better the intuition of removing behaviors that depend on a signal, and can also be applied to output signals. It does not correspond to the mutation defined for circuits, and a corresponding mutation on circuits for it involves "alternating" circuits (transitions may be conjunctively related to other transitions – these that correspond to the dual value of the signal).

3.1.3 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 (restricting x to 1 is similar) by disabling transitions in which the value of x is changed to 1. Formally, $\mathcal{C}' = \langle I, O, C, \theta', \delta', \rho \rangle$, where (the case $x \in I$ is not interesting, as then $\mathcal{C}'$ is clearly not receptive)

- If $x \in O$, then only states t for which $x \notin \rho(t)$ are reachable. Thus, for all $s \in 2^C$ and $i \in 2^I$, we have $\theta'(i) = \theta(i) \cap \{t : x \notin \rho(t)\}$, and $\delta'(s, i) = \delta(s, i) \cap \{t : x \notin \rho(t)\}$.

- If $x \in C$, then only states t for which $x \notin t$ are reachable. Thus, for all $s \in 2^C$ and $i \in 2^I$, we have $\theta'(i) = \theta(i) \cap \{t : x \notin t\}$, and $\delta'(s,i) = \delta(s,i) \cap \{t : x \notin t\}$. Note that we could have defined the set of control signals of $\mathcal{C}'$ to be $C \setminus \{x\}$, as x plays no role in $\mathcal{C}'$ and states $t \in 2^C$ such that $x \in t$ are not reachable.

Note also that when $x \in C \cap O$, the definitions coincide.

Restricting the value of x to 1 is similar: when x is an output signal, only states t with $x \in \rho(t)$ stay reachable, and when x is a control signal, only states t with $x \in t$ stay reachable.

For the setting of a temporal-logic formula, restricting the value of $x \in O$ to 0 amounts to replacing all the positive occurences (that is, occurrences in a scope of an even number of negations) of x by **false**. Likewise, restricting the value of x to 1 amounts to replacing all the negative occurrences of x by **false**. To see that these mutations correspond to the mutations on the circuit, recall that states in $\mathcal{C}_\psi$ are associated with subformulas of ψ, and the language of a state associated with a set S of formulas is exactly all computations that satisfy all the formulas in S. Accordingly, the temporal-logic mutation that replaces positive occurrences of x by **false** causes the language of states that contain a positive occurrence of x to be empty, which amounts to removing the transitions to them.

3.1.4 Analyzing vacuous satisfaction

We now analyze the different types of vacuous satisfaction that the different mutations induce. Our definition of vacuous satisfaction is a *state-based* generalization of temporal-logic vacuity. With temporal logic, one looks for sub-specifications that do not affect the satisfaction of the specification in the system. Technically, we say that a subformula φ of ψ does not affect the satisfaction of ψ in $\mathcal{C}$ if all formulas obtained from ψ by replacing φ by some other formula are still satisfied in $\mathcal{C}$ [3]. In fact, when we talk about a particular occurrence of φ in ψ, it is enough to check the most challenging replacement for φ [19]. For example, in $\psi = G(req \rightarrow Fgrant)$, the most challenging replacement of $grant$ is **false**. Indeed, $grant$ does not affect the satisfaction ψ in $\mathcal{C}$ if $\mathcal{C}$ also satisfies the formula $G\neg req$, obtained from ψ by replacing $grant$ by **false**.

We say that a mutation μ does not affect the satisfaction of $\mathcal{S}$ by $\mathcal{I}$ if $\mathcal{S}_\mu$ is receptive and $\mathcal{I} \models \mathcal{S}_\mu$. We analyze each of the mutations that remove behaviors in Appendix B, and here we only highlight some mutations and describe the general common intuition in these mutation: if μ removes behaviors, and does not affect the satisfaction of $\mathcal{S}$ by $\mathcal{I}$, then we can conclude that each of the behaviors of $\mathcal{S}$ that have been removed is either not exhibited in the implementation, or is subsumed by another behavior of the specification. Of particular interest are maximal restrictions, restrictions that contain a path, restrictions that remove self-loops, the mutations that are parameterized by specific signals, and restrictions that correspond vacuity in temporal logic.

Let us elaborate on the latter restrictions, as it shows that our state-based framework subsumes the traditional temporal-logic-based framework. Replacements of sub-formulas by the most challenging replacement corresponds to removal of transitions in a specification circuit $\mathcal{C}_\psi$ obtained by a translation of an LTL formula ψ [27]. Intuitively, since the states of $\mathcal{C}_\psi$ are associated with sets of subformulas of ψ, it is not hard to map mutations in ψ to mutations in $\mathcal{C}_\psi$.[4]

For simplicity, we assume that the formula is in positive negation form, in which case the polarity of all sub-formulas is positive, thus the most challenging replacement is false. In typical translations, each state of $\mathcal{S}$ is associated with a set Q of sub-formulas of ψ. The set Q is consistent, in the sense that a disjunction $\varphi_1 \vee \varphi_2$ is in Q iff at least one of φ_1 and φ_2 is in Q, and a conjunction $\varphi_1 \wedge \varphi_2$ is in Q iff both φ_1 and φ_2 are in Q. The temporal operators in ψ induce similar consistency requirements that are reflected in the definition of the transitions. Accordingly, given a circuit $\mathcal{S}$ for ψ, it is possible to obtain from $\mathcal{S}$ a circuit $\mathcal{S}'$ for the formula in which φ is replaced by false by removing states and transitions in which consistency of disjunctions is taken care of by φ and consistency of conjunctions depends on φ. For example, given a circuit for $\psi = G(\neg req \vee Fgrant)$, setting $grant$ to false amounts to disabling transitions to states Q such that $\neg req \vee Fgrant$ is in Q but $\neg req$ is not in Q. Note that the new circuit does not limit the value of *grant*, but its value no longer affects the satisfaction of $\neg req \vee Fgrant$.

[4] The user may prefer to construct a circuit for the revised formula directly. The point we make here is that our state-based framework subsumes the temporal-logic-based framework, as replacements of sub-formulas by false corresponds to removal of transitions in the specification circuit.

3.2 Modifying Behaviors

In this section we describe mutations that modify the behavior of a circuit. 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* $\mathcal{S}$.

3.2.1 Forcing a signal to be flipped or get stuck

We start with forcing a signal to be flipped. A mutation in this class is parameterized by a signal x, and the mutation flips the value of x; i.e., it takes the opposite value of what it is supposed to take. For a set X, an element $x \in X$, and a set $Y \subseteq X$, we use $twin_x(Y)$ to denote the set obtained from Y by dualizing the value of x. Thus, $x \in twin_x(Y)$ iff $x \notin Y$. Given a circuit $\mathcal{C}$ and a signal x, the circuit in which the value of x is always flipped is $\mathcal{C}' = \langle I, O, C, \theta', \delta', \rho' \rangle$, where

- If $x \in I$, then $\rho' = \rho$, and for all $s \in 2^C$ and $i \in 2^I$, we have $\theta'(i) = \theta(twin_x(i))$ and $\delta'(s, i) = \delta(s, twin_x(i))$.
- If $x \in O \setminus C$, then $\theta' = \theta$, $\delta' = \delta$, and for all $t \in 2^C$, we have $\rho'(t) = twin_x(\rho(t))$.
- If $x \in C$, then $\rho' = \rho$ and for all $s \in 2^C$ and $i \in 2^I$, we have $\theta'(i) = \{twin_x(t) : t \in \theta(i)\}$, and $\delta'(s, i) = \{twin_x(t) : t \in \delta(s, i)\}$. Thus, whenever $\mathcal{C}$ has a transition to a state t, the mutation $\mathcal{C}'$ goes instead to $twin_x(t)$. Note that while $\rho' = \rho$, the change in the transition causes a change in the observable output.

Note that no matter what the type of x is, the circuit $\mathcal{C}'$ is receptive.

For the setting of a temporal-logic specification, flipping the value of $x \in I \cup O$ amounts to negating all the occurences of x. To see that the temporal-logic mutation corresponds to the one on the circuit, recall that when $\mathcal{C}_\psi$ is in a state associated with a set S of subformulas of ψ, its output is $S \cup O$, and it gets to the state by reading the input assignment $I \cap S$.

A mutation that forces a signal to get stuck is parameterized by a signal $x \in I \cup O \cup C$ and it forces x to get stuck at 0 (forcing x to get stuck at 1 is similar) by acting as if $x = 0$ regardless of its actual value. Thus, the mutation is similar to the one that flips the value of x, only that here the value is flipped only when $x = 1$. As there, the mutant circuit $\mathcal{C}'$ stays receptive. Note that unlike the mutation described in Section 3.1.3, here we do not disable transitions after which the value of x is 1, but rather we flip the value of x in the destination state. The formal definitions of the mutation are given in Appendix A.1.

3.2.2 Forcing a delayed or a prematured output

We start with introducing a delay. A mutation in this class causes a delay (of a fixed number of cycles, specified by the user) in the output of the circuit. For that, the mutant system has additional control signals that remember the output assignment that should have been output in the previous cycles. Given a circuit $\mathcal{C}$ and a number $k \geq 1$ describing the delay, the mutation C' of $\mathcal{C}$ in which a k-cycle delay is introduced has control signals $C' = C \cup (O \times \{1, \ldots, k\})$. Intuitively, each state corresponds to a tuple $\langle s, \sigma_1, \ldots, \sigma_k \rangle \in 2^C \times 2^O \times \cdots \times 2^O$, where σ_1 maintains the output of the state visited in the previous cycle, σ_2 maintains the output of the state visited before that, and so on, until σ_k, which maintains the output of the state visited before k cycles. The transition function is such that for all $\langle s, \sigma_1, \ldots, \sigma_k \rangle \in 2^C \times 2^O \times \cdots \times 2^O$, and $i \in 2^I$, we have that $\delta'(\langle s, \sigma_1, \ldots, \sigma_k \rangle, i) = \delta(s, i) \times \{\langle \rho(s), \sigma_1, \ldots, \sigma_{k-1} \rangle\}$. The full description of the construction can be found in Appendix A.2. It is not hard to see that the output of $\mathcal{C}'$ is indeed a k-cycle delay of the output of $\mathcal{C}$. For example, when $k = 1$, we have that $s_0, s_1, s_2, \ldots \in 2^C$ is a path in $\mathcal{C}$ with output $o_0, o_1, o_2, \ldots$ iff $\langle s_0, \emptyset \rangle, \langle s_1, o_0 \rangle, \langle s_2, o_1 \rangle, \ldots \in 2^C \times 2^O$ is a path in $\mathcal{C}'$ with output $\emptyset, o_0, o_1, \ldots$.

We proceed to introducing prematureness. A mutation in this class causes a premature (by a fixed number of cycles, specified by the user) output of the circuit. For that, the mutant system has additional control signals that maintain a guess for the output assignment expected in the future. The mutant circuit outputs the guessed output, but it gets stuck in computations in which the guess turns out not to be valid. The full details of the construction can be found in Appendix A.2.

For the setting of a temporal-logic formula, forcing a delay of k cycles in the output corresponds to replacing all occurrences of all signals $x \in O$ by $X^k x$, where X is the temporal next-time operator. Likewise, forcing an output to be premature by k cycles corresponds to replacing all occurrences of $x \in O$ by $Y^k x$, for the temporal previous-time operator Y.

3.2.3 Inserting perturbation

This class of mutation contains several sub-classes, all based on the same principle, namely inserting small local perturbation to the circuit. The mutations correspond to common faults. We mention here some examples. A *permutation mutation* is parameterized by a permutation $\pi = \langle i_1, i_2, \ldots, i_k \rangle$ in Π_k (the set of permutations of length k) and whenever it is activated, it permutes the next k output assignments according to π. Technically, this involves a combination of the delay and look ahead techniques used above. A *stuttering mutation* is parameterized by an integer k, and whenever it is activated, it ignores the input and stays in the same state for k cycles (the unbounded version corresponds to adding self loops, and is mentioned in Section 3.3.1). Finally, a *noise mutation* is also parameterized by an integer k and, whenever it is activated, it causes the cycle to output arbitrary output for the next k cycles.

Various perturbations can also be applied to temporal-logic formulas. One can add the X and Y temporal operators arbitrarily, replace X by F in order to inserting unbounded delay, use X or U in order force a bounded or an unbounded stuttering, replace assertions about the present by eventualities in order to insert noise, and so on.

3.2.4 Analyzing diverse satisfaction

Our definition of diverse satisfaction is related to the way coverage is measured in formal verification. There, one looks for components of the circuit that do not play a role in the satisfaction of the specification. The standard way to check coverage is to look for components that can be modified and still satisfy the specification [16, 10]. Mutations that change the output of states are studied in [8]. Here, we have generalize the idea to flips in the input and the control signals, in both the specification and the implementation, and have suggested new mutations. The new mutations are related to physical faults that cause flips of signals, cause a signal to get stuck at a value, or cause a delay in the output.

3.3 Adding behaviors

In this section we describe are mutations that add behaviors to the circuit. 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* $\mathcal{S}$.

3.3.1 Adding a fixed set of behaviors

We first define arbitrary addition of behaviors, dual to the removal of behaviors in Section 3.1.1. Consider a circuit $\mathcal{C}$. An *extension* for $\mathcal{C}$ is a pair of functions $r_\theta : 2^I \to 2^{2^C}$ and $r_\delta : 2^C \times 2^I \to 2^{2^C}$. The circuit obtained from $\mathcal{C}$ by applying the extension $\langle r_\theta, r_\delta \rangle$ 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 $\theta'(i) = \theta(i) \cup r_\theta(i)$ and $\delta'(s, i) = \delta(s, i) \cup r_\delta(s, i)$. Thus, $\mathcal{C}'$ is obtained from $\mathcal{C}$ by *increasing its nondeterminism* according to r_θ and r_δ.

3.3.2 Freeing a signal

A mutation in this class is parameterized by a signal x and it adds to $\mathcal{C}$ behaviors that agree with existing behaviors of $\mathcal{C}$ on everything but x. Formally, $\mathcal{C}' = \langle I, O, C, \theta', \delta', \rho \rangle$, where

- If $x \in I$, then for all $t \in 2^C$ and $i \in 2^I$, we have $t \in \theta'(i)$ iff $t \in \theta(i \setminus \{x\}) \cup \theta(i \cup \{x\})$. Also, for all $s, t \in 2^C$ and $i \in 2^I$, we have $t \in \delta'(s, i)$ iff $t \in \delta(s, i \setminus \{x\}) \cup \delta(s, i \cup \{x\})$.
- If $x \in C$, then for all $t \in 2^C$ and $i \in 2^I$, we have $t \in \theta'(i)$ iff $\{t \setminus \{x\}, t \cup \{x\}\} \cap \theta(i) \neq \emptyset$. Also, for all $s, t \in 2^C$ and $i \in 2^I$, we have $t \in \delta'(s, i)$ iff $\{t \setminus \{x\}, t \cup \{x\}\} \cap (\delta(s \setminus \{x\}, i) \cup \delta_{\mathcal{I}}(s \cup \{x\}, i)) \neq \emptyset$.
- If $x \in O \setminus C$, we need not construct a mutant and just ignore x when we check for containment or simulation.

When the specification is given by a temporal-logic formula, the corresponding mutation frees a signal $x \in I \cup O$ by replacing a formula ψ by the formula $\exists x \psi$. Indeed, this amounts to increasing the nondeterminism in $\mathcal{C}_\psi$ by ignoring the value of x in transitions.

3.3.3 Analyzing loose satisfaction

Our definition of loose satisfaction suggests a new approach to coverage, which is dual to the approach taken in vacuity. Rather than modify components of the implementation, we add to it new behaviors, When we add behaviors that are dual to these exhibited by a component, we can say that the component is covered if the circuit with the dual behavior no longer satisfies the specification. We analyze each of the mutations that add behaviors in Appendix B, and here we only highlight some mutations and describe the general common intuition in these mutation: if μ adds behaviors, and does not affect the satisfaction of $\mathcal{S}$ in $\mathcal{I}$, then we can conclude that each of the behaviors of $\mathcal{I}_\mu$ that have been added is allowed by the specification, or is subsumed by another behavior of the implementation. Of particular interest are maximal extensions, extensions that contain a path, restrictions that add self-loops, and mutations that are parameterized by specific signals.

3.4 Controlled injection of mutations

The mutations described in Sections 3.1, 3.2, and 3.3 correspond to a persistent fault, in the sense that the environment interacts with $\mathcal{C}'$, rather than with $\mathcal{C}$, from the initialization of the communication ad infinitum. In addition, only a single type of fault is injected. In this section we generalize our definitions of mutations and make it possible to control the pattern (over time) in which mutations happen. In addition, it is possible to apply mutations one on top of the other. For example, the user can generate a mutant circuit in which x is restricted to 0 at least once in all 5-cycle windows and y is restricted to 1 at least once in all 4-cycle windows. The mutation can be applied universally (in all computations) or existentially (in at least one computation).

We give the details of the construction in Appendix A.3. Essentially, the original circuit is combined with a circuit $\mathcal{C}_{mut}$ that specifies the pattern in which the mutations are injected. Each letter in the alphabet of $\mathcal{C}_{mut}$ is an ordered subset of the set of possible mutations (for some combinations, the order of application of mutations is important) and the combination with $\mathcal{C}$ applies the faults locally.

We note that it is often desirable to specify fault-patterns that are not safety properties. Then, one has to introduce fairness to $\mathcal{C}_{mut}$, which then induces fairness also in the controlled-fault version of $\mathcal{C}$. Accordingly, the containment and simulation relations we used for satisfaction should be replaced by fair containment and fair simulation.

Also note that taking the product of $\mathcal{C}$ with $\mathcal{C}_{mut}$ changes not only the language of $\mathcal{C}$ according to the mutations but, unless $\mathcal{C}_{mut}$ is deterministic, also changes its branching structure. Accordingly, one should either work with a deterministic $\mathcal{C}_{mut}$, or follow the linear approach.

4 Relating Vacuous, Diverse, and Loose Satisfaction

In this section, we relate vacuous, diverse, and loose satisfaction. We show that many of the mutations suggested in Section 3 can be paired with a *dual mutation*, and that satisfaction of a mutated specification is related to satisfaction by a dually mutated implementation. We begin with the definition of dual mutation.

Definition 1 *Let M_a, M_m, and M_r be the sets of mutations that respectively add, modify, and remove behaviors. For a circuit $\mathcal{C}$ and a mutation μ, let $\mathcal{C}_\mu$ denote the mutant circuit obtained from $\mathcal{C}$ by applying μ.*

Then, mutations $\mu \in M_a \cup M_m$ and $\tilde{\mu} \in M_r \cup M_m$ are fully dual if, for all implementations $\mathcal{I}$ and specifications $\mathcal{S}$, we have that $\mathcal{I}_\mu \models \mathcal{S}$ iff $\mathcal{I} \models \mathcal{S}_{\tilde{\mu}}$.

Definition 1 refers to both the linear and branching approaches, and requires the implication between $\mathcal{I}_\mu \models \mathcal{S}$ and $\mathcal{I} \models \mathcal{S}_{\tilde{\mu}}$ to be two-sided. Mutations may also be *partially dual*, with only one-sided implication or with the duality being valid in only the linear or the branching approach. A *dual mutation* is either a fully-dual or a partially-dual mutation. The notion of duality in general can be understood as follows. Consider a set Δ of behaviors. We can add to $\mathcal{I}$ behaviors in Δ that agree with S, and get a mutant implementation that still satisfies $\mathcal{S}$. Dually, we can remove from

$\mathcal{S}$ behaviors in Δ that are not present in $\mathcal{I}$, and get a mutant specification that is still satisfied by $\mathcal{I}$. The interesting phenomenon is that the above two dual sets of behaviors can be obtained by means of mutations described in Section 3. The rest of this section presents theorems on duality for some interesting mutations.

We start with persistent faults and then study its generalization to arbitrary injection patterns. We first consider diverse satisfaction and show that a mutation that flips the value of a signal is self-dual.

Theorem 1 *Consider a specification $\mathcal{S}$ and an implementation $\mathcal{I}$ for it. Let $\mathcal{I}'$ and $\mathcal{S}'$ be the mutations of $\mathcal{I}$ and $\mathcal{S}$, respectively, obtained by flipping the value of a signal x. If $x \in I \cup O \setminus C$ then $\mathcal{I} \subseteq \mathcal{S}'$ iff $\mathcal{I}' \subseteq \mathcal{S}$ and $\mathcal{I} \leq \mathcal{S}'$ iff $\mathcal{I}' \leq \mathcal{S}$. If $x \in C$ and $O \subseteq C$, then $\mathcal{I} \leq \mathcal{S}'$ iff $\mathcal{I}' \leq \mathcal{S}$.*

Proof: Let $\mathcal{I} = \langle I, O, C, \theta_{\mathcal{I}}, \delta_{\mathcal{I}}, \rho_{\mathcal{I}} \rangle, \mathcal{I}' = \langle I, O, C, \theta'_{\mathcal{I}}, \delta'_{\mathcal{I}}, \rho'_{\mathcal{I}} \rangle, \mathcal{S} = \langle I, O, C, \theta_{\mathcal{S}}, \delta_{\mathcal{S}}, \rho_{\mathcal{S}} \rangle$, and $\mathcal{S}' = \langle I, O, C, \theta'_{\mathcal{S}}, \delta'_{\mathcal{S}}, \rho'_{\mathcal{S}} \rangle$.

We start with the case $x \in I$. Assume that $\mathcal{I} \subseteq \mathcal{S}'$, and let $w' = w'_0, w'_1, \ldots \in (2^{I \cup O})^\omega$ be a computation of $\mathcal{I}'$. Let $w'_j = i'_j \cup o'_j$ with $i'_j \in 2^I$ and $o'_j \in 2^O$. Then, $w = w_0, w_1, \ldots \in 2^{I \cup O}$, with $w_j = twin_x(i'_j) \cup o'_j$ is a computation of $\mathcal{I}$, and, by the assumption, also a computation of $\mathcal{S}'$. Now, by the definition of $\mathcal{S}'$, it follows that $w'' = w''_0, w''_1, \ldots \in 2^{I \cup O}$, with $w''_j = twin_x(twin_x(i'_j)) \cup o'_j$ is a computation of $\mathcal{S}$. Since for all $i \in 2^I$ we have that $twin_x(twin_x(i)) = i$, we have that $w'' = w'$, thus w' is also a computation of $\mathcal{S}$ and we are done. The other direction, namely showing that $\mathcal{I}' \subseteq \mathcal{S}$ implies $\mathcal{I} \subseteq \mathcal{S}'$ is identical.

Assume now that $\mathcal{I} \leq \mathcal{S}'$. Let $\mathcal{H}$ be an initial simulation relation from $\mathcal{I}$ to $\mathcal{S}'$. We claim that $\mathcal{H}$ is an initial simulation relation from $\mathcal{I}'$ to $\mathcal{S}$. Since $\rho_I = \rho'_I$ and $\rho_{\mathcal{S}} = \rho'_S$, proving that $\mathcal{H}$ is a simulation from $\mathcal{I}'$ to $\mathcal{S}$, we only have to prove that for all s and s' such that $\mathcal{H}(s, s')$, and for all $i \in 2^I$ and $t \in \delta'_I(s, i)$ there is $t' \in \delta_{\mathcal{S}}(s', i)$ such that $\mathcal{H}(t, t')$. We first prove that $\mathcal{H}$ is initial with respect to $\mathcal{I}'$ and $\mathcal{S}$; that is, for all $i \in 2^I$ and $t \in \theta'_{\mathcal{I}}(i)$ there is $t' \in \theta_{\mathcal{S}}(i)$ such that $\mathcal{H}(t, t')$.

Consider an input assignment $i \in 2^I$ and a state $t \in \theta'_{\mathcal{I}}(i)$. By the definition of $\theta'_{\mathcal{I}}$, we know that $t \in \theta_{\mathcal{I}}(twin_x(i))$. Since $\mathcal{H}$ initial with respect to $\mathcal{I}$ and $\mathcal{S}'$, there is a state $t' \in \theta'_{\mathcal{S}}(twin_x(i))$ such that $\mathcal{H}(t, t')$. By the definition of $\theta'_{\mathcal{S}}$, we know that $t' \in \theta_{\mathcal{S}}(i)$, and we are done. We now proceed to prove that $\mathcal{H}$ is a simulation. Let s and s' such that $\mathcal{H}(s, s')$. Consider an input assignment $i \in 2^I$ and a state $t \in \delta'_I(s, i)$. By the definition of $\delta'_{\mathcal{I}}$, we know that $t \in \delta_{\mathcal{I}}(s, twin_x(i))$. Since $\mathcal{H}$ is a simulation from $\mathcal{I}$ to $\mathcal{S}'$, there is a state $t' \in \delta'_{\mathcal{S}}(s', twin_x(i))$ such that $\mathcal{H}(t, t')$. By the definition of $\delta'_{\mathcal{S}}$, we know that $t' \in \delta_{\mathcal{S}}(s, i)$, and we are done.

The other direction, namely showing that $\mathcal{I}' \leq \mathcal{S}$ implies $\mathcal{I} \leq \mathcal{S}'$ is identical: an initial simulation relation from $\mathcal{I}'$ to $\mathcal{S}$ is also an initial simulation relation from $\mathcal{I}$ to $\mathcal{S}'$.

We proceed to the case $x \in O \setminus C$. Assume that $I \subseteq S'$, and let $w' = w'_0, w'_1, \ldots \in 2^{I \cup O}$ be a computation of $\mathcal{I}'$. Let $w'_j = i'_j \cup o'_j$ with $i'_j \in 2^I$ and $o'_j \in 2^O$. Then, $w = w_0, w_1, \ldots \in 2^{I \cup O}$, with $w_j = i'_j \cup twin_x(o'_j)$ is a computation of $\mathcal{I}$, and, by the assumption, also a computation of $\mathcal{S}'$. Now, by the definition of $\mathcal{S}'$, it follows that $w'' = w''_0, w''_1, \ldots \in 2^{I \cup O}$, with $w''_j = i'_j \cup twin_x(twin_x(o'_j))$ is a computation of $\mathcal{S}$. Since for all $o \in 2^O$ we have that $twin_x(twin_x(o)) = o$, we have that $w'' = w'$, thus w' is also a computation of $\mathcal{S}$ and we are done. The other direction is identical.

Assume now that $\mathcal{I} \leq \mathcal{S}'$. Let $\mathcal{H}$ be an initial simulation relation from $\mathcal{I}$ to $\mathcal{S}'$. We claim that $\mathcal{H}$ is an initial simulation relation from $\mathcal{I}'$ to $\mathcal{S}$. Since $\theta_I = \theta'_I$, $\theta_{\mathcal{S}} = \theta'_{\mathcal{S}}$, $\delta_{\mathcal{I}} = \delta'_{\mathcal{I}}$, and $\delta_{\mathcal{S}} = \delta'_{\mathcal{S}}$, we only have to prove that for all s and s' such that $\mathcal{H}(s, s')$, we have $\rho'_{\mathcal{I}}(s) = \rho_{\mathcal{S}}(s')$. Since $\mathcal{H}$ is a simulation from $\mathcal{I}$ to $\mathcal{S}'$, we know that $\rho_{\mathcal{I}}(s) = \rho'_{\mathcal{S}}(s')$. Recall that $\rho'_{\mathcal{I}}(s) = twin_x(\rho_{\mathcal{I}}(s))$ and $\rho'_{\mathcal{S}}(s') = twin_x(\rho_{\mathcal{I}}(s'))$. Hence, $\rho_{\mathcal{I}}(s) = \rho'_{\mathcal{S}}(s')$ implies that $\rho_{\mathcal{I}}(s) = twin_x(\rho_{\mathcal{S}}(s'))$, which implies, by the definition of $twin_x$, that $twin_x(\rho_{\mathcal{I}}(s)) = \rho_{\mathcal{S}}(s')$, which implies our goal, namely $\rho'_{\mathcal{I}}(s) = \rho_{\mathcal{S}}(s')$. The other direction is identical.

We now move to the case $x \in C$ and $O \subseteq C$. In Example 3 below, we demonstrate that the requirement $O \subseteq C$ is essential. Assume that $\mathcal{I} \leq \mathcal{S}'$. Let $\mathcal{H}$ be an initial simulation relation from $\mathcal{I}$ to $\mathcal{S}'$. Let $\mathcal{H}_x = \{\langle twin_x(t), twin_x(t') \rangle : \mathcal{H}(t, t')\}$. We claim that $\mathcal{H}_x$ is an initial simulation relation from $\mathcal{I}'$ to $\mathcal{S}$. We first prove that if s and s' are such that $\mathcal{H}_x(s, s')$, then $\rho'_{\mathcal{I}}(s) = \rho_{\mathcal{S}}(s')$. Since $\mathcal{H}_x(s, s')$, then $\mathcal{H}(twin_x(s), twin_x(s'))$. Therefore, $\rho_{\mathcal{I}}(twin_x(s)) = \rho'_{\mathcal{S}}(twin_x(s'))$. Since $O \subseteq C$, we have that $\rho_{\mathcal{I}}(twin_x(s)) = twin_x(\rho_{\mathcal{I}}(s))$ and $\rho'_{\mathcal{S}}(twin_x(s')) = twin_x(\rho'_{\mathcal{S}}(s'))$. Hence, $twin_x(\rho_{\mathcal{I}}(s)) = twin_x(\rho'_{\mathcal{S}}(s'))$, implying that $\rho_{\mathcal{I}}(s) = \rho'_{\mathcal{S}}(s')$. Now, since $\rho'_{\mathcal{I}} = \rho_{\mathcal{I}}$ and $\rho'_{\mathcal{S}} = \rho_{\mathcal{S}}$, the latter implies that $\rho'_{\mathcal{I}}(s) = \rho_{\mathcal{S}}(s')$, and we are done.

We now prove that $\mathcal{H}_x$ is initial with respect to $\mathcal{I}'$ and $\mathcal{S}$. Consider an input assignment $i \in 2^I$ and a state $t \in \theta'_{\mathcal{I}}(i)$. By the definition of $\theta'_{\mathcal{I}}$, we know that $twin_x(t) \in \theta_{\mathcal{I}}(i)$. Since $\mathcal{H}$ initial with respect to $\mathcal{I}$ and $\mathcal{S}'$,

there is a state $t' \in \theta'_{\mathcal{S}}(i)$ such that $\mathcal{H}(twin_x(t), t')$. The state $twin_x(t')$ then satisfies both $twin_x(t') \in \theta_{\mathcal{S}}(i)$ and $\mathcal{H}_x(t, twin_x(t'))$, and we are done.

It is left to prove that $\mathcal{H}_x$ is a simulation. Let s and s' be such that $\mathcal{H}_x(s, s')$. Consider an input assignment $i \in 2^I$ and a state $t \in \delta'_I(s, i)$. By the definition of $\delta'_{\mathcal{I}}$, we know that $twin_x(t) \in \delta_{\mathcal{I}}(s, i)$. Since $\mathcal{H}$ is a simulation from $\mathcal{I}$ to $\mathcal{S}'$, there is a state $t' \in \delta'_{\mathcal{S}}(s', i)$ such that $\mathcal{H}(twin_x(t), t')$. The state $twin_x(t')$ then satisfies both $twin_x(t') \in \delta_{\mathcal{S}}(s', i)$ and $\mathcal{H}_x(t, twin_x(t'))$, and we are done.

The other direction is identical.

□

Note that for the case $x \in C$, we require that $O \subseteq C$. In Example 1 we show that the requirement is essential.

Example 1 In Figure 1 below, we describe an implementation $\mathcal{I}$ with $I = \{z\}$, $C = \{x, y\}$, and $O = \{x, v\}$. We describe the value of z on the edges, the values of x and y in the two-bit vector at the top of each state, and the value of v is at the bottom of each state. For example, on input $z = 0$, the implementation $\mathcal{I}$ goes to a state with $x = 0$ and $y = 1$, in which $v = 1$. A specification $\mathcal{S}$ for $\mathcal{I}$ is described at the right. A bold edge stands for edges from all states (including a self loop). Thus, for example, no matter what the current state is, the specification can move, on input $z = 1$, either to a state with $x = y = 0$, in which $v = 0$, or to a state with $x = 1$ and $y = 0$, in which $v = 0$. It is easy to see that the nondeterministic specification has more behaviors than the implementation, thus $\mathcal{I} \models \mathcal{S}$ (in both the linear and branching approaches).

Consider now the mutation $\mathcal{S}'$ of $\mathcal{S}$ that flips the value of the (observable) control signal x. It is easy to see that the value of x does not affect the behavior of $\mathcal{S}$ and that $\mathcal{S}' = \mathcal{S}$. On the other hand, flipping the value of x in $\mathcal{I}$ results in new behaviors. For example, on input $z = 0$, the implementation $\mathcal{I}'$ goes to a state in which $v = 0$, which is not possible in $\mathcal{I}$, and is not allowed by $\mathcal{S}$. Thus, $\mathcal{I} \not\models \mathcal{S}$.

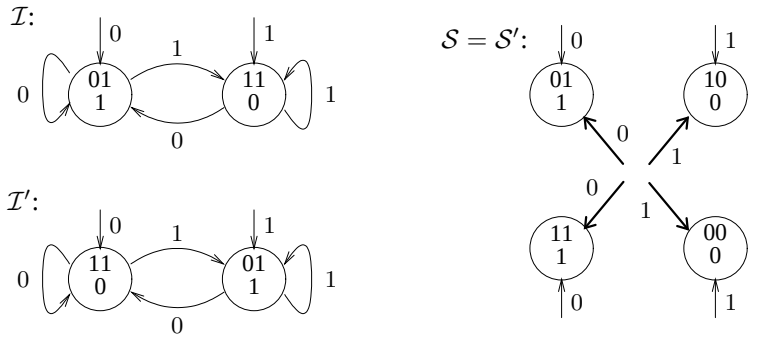

Figure 1: A counter example for the case $x \in C$, all states are reachable, but $O \not\subseteq C$.

Note also that if $C \subseteq O$, then $\mathcal{H}(s, s')$ implies that $s = s'$. Thus, in this case, if all the states that are reachable in $\mathcal{I}'$ are also reachable in $\mathcal{I}$, the flip mutation is self-dual. On the other hand, states that are not reachable in $\mathcal{I}$ and become reachable in $\mathcal{I}'$ are problematic, as we show in Example 2.

Example 2 In Figure 2 below, we describe an implementation $\mathcal{I}$ with $I = \{z\}$, $O = \{x, v\}$, and $C = \{x\}$. The state with $x = 0$ is not reachable in $\mathcal{I}$. It is easy to see that $\mathcal{I} \models \mathcal{S}$ (in both the linear and branching approaches). Flipping x in $\mathcal{S}$ does not change $\mathcal{S}$, thus $\mathcal{I} \models \mathcal{S}'$. On the other hand, flipping x in $\mathcal{I}$ makes the state with $x = 0$ and $v = 1$ reachable. Since the state with $x = 0$ in $\mathcal{S}$ has $v = 0$, we have that $\mathcal{I}' \not\models \mathcal{S}$.

Next, we study duality between a delayed implementation and a premature specification. As demonstrated in Example 3, changing the timing of the outputs may change the branching nature of the circuit. This is why Theorem 2 is valid for the linear approach but not for the branching approach.

Theorem 2 *Consider a specification $\mathcal{S}$, an implementation $\mathcal{I}$, and an integer $k \geq 1$. Let $\mathcal{I}'$ be the mutation of $\mathcal{I}$ in which a k-cycle delay is introduced, and let $\mathcal{S}'$ be the mutation of $\mathcal{S}$ in which a k-cycle prematureness is introduced. Then, $\mathcal{I}' \subseteq \mathcal{S}$ implies that $\mathcal{I} \subseteq \mathcal{S}'$.*

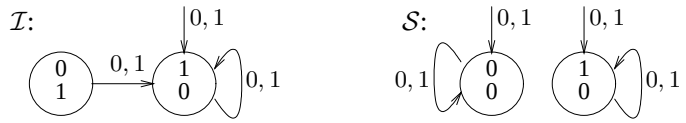

Figure 2: A counter example for the case $x \in C \subseteq O$, yet $O \not\subseteq C$.

Proof: Let $\mathcal{I} = \langle I, O, C, \theta_{\mathcal{I}}, \delta_{\mathcal{I}}, \rho_{\mathcal{I}} \rangle$, $\mathcal{I}' = \langle I, O, C, \theta'_{\mathcal{I}}, \delta'_{\mathcal{I}}, \rho'_{\mathcal{I}} \rangle$, $\mathcal{S} = \langle I, O, C, \theta_{\mathcal{S}}, \delta_{\mathcal{S}}, \rho_{\mathcal{S}} \rangle$, and $\mathcal{S}' = \langle I, O, C, \theta'_{\mathcal{S}}, \delta'_{\mathcal{S}}, \rho'_{\mathcal{S}} \rangle$.

Consider a computation $w = w_0, w_1, \ldots \in (2^{I \cup O})^\omega$ of $\mathcal{I}$. Recall that there is an input sequence $\xi = i_0, i_1, \ldots \in (2^I)^\omega$ and a path $s = s_0, s_1, \ldots \in (2^C)^\omega$ in $\mathcal{I}$ such that π can be traversed while reading ξ and w describes the input and output along this traversal. By the definition of $\mathcal{I}'$, the path $s' = \langle s_0, \emptyset, \ldots, \emptyset \rangle$, $\langle s_1, \rho_{\mathcal{I}}(s_0), \emptyset, \ldots, \emptyset \rangle$, $\langle s_2, \rho_{\mathcal{I}}(s_1), \rho_{\mathcal{I}}(s_0)\emptyset, \ldots, \emptyset \rangle, \ldots$ is a path in $\mathcal{I}'$. Since $\mathcal{I}' \subseteq \mathcal{S}$, there is a path $t = t_0, t_1, \ldots$ in $\mathcal{S}$ that can be traversed while reading ξ and for which $\rho'_{\mathcal{I}}(\langle s_j, \rho_{\mathcal{I}}(s_{j-1}), \ldots, \rho_{\mathcal{I}}(s_{j-k}) \rangle) = \rho_{\mathcal{S}}(t_j)$ for all $j \geq k$. Since $\rho'_{\mathcal{I}}(\langle s_j, \rho_{\mathcal{I}}(s_{j-1}), \ldots, \rho_{\mathcal{I}}(s_{j-k}) \rangle) = \rho_{\mathcal{I}}(s_{j-k})$, it follows that $\rho_{\mathcal{I}}(s_j) = \rho_{\mathcal{S}}(t_{j+k})$ for all $j \geq 0$. Then, however, the path $t' = \langle t_0, \rho_{\mathcal{S}}(t_1), \ldots, \rho_{\mathcal{S}}(t_k) \rangle$, $\langle t_1, \rho_{\mathcal{S}}(t_2), \ldots, \rho_{\mathcal{S}}(t_{k+1}) \rangle$, $\langle t_2, \rho_{\mathcal{S}}(t_3), \ldots, \rho_{\mathcal{S}}(t_{k+2}) \rangle, \ldots$ is a path of $\mathcal{S}'$ that can be traversed while reading ξ. Since $\rho'_{\mathcal{S}}(\langle t_j, \rho_{\mathcal{S}}(t_{j+1}), \ldots, \rho_{\mathcal{S}}(t_{j+k}) = \rho_{\mathcal{S}}(t_{j+k}) \rangle$ for all $j \geq 0$, we got that $\rho_{\mathcal{I}}(s_j)$ is equal to $\rho'_{\mathcal{S}}(\langle t_j, \rho_{\mathcal{S}}(t_{j+1}), \ldots, \rho_{\mathcal{S}}(t_{j+k}) \rangle$ for all $j \geq 0$, and we are done.

□

Note that only one direction of the duality is established in Theorem 2. The other direction is correct up to the labeling of the first k output assignments in each computation. Indeed, since $\mathcal{I}'$ does not restrict the first k output assignments, $\mathcal{I} \subseteq \mathcal{S}'$ only implies *post-stabilization* containment of $\mathcal{I}'$ in $\mathcal{S}$, namely containment ignoring the first k output assignments. By making the definition of the delay mutation more complicated, one can work this through and have full duality in the linear approach.

In Example 3 below, we demonstrate that changing the timing of the outputs may change the branching nature of the circuit.

Example 3 In Figure 1 below, we describe a circuit $\mathcal{C}$ and its 1-cycle delay $\mathcal{C}'$ (note that $I = \emptyset$). The output of a state xy in $\mathcal{C}'$ is y. While $L(\mathcal{C}') = \{a \cdot w : w \in L(\mathcal{C})\}$, the branching structure of $\mathcal{C}'$ is substantially different than that of $\mathcal{C}$, and $\mathcal{C}'$ is does not simulate a circuit obtained from $\mathcal{C}$ by adding an initial state labeled by a.

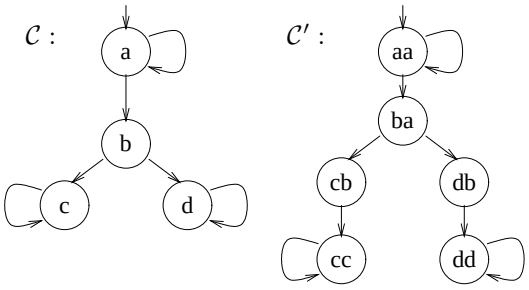

Figure 3: Introducing a delay changes the branching structure.

We now turn to study vacuous versus loose satisfaction. Here too, we do not have full duality, and the implication that holds is in the opposite direction from that one in Theorem 2. We discuss below the intuition for this difference.

Theorem 3 *Consider a specification $\mathcal{S}$ and an implementation $\mathcal{I}$ for it. Let $\mathcal{S}'$ be the mutation of $\mathcal{S}$ obtained by removing behaviors that depend on x, and let $\mathcal{I}'$ be the mutation of $\mathcal{I}$ obtained by freeing x. If $x \in I$, then $\mathcal{I} \subseteq \mathcal{S}'$ implies that $\mathcal{I}' \subseteq \mathcal{S}$ and $\mathcal{I} \leq \mathcal{S}'$ implies that $\mathcal{I}' \leq \mathcal{S}$. If $x \in C \setminus O$ and $O \subseteq C$, then $\mathcal{I} \leq \mathcal{S}'$ implies that $\mathcal{I}' \leq \mathcal{S}$.*

Proof: Let $\mathcal{I} = \langle I, O, C, \theta_{\mathcal{I}}, \delta_{\mathcal{I}}, \rho_{\mathcal{I}} \rangle$, $\mathcal{I}' = \langle I, O, C, \theta'_{\mathcal{I}}, \delta'_{\mathcal{I}}, \rho'_{\mathcal{I}} \rangle$, $\mathcal{S} = \langle I, O, C, \theta_{\mathcal{S}}, \delta_{\mathcal{S}}, \rho_{\mathcal{S}} \rangle$, and $\mathcal{S}' = \langle I, O, C, \theta'_{\mathcal{S}}, \delta'_{\mathcal{S}}, \rho'_{\mathcal{S}} \rangle$.

We start with the case $x \in I$. Assume that $\mathcal{I} \subseteq \mathcal{S}'$, and let $w' = w'_0, w'_1, \ldots \in 2^{I \cup O}$ be a computation of $\mathcal{I}'$. Let $w'_j = i'_j \cup o'_j$ with $i'_j \in 2^I$ and $o'_j \in 2^O$. Then, there is a computation $w = w_0, w_1, \ldots \in 2^{I \cup O}$ of $\mathcal{I}$ such that$w_j = i_j \cup o'_j$ for $i_j \in \{i'_j \setminus \{x\}, i'_j \cup \{x\}\}$. By the assumption, w is also a computation of $\mathcal{S}'$. Now, by the definition of $\mathcal{S}'$, it follows that all computations $w'' = w''_0, w''_1, \ldots \in 2^{I \cup O}$, for which $w''_j = i''_j \cup o_j$ with $i''_j \in \{i_j \setminus \{x\}, i_j \cup \{x\}\}$ are computations of $\mathcal{S}$. Since for all $j \geq 0$ we have that $i'_j \in \{i_j \setminus \{x\}, i_j \cup \{x\}\}$, independent of whether $i_j = i'_j \setminus \{x\}$ or $i'_j \cup \{x\}$, it follows that w' is also a computation of $\mathcal{S}$ and we are done.

Assume now that $\mathcal{I} \leq \mathcal{S}'$. Let $\mathcal{H}$ be an initial simulation relation from $\mathcal{I}$ to $\mathcal{S}'$. We claim that $\mathcal{H}$ is an initial simulation relation from $\mathcal{I}'$ to $\mathcal{S}$. Since $\rho_I = \rho'_I$ and $\rho_{\mathcal{S}} = \rho'_S$, proving that $\mathcal{H}$ is a simulation from $\mathcal{I}'$ to $\mathcal{S}$, we only have to prove that for all s' and s such that $\mathcal{H}(s', s)$, and for all $i \in 2^I$ and $t' \in \delta'_I(s', i)$ there is $t \in \delta_{\mathcal{S}}(s, i)$ such that $\mathcal{H}(t', t)$. In first prove that $\mathcal{H}$ is initial with respect to $\mathcal{I}'$ and S; that is, for all $i \in 2^I$ and $t' \in \theta'_{\mathcal{I}}(i)$ there is $t \in \theta_{\mathcal{S}}(i)$ such that $\mathcal{H}(t', t)$.

Consider an input assignment $i \in 2^I$ and a state $t \in \theta'_{\mathcal{I}}(i)$. By the definition of $\theta'_{\mathcal{I}}$, we know that $t \in \theta_{\mathcal{I}}(i \setminus \{x\}) \cup \theta_{\mathcal{I}}(i \cup \{x\})$. Assume without loss of generality that $t \in \theta_{\mathcal{I}}(i \setminus \{x\})$. Since $\mathcal{H}$ is a simulation from $\mathcal{I}$ to $\mathcal{S}'$, there is a state $t' \in \theta'_{\mathcal{S}}(i \setminus \{x\})$ such that $\mathcal{H}(t, t')$. By the definition of $\theta'_{\mathcal{S}}$, we know that $t' \in \theta_{\mathcal{S}}(i \setminus \{x\}) \cap \theta_{\mathcal{S}}(i \cup \{x\})$. Since either $i \setminus \{x\} = i$ or $i \cup \{x\} = i$, we got that there is $t' \in \theta_{\mathcal{S}}(i)$ such that $\mathcal{H}(t, t')$, and we are done.

We now proceed to prove that $\mathcal{H}$ is a simulation. Let s and s' such that $\mathcal{H}(s, s')$. Consider an input assignment $i \in 2^I$ and a state $t \in \delta'_I(s, i)$. By the definition of $\delta'_{\mathcal{I}}$, we know that $t \in \delta_{\mathcal{I}}(s, i \setminus \{x\}) \cup \delta_{\mathcal{I}}(s, i \cup \{x\})$. Assume without loss of generality that $t \in \delta_{\mathcal{I}}(s, i \setminus \{x\})$. Since $\mathcal{H}$ is a simulation from $\mathcal{I}$ to $\mathcal{S}'$, there is a state $t' \in \delta'_{\mathcal{S}}(s', i \setminus \{x\})$ such that $\mathcal{H}(t, t')$. By the definition of $\delta'_{\mathcal{S}}$, we know that $t' \in \delta_{\mathcal{S}}(s', i \setminus \{x\}) \cap \delta_{\mathcal{S}}(s', i \cup \{x\})$. Since either $i \setminus \{x\} = i$ or $i \cup \{x\} = i$, we got that there is there is $t' \in \delta_{\mathcal{S}}(s, i)$ such that $\mathcal{H}(t, t')$, and we are done.

We proceed to the case $x \in C$ and $O \subseteq C$. Assume that $\mathcal{I} \leq \mathcal{S}'$, and let $\mathcal{H} \subseteq 2^C \times 2^C$ be the maximal simulation relation from $\mathcal{I}$ to $\mathcal{S}'$. We claim that $\mathcal{H}' = \{\langle t, t' \rangle : \mathcal{H}(t \setminus \{x\}, t')$ or $\mathcal{H}(t \cup \{x\}, t')\}$ is an initial simulation from $\mathcal{I}'$ to $\mathcal{S}$. We first prove that $\mathcal{H}'$ is initial. Consider an input assignment $i \in 2^I$ and a state $t \in \theta'_{\mathcal{I}}(i)$. By the definition of $\theta'_{\mathcal{I}}$, we know that $\{t \setminus \{x\}, t \cup \{x\}\} \cap \theta(i) \neq \emptyset$. Assume without loss of generality that $t \setminus \{x\} \in \theta_{\mathcal{I}}(i)$. Since $\mathcal{I} \leq \mathcal{S}'$, there is $t' \in \theta'_{\mathcal{S}}(i)$ such that $\mathcal{H}(t \setminus \{x\}, t')$. By the definition of $\theta'_{\mathcal{S}}$, we have that $t' \in \theta_{\mathcal{S}}(i)$. Also, by the definition of $\mathcal{H}'$, we also have that $\mathcal{H}'(t, t')$, and we are done.

It is left to prove that $\mathcal{H}'$ is a simulation. Let s and s' such that $\mathcal{H}(s, s')$. Consider an input assignment $i \in 2^I$ and a state $t \in \delta'_I(s, i)$. By the definition of $\delta'_{\mathcal{I}}$, we know that $\{t \setminus \{x\}, t \cup \{x\}\} \cap (\delta_{\mathcal{I}}(s \setminus \{x\}, i) \cup \delta_{\mathcal{I}}(s \cup \{x\}, i)) \neq \emptyset$ Assume without loss of generality that $t \setminus \{x\} \in \delta_{\mathcal{I}}(s \setminus \{x\}, i)$. Since $\mathcal{H}$ is a simulation from $\mathcal{I}$ to $\mathcal{S}'$, there is a state $t' \in \delta'_{\mathcal{S}}(s', i)$ such that $\mathcal{H}(t, t')$. By the definition of $\delta'_{\mathcal{S}}$, we know that $t' \in \delta_{\mathcal{S}}(s' \setminus \{x\}, i) \cap \delta_{\mathcal{S}}(s' \cup \{x\}, i)$. Since both $\mathcal{H}(t \setminus \{x\}, t')$ and $\mathcal{H}(t \cup \{x\}, t')$, we are done. Finally, since $O \subseteq C$, and $x \notin O$, we have that $\rho'_{\mathcal{I}}(s \setminus \{x\}) = \rho'_{\mathcal{I}}(s \cup \{x\}) = \rho_{\mathcal{I}}(s)$. Also, $\rho_{\mathcal{S}}(s') = \rho'_{\mathcal{S}}(s')$. Hence, $\mathcal{H}'(s, s')$ implies that $\rho'_{\mathcal{I}}(s) = \rho_{\mathcal{S}}(s')$.

□

Note that duality does not hold for the linear approach in the case x is a control signal. The reason is that a path in $\mathcal{I}'$ need not correspond to a path in $\mathcal{I}$. If, for example, $\delta_{\mathcal{I}}(s_0, i_1) = s_1 \setminus \{x\}$, $\delta_{\mathcal{I}}(s_1 \cup \{x\}, i_2) = s_3$, and $\{s_3 \setminus \{x\}, s_3 \cup \{x\}\} \cap \delta_{\mathcal{I}}(s_1 \setminus \{x\}, i_2) = \emptyset$, then s_3 is reachable from s_0 in $\mathcal{I}'$ by reading i_1 and i_2, but neither $s_3 \setminus \{x\}$ nor $s_3 \cup \{x\}$ are reachable from s_0 in $\mathcal{I}$ by reading i_1 and i_2. In the branching approach, on the other hand, simulation guarantees that intermediate states are related too, and hence duality exists.

Remark 3 The challenges in relating the mutations in Theorem 3 are related to the challenges in reachability analysis in abstractions. Indeed, the mutant specification $\mathcal{S}'$ corresponds to an under-appoximation of $\mathcal{S}$, and the mutant implementation $\mathcal{I}'$ corresponds to an over-approximation of $\mathcal{I}$. Theorem 3 relates satisfaction of an under-approximation of the specification to satisfaction by an over-approximation of the implementation. □

In Example 4, we show that the other direction (that is, $\mathcal{I}' \models \mathcal{S}$ implies $\mathcal{I} \models \mathcal{S}'$) does not hold, in both the linear and branching approaches, even if we apply bisimulation minimization to the specification before we remove behaviors.

Example 4 In Figure 4 below, we describe an implementation $\mathcal{I}$ and a specification $\mathcal{S}$ with $I = \{x\}$, $O = \{v\}$, and $C = \{y\}$. The value of y is described at the top of the states and the value of v at the bottom. Note that $\mathcal{I} = \mathcal{S}$.

Adding to $\mathcal{I}$ behaviors that dualize the behavior of x results in an implementation $\mathcal{I}'$ in which an initial transition to both $y = 0$ and $y = 1$ is enabled with both $x = 0$ and $x = 1$. Since the two states $y = 0$ and $y = 1$ have $v = 0$, we have that $\mathcal{I}' \models \mathcal{S}$ (in both approaches). On the other hand, the specification $\mathcal{S}'$ obtained by removing behaviors that depend on x is not receptive, thus $\mathcal{I} \not\models \mathcal{S}'$. The example may seem weak, as the states $y = 0$ and $y = 1$ of $\mathcal{S}$ are bisimilar. Note, however, we could strengthen it by adding to $\mathcal{S}$ different behaviors from the state $y = 0$ and $y = 1$. For example, we could add a transition from $y = 0$ to a new state in which the value of v is 1. Adding behaviors to the specification does not violate the satisfaction of $\mathcal{S}'$ in $\mathcal{I}'$, and shows that the problem exists even if we require the specification not to have bisimilar states.

A counterexample for the case $x \in C$ is similar. Let $I = \{z\}$, $O = \{v\}$, and $C = \{x\}$, and let $\mathcal{I} = \mathcal{S}$ go to both $x = 0$ and $x = 1$ on input $z = 0$. Also, the value of v agrees with that of x. While the mutant $\mathcal{I}'$ coincides with $\mathcal{I}$, the mutant $\mathcal{S}'$ is not receptive, as the states $x = 0$ and $x = 1$ have different outputs. One can strengthen the example by adding v to C, making $O \subseteq C$.

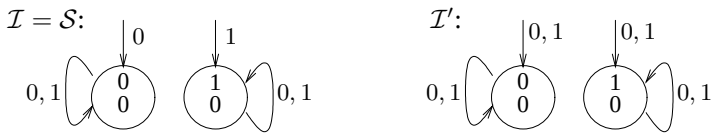

Figure 4: A counter example for the direction $\mathcal{I}' \models \mathcal{S}$ implies $\mathcal{I} \models \mathcal{S}'$

Recall that some of the dualities are not full. We considered both the linear and the branching approaches to specification and it turned out that in some cases, duality exists only in one of the approaches. Intuitively, the linear approach has the advantage that mutations may change not only the set of computations, but also the branching structure of the circuit. On the other hand, the branching approach has the advantage of being local, which is useful for mutations parameterized by control signals.

We also considered both directions of the implication in Definition 1. It turns out that full duality goes only with diverse satisfaction. On the other hand, while vacuous satisfaction implies loose satisfaction, the other direction does not always hold. To see the intuition behind this, recall that vacuous satisfaction hints on absence of behaviors that the designer expects to find in the implementation – behaviors that would cause the mutant specification not to be satisfied. Applying the dual mutation to the implementation adds these behaviors. On the other hand, loose satisfaction suggests that we can add to the implementation behaviors that agree with original behaviors on everything but x. These new behaviors can be simulated (only) by behaviors of the specification that do depend on x, so vacuous satisfaction is not guaranteed.

Finally, the dualities presented in Theorems 1, 2, and 3 refer to persistent faults. The dualities are maintained in the presence of a controlled injection of faults if the faults are injected to the specification and the implementation at the *same cycles*. If, for example, x is flipped in $\mathcal{S}'$ in the 5th cycle, it has to be flipped in $\mathcal{I}'$ also at the 5th cycle. Note that a persistent fault is a special case of the above. On the other hand, if the injection of faults in the specification and implementation follows the same pattern-circuit $\mathcal{C}_{mut}$ (see Section 3.4), but the language of $\mathcal{C}_{mut}$ is not a singleton, the dualities are, in general, no longer valid.

5 Discussion and Experimental Results

We described a state-based approach to vacuity and coverage and showed that when the implementation and the specification are both modeled as circuits, vacuity and coverage are dual. We now describe some applications of this duality.

First, amongst the new coverage metrics and vacuity definitions that duality suggests, there is one that is of particular interest. As discussed in previous work [11], coverage metrics that are induced from simulation-based coverage often involve a disabling of a behavior. For example, an adoption of *branch coverage* to model checking involves model checking mutations of the implementations in which branches are disabled. Thus, the mutant implementation

has fewer behaviors and is guaranteed to satisfy all universal specifications (i.e., specifications that apply to all behaviors, as in linear temporal logic) that the original implementation has satisfied. This is problematic, as we need to asses the role of a component in the satisfaction of a specification that is clearly satisfied in a mutant implementation without this component. This problem has been addressed [11] by checking whether the satisfaction of the specification in the mutant implementation has become vacuous. The duality between coverage and vacuity we present here suggests that this solution coincides with one that captures better our intuition of coverage: a behavior of the implementation is covered by a universal specification if *adding* to the implementation the dual behavior results in a mutant implementation that no longer satisfies the specification.

In addition, the duality suggests, for the first time, a *feasible, automatable methodology for tightening specifications*. One of the popular examples for the effectiveness of coverage is an implementation containing a computation in which a request is followed by two grants. Such a computation satisfies the specification "every request is followed by a grant" and coverage information, which reveals the fact that each of the two grants can be flipped, is likely to urge the designer to tighten the specification to one that requires a correspondence between the requests and the grants. All this is very nice, but the way from the actual output of the coverage process (a list of uncovered elements) to the realization that some uncovered elements are related (as is the case with the two grants), and then to a tighter specification, is long. Returning more meaningful coverage information to the user is a challenging problem. One approach is to arrange the information gathered by the coverage algorithm in helpful ways, such as returning computations that contain many uncovered states. The approach taken here is different. Applying the dual mutant to the specification automatically generates a tighter specification. In the above case, the designer works with a "flip grant once" mutation, and applying the dual flip to the specification teaches him that some computations also satisfy a specification in which requests are followed by two grants. This is not the end of the story, as the application of the dual mutation results in a mutant specification that is satisfied in the implementation. Thus, it is not going to immediately detect bugs. The mutant specification, however, and the fact it is still satisfied, is of great help to the designer in manually tightening the specification, as it directly points to the unexpected behavior.[5] In this sense, the dual specification plays a role that is similar to the one played by a counterexample – while it does not directly suggest a repair to an erroneous implementation, it is of great help in understanding where the error is. We plan to investigate this relation between tighter specifications and counterexamples further and check whether the duality between coverage and vacuity studied here is derived from a more general duality, namely the one between program repair and tightening of specifications.

5.1 Experimental Results

We conclude by illustrating two practical use of the duality, specifically of Theorems 1 and 2.

5.1.1 Illustration of Vacuity from Coverage

The first experiment illustrates how useful vacuity information can be obtained *for free* once coverage has been checked. On an implementation of the Peripheral Interconnect Bus (PI-BUS) from the texas-97 benchmarks [25], we checked an LTL property of the form $\mathbf{G}(\phi \implies x = 1)$, and found it to hold even when a persistent flip was performed on x. This implied that the implementation also satisfied $\mathbf{G}(\phi \implies x = 0)$, and thus that $\mathbf{G}\neg\phi$, computing the same result that vacuity checking would.

We elaborate on this experiment below. The specification of interest, denoted $\mathcal{S}$, checks the read operation: in the absence of an error or timeout, the 31st bit of the master's data buffer should have the same value as the 31st bit of the data that the slave sends out at the same cycle. In LTL, $\mathcal{S}$ is the formula

$$\mathbf{G}\quad\Big((\texttt{slave.state} = \texttt{ADDRESS} \vee \texttt{slave.state} = \texttt{DATA\_WAIT}) \wedge (\texttt{slave.acknowledgment} = \texttt{RDY})$$
$$\wedge\quad (\texttt{slave.READ} = 1) \wedge (\texttt{slave.dataout}[31] = 1) \wedge (\texttt{slave.SEL} = 1) \Rightarrow (\texttt{master.datain}[31] = 1)\Big)$$

[5] Our framework applies to other settings in which the implementation and the specification are described using similar formalism. In particular, in scenario-based specifications [15], mutating a specification directly suggests a repair to the implementation.

where slave.state is the state of the slave interface FSM, slave.acknowdgement is the acknowledgement signal from slave to master, slave.READ is the read request from master to slave, and slave.SEL indicates whether this slave is selected by the master.

Let μ be the mutation that persistently flips the value of master.datain[31] and $\mathcal{I}$ be the original implementation. In our coverage analysis, $\mathcal{I}_\mu \models \mathcal{S}$. By Theorem 1, we have $I \models S_{\widetilde{\mu}}$, where $\widetilde{\mu}$ is the mutation of flipping master.datain[31] in $\mathcal{S}$. Combining $\mathcal{S}$ and $\mathcal{S}_{\widetilde{\mu}}$, we get a new specification $\mathcal{S}'$:

$$\mathbf{G} \quad \neg \quad \Big((\texttt{slave.state} = \texttt{ADDRESS} \vee \texttt{slave.state} = \texttt{DATA\_WAIT}) \wedge (\texttt{slave.acknowledgment} = \texttt{RDY}) \\ \wedge \quad (\texttt{slave.READ} = 1) \wedge (\texttt{slave.dataout}[31] = 1) \wedge (\texttt{slave.SEL} = 1) \Big)$$

which is satisfied by $\mathcal{I}$. This is the same as the antecedent of $\mathcal{S}$ being always false, and hence indicates vacuous satisfaction of $\mathcal{S}$ in $\mathcal{I}$. Specifically, $\mathcal{S}$ is vacuously satisfied because slave.SEL is never high in the implementation $\mathcal{I}$.

5.1.2 Monitoring Faults in a Router Design

The setting for our second experiment is the synthesis of circuits that monitor themselves for the occurrence of physical faults, such as soft (transient) flips in latches. The goal is to synthesize monitors (for, say, temporal-logic properties) that detect such faults locally within a module, before the fault propagates all the way to the output of the circuit, so that local error recovery can be performed.

Our case study was a version of a chip multiprocessor router designed in Verilog [24]. The router has two input and two output ports; a block diagram of the router is given in Figure 5. There are four main modules of the router. The first, called the *input controller*, buffers incoming flits, determines their destination port, and interacts with an *arbiter* module in order to reserve an output port. In Peh's design [24], the reservation of an output port is performed on receipt of a head flit. Thereafter, all body flits and tail flits are directed to the output port without incurring any further latency. The arbiter is fair, assigning priorities to input ports based on a simple round-robin scheme. The remaining modules are the *encoder* and *crossbar*, which contain logic to copy flits to the output port from the input port that has been assigned that output port. The router's function is to direct incoming packets, called *flits*, to the correct output

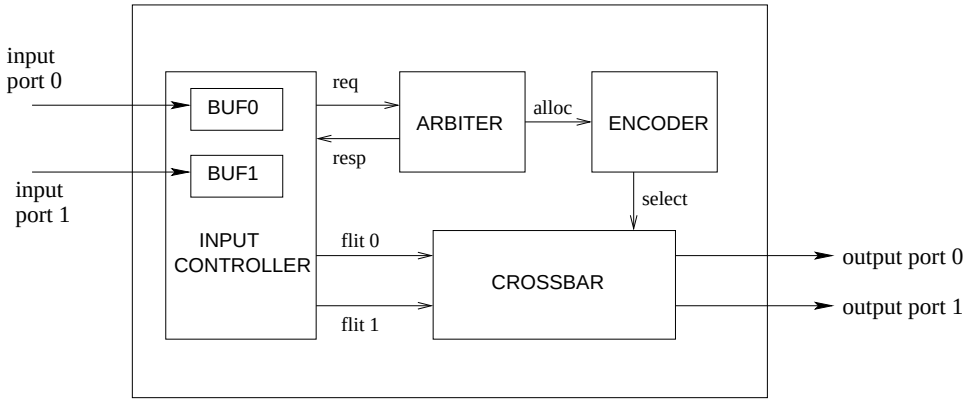

Figure 5: **Chip multiprocessor router block diagram.** There are four main modules: the input controller, the arbiter, the encoder, and the crossbar. Not all interconnections are shown.

port, within a latency bound of 8 cycles. The arbiter module of the router mediates access to the output ports based on a fair round-robin scheme.

The Verilog was automatically translated to an SMV model, which was instrumented to latch an incoming flit at a single, non-deterministically chosen cycle on each port; for port i, we track this latched flit with the control signal latched_input_flit$_i$. The control signal latched_a_flit$_i$ is a flag that starts at 0 and is permanently set to 1

after an incoming flit is latched. The output flit at port j, at any cycle, is stored in $\texttt{flit\_out}_j$. The overall router specification can thus be written as the LTL property[6]

$$\mathbf{G}\left(\texttt{latched\_a\_flit}_i \implies \bigvee_{k=1}^{8} \mathbf{X}^k(\texttt{latched\_input\_flit}_i = \texttt{flit\_out}_j)\right)$$

If a transient bit flip μ occurs in the priority state bit of the arbiter module, the above property will not hold because the flip makes the arbiter unfair long enough that the flit arrives at the output port later than the specified latency bound. This was verified using Cadence SMV. However, the transient fault μ did not violate the following local property specified on the request/grant lines of arbiter module for port j: $\mathbf{G}(\texttt{request}_i \implies \bigvee_{k=1}^{8} \mathbf{X}^k \texttt{grant}_i)$.

Treating the arbiter module as the implementation $\mathcal{I}$, the above temporal-logic specification as $\mathcal{S}$, and a 4-cycle delay fault in the $\texttt{grant}_i$ output signal as a mutation ν, we found that $\mathcal{I}_\nu \models \mathcal{S}$. Using Theorem 2, we obtain $\mathcal{I} \models \mathcal{S}_{\tilde{\nu}}$, where $\tilde{\nu}$ makes $\texttt{grant}_i$ premature by 4 cycles in $\mathcal{S}$, yielding the property $\mathbf{G}(\texttt{request}_i \implies \bigvee_{k=1}^{4} \mathbf{X}^k \texttt{grant}_i)$. We found that this property catches the transient bit flip μ, i.e., $\mathcal{I}_\mu \not\models \mathcal{S}_{\tilde{\nu}}$. Moreover, $\mathcal{I} \models \mathcal{S}_{\tilde{\nu}}$. We can thus use $\mathcal{S}_{\tilde{\nu}}$ to synthesize a monitor that catches transient errors in the arbiter module.

In general, the effectiveness of using duality to tighten specifications depends greatly on the choice of the mutation used and the available duality results. For future work, we plan to study more mutations and extend the set of duality results.

Acknowledgments

This work was supported in part by the Gigascale Systems Research Focus Center, one of five research centers funded under the Focus Center Research Program, a Semiconductor Research Corporation program.

References

[1] M. Abadi and L. Lamport. The existence of refinement mappings. *TCS*, 82(2):253–284, 1991.

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

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

[4] L. Bening and H. Foster. *Principles of verifiable RTL design – a functional coding style supporting verification processes*. Kluwer Academic Publishers, 2000.

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

[6] D. Bustan, A. Flaisher, O. Grumberg, O. Kupferman, and M.Y. Vardi. Regular vacuity. In *Proc. 13th CHARME*, LNCS 3725, pages 191–206. 2005.

[7] M. Chechik and A. Gurfinkel. Extending extended vacuity. In *Proc. 5th Int. Conf. on Formal Methods in Computer-Aided Design*, LNCS 3312, pages 306–321, 2004.

[8] H. Chockler and O. Kupferman. Coverage of implementations by simulating specifications. In *Proc. 2nd IFIP TCS*, volume 223, *IFIP Conf. Proceedings*, pages 409–421. Kluwer Acad. Publishers, 2002.

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

[6] There is one such property for each of the four input-output port pairs (i, j), but by symmetry, we need to check it for only such pair.

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

[11] H. Chockler, O. Kupferman, and M.Y. Vardi. Coverage metrics for formal verification. In *Proc. 12th CHARME*, LNCS 2860 , pages 111–125, 2003.

[12] E. M. Clarke, O. Grumberg, K.L. McMillan, and X. Zhao. Efficient generation of counterexamples and witnesses in symbolic model checking. In *Proc. 32th DAC*, pages 427–432. IEEE Computer Society, 1995.

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

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

[15] D. Harel, H. Kugler, and A. Pnueli. Synthesis revisited: Generating statechart models from scenario-based requirements. In *Formal Methods in Software and System Modeling*, LNCS 3393 , pages 309–324, 2005.

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

[17] O. Kupferman. Sanity checks in formal verification. In *17th CONCUR*, LNCS 4137, pages 37–51, 2006.

[18] O. Kupferman and M.Y. Vardi. Modular model checking, *Proc. Compositionality Workshop*, LNCS 1536, pages 381-401, 1998.

[19] O. Kupferman and M.Y. Vardi. Vacuity detection in temporal model checking. *STT& T*, 4(2):224–233, 2003.

[20] R.P. Kurshan. *Computer Aided Verification of Coordinating Processes*. Princeton Univ. Press, 1994.

[21] S.S. Lam and A.U. Shankar. Protocol verification via projection. *IEEE TSE*, 10:325–342, 1984.

[22] N. A. Lynch and M.R. Tuttle. Hierarchical correctness proofs for distributed algorithms. In *Proc. 6th PODC*, pages 137–151, 1987.

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

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

[25] Texas-97 benchmarks, `http://embedded.eecs.berkeley.edu/Respep/Research/vis/texas-97/`.

[26] S. Tasiran and K. Keutzer. Coverage metrics for functional validation of hardware designs. *IEEE Design and Test of Computers*, 18(4):36–45, 2001.

[27] M.Y. Vardi and P. Wolper. Reasoning about infinite computations. *Information & Computation*, 115(1):1–37, 1994.

A Mutations

A.1 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 (forcing x to get stuck at 1 is similar) by acting as if $x = 0$ regardless of its actual value. Thus the mutation is similar to the one that flips the value of x, only that here the value is flipped only when $x = 1$. As there, the mutant circuit $\mathcal{C}'$ stays receptive. Note that unlike the mutation described in Section 3.1.3, here we do not disable transitions after which the value of x is 1, but rather we flip the value of x in the destination state. Formally, $\mathcal{C}' = \langle I, O, C, \theta', \delta', \rho \rangle$, where

- If $x \in I$, then $\rho' = \rho$, and for all $s \in 2^C$ and $i \in 2^I$, we have $\theta'(i) = \theta(i \setminus \{x\})$ and $\delta'(s, i) = \delta(s, i \setminus \{x\})$.

- If $x \in O \setminus C$, then $\theta' = \theta$, $\delta' = \delta$, and for all $t \in 2^C$, we have $\rho'(t) = \rho(t) \setminus \{x\}$.
- If $x \in C$, then $\rho' = \rho$ and for all $s \in 2^C$ and $i \in 2^I$, we have $\theta'(i) = \{t \setminus \{x\} : t \in \theta(i)\}$, and $\delta'(s,i) = \{t \setminus \{x\} : t \in \delta(s,i)\}$. Thus, whenever $\mathcal{C}$ has a transition to a state t, the mutation $\mathcal{C}'$ goes instead to $t \setminus \{x\}$.

Sticking x at 1 is similar, and is done by replacing x by $x \cup \{1\}$.

For the setting of a temporal-logic formula, sticking the value of $x \in I \cup O$ to 0 amounts to negating all the positive occurences of x. Likewise, sticking x to 1 amounts to negating all its negative occurrences.

A.2 Introducing Delay and Prematurness

Given a circuit $\mathcal{C}$ and a number $k \geq 1$ describing the delay, the mutation of $\mathcal{C}$ in which a k-cycle delay is introduced is $\mathcal{C}' = \langle I, O, C', \theta', \delta', \rho' \rangle$, where

- $C' = C \cup (O \times \{1, \ldots, k\})$. We refer to states of $\mathcal{C}'$ as tuples $\langle s, \sigma_1, \ldots, \sigma_k \rangle \in 2^C \times 2^O \times \cdots \times 2^O$, where σ_1 maintains the output of the state visited in the previous cycle, σ_2 maintains the output of the state visited before that, and so on, until σ_k, which maintains the output of the state visited before k cycles.
- For all $i \in 2^I$, we have that $\theta'(i) = \theta(i) \times 2^O \times \cdots \times 2^O$. Thus, the output in the first k cycles is arbitrary. The user can choose a different initialization to the new control signals.
- For all $\langle s, \sigma_1, \ldots, \sigma_k \rangle \in 2^C \times 2^O \times \cdots \times 2^O$, and $i \in 2^I$, we have that $\delta'(\langle s, \sigma_1, \ldots, \sigma_k \rangle, i) = \delta(s,i) \times \{\langle \rho(s), \sigma_1, \ldots, \sigma_{k-1} \rangle\}$.
- For all $\langle s, \sigma_1, \ldots, \sigma_k \rangle \in 2^C \times 2^O \times \cdots \times 2^O$, we have that $\rho'(\langle s, \sigma_1, \ldots, \sigma_k \rangle) = \sigma_k$.

Given a circuit $\mathcal{C}$ and a number $k \geq 1$ describing the prematureness, the mutation of $\mathcal{C}$ in which a k-cycle prematureness is introduced is $\mathcal{C}' = \langle I, O, C', \theta', \delta', \rho' \rangle$, where

- $C' = C \cup (O \times \{1, \ldots, k\})$. We refer to states of $\mathcal{C}'$ as tuples $\langle s, \sigma_1, \ldots, \sigma_k \rangle \in 2^C \times 2^O \times \cdots \times 2^O$, where σ_1 maintains a guess for the output of the next state of $\mathcal{C}$, σ_2 maintains a guess for the output of the state after it, and so on.
- For all $i \in 2^I$, we have that $\theta'(i) = \theta(i) \times 2^O \times 2^O \times \cdots \times 2^O$. Thus, the initial transitions nondeterministically guess the output for the first k states of $\mathcal{C}$.
- For all $\langle s, \sigma_1, \ldots, \sigma_k \rangle \in 2^C \times 2^O \times \cdots \times 2^O$, and $i \in 2^I$, we have that $\delta'(\langle s, \sigma_1, \ldots, \sigma_k \rangle, i)$ is $(\delta(s,i) \cap \rho^{-1}(\sigma_1)) \times \{\langle \sigma_2, \ldots, \sigma_k \rangle\} \times 2^O$. Thus, $\mathcal{C}'$ goes only to successors of s for which the guess maintained in σ_1 turns out to be valid, and it updates the guesses by shifting them and adding a guess for the state of $\mathcal{C}$ to be reached after k transitions.
- For all $\langle s, \sigma_1, \ldots, \sigma_k \rangle \in 2^C \times 2^O \times \cdots \times 2^O$, we have that $\rho'(\langle s, \sigma_1, \ldots, \sigma_k \rangle) = \sigma_k$. Thus, the output of a state is the guess for the output of the state of $\mathcal{C}$ to be reached after k transitions.

Note that the output of $\mathcal{C}'$ is indeed k-cycle prematureness of the output of $\mathcal{C}$. For example, when $k = 1$, we have that $s_0, s_1, s_2, \ldots \in 2^C$ is a path in $\mathcal{C}$ with output $o_0, o_1, o_2, \ldots$ iff $\langle s_0, o_1 \rangle, \langle s_1, o_2 \rangle, \langle s_2, o_3 \rangle, \ldots \in 2^C \times 2^O$ is a path in $\mathcal{C}'$ with output $o_1, o_2, o_3, \ldots$.

Note also that when a state of $\mathcal{C}'$ maintains a wrong guess, it may not have outgoing transitions. Thus, $\mathcal{C}'$ may not be receptive. We still refer to $\mathcal{C}'$ as a receptive circuit, as for all states $s \in 2^C$ and computation π that starts at s, the circuit $\mathcal{C}'$ contains a state $\langle s, \sigma_1, \ldots, \sigma_k \rangle \in 2^C \times 2^O \times \cdots \times 2^O$ that maintains a guess that is correct for π, and along which the interaction of $\mathcal{C}'$ with the environment does not get stuck.

A.3 Controlled Injection of Mutations

In this section we give the details of the generalization of our definitions of mutations to make it possible to apply mutations one on top of the other and to control the pattern (over time) in which mutations happen.

Since the method involves applications of faults one on top of the other, it is convenient to assume that the mutant circuit has the same set of signals as the original one. This is true for most of the mutations described above, and it is not hard to adjust the definition of mutations or the construction here to the case the set of signals does change. Another technical assumption we make for is the existence of a function $apply$ that takes an initialization function, a transition function, or an output function, and a type of mutation, and returns the mutated initialization function, transition function, or output functions. It is not hard to see that the descriptions of mutations in Sections 3.1, 3.2, and 3.3 induce such a function.

For some sets of mutations, the order of application is not important. For example, if we remove behaviors that depend on x and then remove behaviors that depend on y, the result is identical to the circuit we get if we first remove behaviors that depend on y and then remove behaviors that depend on x. In general, however, the order is important. For example, if $x \in C$ and the states s and t are such that $\{t \setminus \{x\}, t \cup \{x\}\} \subseteq \delta(s \setminus \{x\}, i) \cap \delta(s \cup \{x\}, i)$, for some input assignment $i \in 2^I$, and we first remove behaviors that depend on x and then stick x to 0 by disabling transitions that go to states in which the value of x is 1, then $t \setminus \{x\} \in \delta'(s \setminus \{x\})$. But if we first stick x to 0 and then remove behaviors that depend on x, then $t \setminus \{x\} \notin \delta'(s \setminus \{x\})$. Note that this example is for two mutations both of which remove behaviors.

Accordingly, the fault pattern that the user specifies describes not only the set of mutations that occur at each moment in time, but also the order in which they are applied. Let M be a set of signals that encode ordered subsets of the set of all possible mutations. A *mutation pattern* is a safety language $\mathcal{L}_{mut}$ over M. For example, if there are two possible mutations, m_1 and m_2, then M encodes elements of the set $\{\langle\,\rangle, \langle m_1\rangle, \langle m_2\rangle, \langle m_1, m_2\rangle, \langle m_2, m_1\rangle\}$ and $\mathcal{L}_{mut}$ may be the set of all words in which every subword of length 5 contains a letter in $\{\langle m_1\rangle, \langle m_1, m_2\rangle, \langle m_2, m_1\rangle\}$ and a letter in $\{\langle m_2\rangle, \langle m_1, m_2\rangle, \langle m_2, m_1\rangle\}$. Such a language corresponds to a pattern in which a fault inducing m_1 occurs in all 5-cycle windows, the same for a fault inducing m_2, and the fault may occur at the same cycle, in which case the order of application is not important. A natural way to describe $\mathcal{L}_{mut}$ is by a looping automaton over 2^M. Since we are going to combine the fault pattern with the original circuit, we describe this automaton by means of a circuit $\mathcal{C}_{mut}$ with input M and output $\{\top\}$ such that $\mathcal{C}_{mut}$ outputs $\top$ as long as the sequence of mutations read so far can be extended to a word in $\mathcal{L}_{mut}$. Formally, $\mathcal{L}_{mut}$ is given by a *mutation-pattern circuit* $\mathcal{C}_{mut} = \langle M, \{\top\}, Q, \theta_{mut}, \delta_{mut}, \rho_{mut}\rangle$ in which a word w over M leads to a state q for which $\rho_{mut}(q) = \{\top\}$ iff w can be extended to a word in $\mathcal{L}_{mut}$.

For a circuit $\mathcal{C}$ and a mutation-pattern circuit $\mathcal{C}_{mut} = \langle M, \{\top\}, Q, \theta_{mut}, \delta_{mut}, \rho_{mut}\rangle$, we define the *controlled-fault version of* $\mathcal{C}$ as the circuit $\mathcal{C}' = \langle I, O, C \cup Q, \theta', \delta', \rho'\rangle$ defined below. Let $\{m_1, \ldots, m_k\}$ be the set of all possible mutations. For an ordered sequence $\sigma = \langle m_{j_1}, m_{j_2}, \ldots, m_{j_l}\rangle$ of mutations, and an initialization function θ, we use $apply(\theta, \sigma)$ as an abbreviation of $apply(\cdots\ apply(apply(\theta, m_{j_1}), m_{j_2}) \cdots, m_{j_l})$. That is, $apply(\theta, \sigma)$ is the initialization function obtained from θ by first applying m_{j_1} on it, then m_{j_2}, and so on until m_{j_l}. The notation for the transition function and the output function is similar. Now,

- $\theta'(i) = \bigcup_{\sigma \in 2^M} \{c' \cup q' : c' \in apply(\theta, \sigma)(i), q' \in \theta_{mut}(\sigma),$ and $\rho_{mut}(q') = \{\top\}\}$.
- $\delta'(c \cup q, i) = \bigcup_{\sigma \in 2^M} \{c' \cup q' : c' \in apply(\delta, \sigma)(c, i), q' \in \delta_{mut}(q, \sigma),$ and $\rho_{mut}(q') = \{\top\}\}$.
- $\rho'(c \cup q) = apply(\rho, \sigma)(c)$.

B Analyzing Vacuous and Loose Satisfaction

In this section we give the details for the analysis of vacuous and loose satisfaction for the different mutations described in Sections 3.1 and 3.3.

B.1 Analyzing vacuous satisfaction

Removing a given set of behaviors We say that a restriction $\langle r_\theta, r_\delta\rangle$ of $\mathcal{S}$ does not affect the satisfaction of $\mathcal{S}$ in $\mathcal{I}$ if $\langle r_\theta, r_\delta\rangle$ retains the receptiveness of $\mathcal{S}$ and $\mathcal{I} \models \mathcal{S}'$, for the circuit $\mathcal{S}'$ obtained from $\mathcal{S}$ by applying $\langle r_\theta, r_\delta\rangle$. Consider a

restriction $\langle r_\theta, r_\delta \rangle$ of $\mathcal{S}$ that does not affect the satisfaction of $\mathcal{S}$ in $\mathcal{I}$. If $t \in r_\theta(i)$, then we can conclude that each of the behaviors of $\mathcal{S}$ that starts with input i and in state t is either not exhibited in the implementation, or is subsumed by another behavior of the specification. Likewise, if $t \in r_\delta(s, i)$, then we can conclude that each of the behaviors of $\mathcal{S}$ that reaches s and continues to t on input i is either not exhibited in the implementation, or is subsumed by another behavior of the specification. While the user can check arbitrary restrictions, some restrictions are of particular interest.

We say that a restriction $\langle r_\theta, r_\delta \rangle$ is *maximal* if $\langle r_\theta, r_\delta \rangle$ does not affect the satisfaction of $\mathcal{S}$ in $\mathcal{I}$ and every restriction $\langle r'_\theta, r'_\delta \rangle$ that strictly contains $\langle r_\theta, r_\delta \rangle$ does affect the satisfaction. Note that there may be several maximal restrictions. Maximal restrictions are interesting as they correspond to tightest specifications that are based on $\mathcal{S}$ and are still satisfied by $\mathcal{I}$.

Some restrictions correspond to physical failures. For example, if a restriction that removes self loops from the circuit (that is, $r_\delta(s, i) = \{s\}$) does not affect the satisfaction of $\mathcal{S}$ in $\mathcal{I}$, we can conclude that the specification allows unbounded delays that are actually fulfilled in the implementation in the present. The connection of restrictions to physical failures is more interesting for mutations that modify or add behaviors, as such mutations are applied to the implementation.

Another class of interesting restrictions are these that contain a path (rather than isolated transitions). Thus, there is a sequence $i_0, i_1, \ldots, i_k$ of input assignments and a sequence $s_0, s_1, \ldots, i_k$ of states, such that $s_0 \in r_\theta(i_0)$ and for all $0 \models j < k$, we have $s_{j+1} \in r_\delta(s_j, i_{j+1})$. Such a restriction that does not affect the satisfaction of $\mathcal{S}$ in $\mathcal{I}$ points to a linear behavior that is allowed by the (abstract) specification and is still either not exhibited by the implementation or subsumed by another behavior of the specification.

Removing behaviors that depend on a signal Assume that the signal x is such that the mutation $\mathcal{S}'$ obtained by removing behaviors that depend on x, as defined in Section 3.1.2, is receptive, and $\mathcal{I} \models \mathcal{S}'$. Then, further abstraction of the specification (namely, one in which x is abstracted) is possible. This hints on a problem: either the specifier does not fully understand the specification, as he expects x to do play a role, or the design does not exhibit behaviors that distinguish between the different values of x and that the specifier expects the design to exhibit.

Restricting a signal to a value Assume that the signal x is such that if $\mathcal{I} \models \mathcal{S}'$ for the mutation $\mathcal{S}'$ obtained by restricting x to 0, as defined in Section 3.1.3. When $x \in O \setminus C$ and $\mathcal{S}'$ is defined over the same set O of output signals as $\mathcal{I}$, we can conclude that no states in which $x = 1$ are reachable in $\mathcal{I}$. We note that such a conclusion may be reached by performing simpler checks than $\mathcal{I} \models \mathcal{S}'$. When $x \in C \setminus O$, it may still be the case that the implementation reaches states with $x = 1$, but these states agree on their label with states of the specifications in which $x = 0$. Thus, behaviors of the implementations in which $x = 1$ exist also in the specification, but there they are exhibited along states in which $x = 0$.

B.2 Analyzing loose satisfaction

Adding a given set of behaviors We say that an extension $\langle r_\theta, r_\delta \rangle$ of $\mathcal{S}$ does not affect the satisfaction of $\mathcal{S}$ in $\mathcal{I}$ if $\mathcal{I}' \models \mathcal{S}$, for the circuit $\mathcal{I}'$ obtained from $\mathcal{I}$ by applying $\langle r_\theta, r_\delta \rangle$. Consider an extension $\langle r_\theta, r_\delta \rangle$ of $\mathcal{S}$ that does not affect the satisfaction of $\mathcal{S}$ in $\mathcal{I}$. If $t \in r_\theta(i)$, then it means that a behaviors that starts with input i and in state t and then continue with an existing behavior of the circuit from state t does not violate the specification. Thus, such a behavior is exhibited in the specification, or is subsumed by another behavior of the implementation.

As in the case of removal of behaviors, of particular interests are extensions that contain a path. Thus, there is a sequence $i_0, i_1, \ldots, i_k$ of input assignments and a sequence $s_0, s_1, \ldots, i_k$ of states, such that $s_0 \in r_\theta(i_0)$ and for all $0 \models j < k$, we have $s_{j+1} \in r_\delta(s_j, i_{j+1})$. Such an extension that does not affect the satisfaction of $\mathcal{S}$ in $\mathcal{I}$ points to a linear behavior that can be added to the implementation without violating the specification.

Another interesting type of extensions are these that add self loops to the circuit. If $\mathcal{I}' \models \mathcal{S}$, we can conclude that the the specification actually allows an unbounded, and possibly infinite stuttering in the implementation.

Adding behaviors that depend on a signal Assume that the signal x is such that $\mathcal{I}' \models \mathcal{S}$, for the mutation $\mathcal{I}'$ of $\mathcal{I}$ obtained by adding behaviors that dualize the behavior of x, as defined in Section 3.3.2, and $\mathcal{I} \models \mathcal{S}'$. Then, the specification does not restrict the behaviors to the particular value of x that the designer has implemented. This hints on a problem: either the specification should be tightened, or the design disables behaviors expected by the designer.