Variations on Safety

Orna Kupferman

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

Abstract. Of special interest in formal verification are *safety* properties, which assert that the system always stays within some allowed region, in which nothing "bad" happens. Equivalently, a property is a safety property if every violation of it occurs after a finite execution of the system. Thus, a computation violates the property if it has a "bad prefix", all whose extensions violate the property. The theoretical properties of safety properties as well as their practical advantages with respect to general properties have been widely studied. The paper surveys several extensions and variations of safety. We start with *bounded* and *checkable* properties – fragments of safety properties that enable an even simpler reasoning. We proceed to a *reactive* setting, where safety properties require the system to stay in a region of states that is both allowed and from which the environment cannot force it out. Finally, we describe a probability-based approach for defining different levels of safety.

1 Introduction

Today's rapid development of complex and safety-critical systems requires reliable verification methods. In formal verification, we verify that a system meets a desired property by checking that a mathematical model of the system meets a formal specification that describes the property. Of special interest are properties asserting that the observed behavior of the system always stays within some allowed region, in which nothing "bad" happens. For example, we may want to assert that every message sent is acknowledged in the next cycle. Such properties of systems are called *safety properties*. Intuitively, a property ψ is a safety property if every violation of ψ occurs after a finite execution of the system. In our example, if in a computation of the system a message is sent without being acknowledged in the next cycle, this occurs after some finite execution of the system. Also, once this violation occurs, there is no way to "fix" the computation.

In order to formally define what safety properties are, we refer to computations of a nonterminating system as infinite words over an alphabet Σ. Consider a language L of infinite words over Σ. A finite word x over Σ is a *bad prefix* for L iff for all infinite words y over Σ, the concatenation $x \cdot y$ of x and y is not in L. Thus, a bad prefix for L is a finite word that cannot be extended to an infinite word in L. A language L is a *safety language* if every word not in L has a finite bad prefix. For example, if $\Sigma = \{0, 1\}$, then $L = \{0^\omega, 1^\omega\}$ is a safety language. Indeed, every word not in L contains either the

sequence 01 or the sequence 10, and a prefix that ends in one of these sequences cannot be extended to a word in L. [1].

The interest in safety started with the quest for natural classes of specifications. The theoretical aspects of safety have been extensively studied [2, 28, 29, 33]. With the growing success and use of formal verification, safety has turned out to be interesting also from a practical point of view [14, 20, 23]. Indeed, the ability to reason about finite prefixes significantly simplifies both enumerative and symbolic algorithms. In the first, safety circumvents the need to reason about complex ω-regular acceptance conditions. For example, methods for temporal synthesis, program repair, or parametric reasoning are much simpler for safety properties [18, 32]. In the second, it circumvents the need to reason about cycles, which is significant in both BDD-based and SAT-based methods [5, 6]. In addition to a rich literature on safety, researchers have studied additional classes, such as liveness and co-safety properties [2, 28].

The paper surveys several extensions and variations of safety. We start with *bounded* and *checkable* properties – fragments of safety properties that enable an even simpler reasoning. We proceed to a *reactive* setting, where safety properties require the system to stay in a region of states that is both allowed and from which the environment cannot force it out. Finally, we describe a probability-based approach for defining different levels of safety. The survey is based on the papers [24], with Moshe Y. Vardi, [21], with Yoad Lustig and Moshe Y. Vardi, [25], with Sigal Weiner, and [10], with Shoham Ben-David.

2 Preliminaries

Safety and Co-Safety Languages. Given an alphabet Σ, a *word over* Σ is a (possibly infinite) sequence $w = \sigma_0 \cdot \sigma_1 \cdots$ of letters in Σ. Consider a language $L \subseteq \Sigma^\omega$ of infinite words. A finite word $x \in \Sigma^*$ is a *bad prefix* for L iff for all $y \in \Sigma^\omega$, we have $x \cdot y \notin L$. Thus, a bad prefix is a finite word that cannot be extended to an infinite word in L. Note that if x is a bad prefix, then all the finite extensions of x are also bad prefixes. A language L is a *safety* language iff every infinite word $w \notin L$ has a finite bad prefix. For a safety language L, we denote by *bad-pref*(L) the set of all bad prefixes for L.

For a language $L \subseteq \Sigma^\omega$, we use *comp*$(L)$ to denote the complement of L; i.e., *comp*$(L) = \Sigma^\omega \setminus L$. A language $L \subseteq \Sigma^\omega$ is a *co-safety* language iff *comp*(L) is a safety language. (The term used in [28] is *guarantee* language.) Equivalently, L is co-safety iff every infinite word $w \in L$ has a *good prefix* $x \in \Sigma^*$: for all $y \in \Sigma^\omega$, we have $x \cdot y \in L$. For a co-safety language L, we denote by *good-pref*(L) the set of good prefixes for L. Note that for a safety language L, we have that *good-pref*$($*comp*$(L)) =$ *bad-pref*(L).

Word Automata. A *nondeterministic Büchi word automaton* (NBW, for short) is $\mathcal{A} = \langle \Sigma, Q, \delta, Q_0, F \rangle$, where Σ is the input alphabet, Q is a finite set of states, $\delta : Q \times \Sigma \to 2^Q$ is a transition function, $Q_0 \subseteq Q$ is a set of initial states, and $F \subseteq Q$ is a set of

[1] The definition of safety we consider here is given in [1, 2], it coincides with the definition of limit closure defined in [12], and is different from the definition in [26], which also refers to the property being closed under stuttering.

accepting states. If $|Q_0| = 1$ and δ is such that for every $q \in Q$ and $\sigma \in \Sigma$, we have that $|\delta(q, \sigma)| \leq 1$, then $\mathcal{A}$ is a *deterministic* Büchi word automaton (DBW, for short).

Given an input word $w = \sigma_0 \cdot \sigma_1 \cdots$ in Σ^ω, a *run* of $\mathcal{A}$ on w is a sequence $r_0, r_1, \ldots$ of states in Q such that $r_0 \in Q_0$ and for every $i \geq 0$, we have $r_{i+1} \in \delta(r_i, \sigma_i)$. For a run r, let $inf(r)$ denote the set of states that r visits infinitely often. That is, $inf(r) = \{q \in Q : r_i = q \text{ for infinitely many } i \geq 0\}$. As Q is finite, it is guaranteed that $inf(r) \neq \emptyset$. The run r is *accepting* iff $inf(r) \cap F \neq \emptyset$. That is, iff there exists a state in F that r visits infinitely often. A run that is not accepting is *rejecting*. When $\alpha = Q$, we say that $\mathcal{A}$ is a *looping* automaton. We use NLW and DLW to denote nondeterministic and deterministic lopping automata. An NBW $\mathcal{A}$ accepts an input word w iff there exists an accepting run of $\mathcal{A}$ on w. The *language* of an NBW $\mathcal{A}$, denoted $\mathcal{L}(\mathcal{A})$, is the set of words that $\mathcal{A}$ accepts. We assume that a given NBW $\mathcal{A}$ has no empty states, except maybe the initial state (that is, at least one word is accepted from each state – otherwise we can remove the state).

Linear Temporal Logic. The logic *LTL* is a linear temporal logic. Formulas of LTL are constructed from a set AP of atomic propositions using the usual Boolean operators and the temporal operators G ("always"), F ("eventually"), X ("next time"), and U ("until"). Formulas of LTL describe computations of systems over AP. For example, the LTL formula $G(req \rightarrow Fack)$ describes computations in which every position in which req holds is eventually followed by a position in which ack holds. Thus, each LTL formula ψ corresponds to a language, denoted $||\psi||$, of words in $(2^{AP})^\omega$ that satisfy it. For the detailed syntax and semantics of LTL, see [30]. The *model-checking problem* for LTL is to determine, given an LTL formula ψ and a system M, whether all the computations of M satisfy ψ.

General methods for LTL model checking are based on translation of LTL formulas to nondeterministic Büchi word automata. By [36], given an LTL formula ψ, one can construct an NBW $\mathcal{A}_\psi$ over the alphabet 2^{AP} that accepts exactly all the computations that satisfy ψ. The size of $\mathcal{A}_\psi$ is, in the worst case, exponential in the length of ψ.

Given a system M and an LTL formula ψ, model checking of M with respect to ψ is reduced to checking the emptiness of the product of M and $\mathcal{A}_{\neg\psi}$ [36]. This check can be performed on-the-fly and symbolically [7, 35], and the complexity of model checking that follows is PSPACE, with a matching lower bound [34].

It is shown in [2, 33, 22] that when ψ is a safety formula, we can assume that all the states in $\mathcal{A}_\psi$ are accepting. Indeed, $\mathcal{A}_\psi$ accepts exactly all words all of whose prefixes have at least one extension accepted by $\mathcal{A}_\psi$, which is what we get if we define all the states of $\mathcal{A}_\psi$ to be accepting. Thus, safety properties can be recognized by NLWs. Since every NLW can be determined to an equivalent DLW by applying the subset construction, all safety formulas can be translated to DLWs.

3 Interesting Fragments

In this section we discuss two interesting fragments of safety properties: *clopen* (a.k.a. bounded) properties, which are useful in bounded model checking, and *checkable* properties, which are useful in real-time monitoring.

3.1 Clopen Properties

Bounded model checking methodologies check the correctness of a system with respect to a given specification by examining computations of a bounded length. Results from set-theoretic topology imply that sets in Σ^ω that are both open and closed (*clopen sets*) are bounded: membership in a clopen set can be determined by examining a bounded number of letters in Σ.

In [24] we studied safety properties from a topological point of view. We showed that clopen sets correspond to properties that are both safety and co-safety, and show that when clopen specifications are given by automata or LTL formulas, we can point to a bound and translate the specification to bounded formalisms such as bounded LTL and cycle-free automata.

Topology Consider a set X and a distance function $d : X \times X \to \mathbb{R}$ between the elements of X. For an element $x \in X$ and $\gamma \geq 0$, let $K(x, \gamma)$ be the set of elements x' such that $d(x, x') \leq \gamma$. Consider a set $S \subseteq X$. An element $x \in S$ is called an *interior element* of S if there is $\gamma > 0$ such that $K(x, \gamma) \subseteq S$. The set S is *open* if all the elements in S are interior. A set S is *closed* if $X \setminus S$ is open. So, a set S is open if every element in S has a nonempty "neighborhood" contained in S, and a set S is closed if every element not in S has a nonempty neighborhood whose intersection with S is empty. A set that is both open and close is called a *clopen* set.

A *Cantor space* consists of $X = D^\omega$, for some finite set D, and d defined by $d(w, w') = \frac{1}{2^n}$, where n is the first position where w and w' differ. Thus, elements of X can be viewed as infinite words over D and two words are close to each other if they have a long common prefix. If $w = w'$, then $d(w, w') = 0$. It is known that clopen sets in Cantor space are *bounded*, where a set S is bounded if it is of the form $W \cdot D^\omega$ for some finite set $W \subseteq D^*$. Hence, clopen sets in our Cantor space correspond exactly to bounded properties: each clopen language $L \subseteq \Sigma^\omega$ has a bound $k \geq 0$ such that membership in L can be determined by the prefixes of length k of words in Σ^ω.

It is not hard to see that a language $L \subseteq \Sigma^\omega$ is co-safety iff L is an open set in our Cantor space [27, 17]. To see this, consider a word w in a co-safety language L, and let x be a good prefix of w. All the words w' with $d(w, w') \leq \frac{1}{2^{|x|}}$ have x as their prefix, so they all belong to L. For the second direction, consider a word w in an open set L, and let $\gamma > 0$ be such that $K(w, \gamma) \subseteq L$. The prefix of w of length $\lfloor \log \frac{1}{\gamma} \rfloor$ is a good prefix for L. It follows that the clopen sets in Σ^ω are exactly these properties that are both safety and co-safety!

Bounding Clopen Properties Our goal in this section is to identify a bound for a clopen property given by an automaton. Consider a clopen language $L \subseteq \Sigma^\omega$. For a finite word $x \in \Sigma^*$, we say that x is *undetermined* with respect to L if there are $y \in \Sigma^\omega$ and $z \in \Sigma^\omega$ such that $x \cdot y \in L$ and $x \cdot z \notin L$. As shown in [24], every word in Σ^ω has only finitely many prefixes that are undetermined with respect to L. It follows that L is *bounded*: there are only finitely many words in Σ^* that are undetermined with respect to L. For an integer k, we say that L is *bounded by* k if all the words $x \in \Sigma^*$ such that $|x| \geq k$ are determined with respect to L. Moreover, since L is bounded, then

a minimal DLW that recognizes L must be cycle free. Indeed, otherwise we can pump a cycle to infinitely many undetermined prefixes. Let $diameter(L)$ be the diameter of the minimal DLW for L.

Lemma 1. *A clopen ω-regular language $L \subseteq \Sigma^\omega$ is bounded by $diameter(L)$.*

Proof: Let $\mathcal{A}$ be the minimal deterministic looping automaton for L. Consider a word $x \in \Sigma^*$ with $|x| \geq diameter(L)$. Since $\mathcal{A}$ is cycle free, its run on x either reaches an accepting sink, in which case x is a good prefix, or it does not reach an accepting sink, in which case, by the definition of $diameter(\mathcal{A})$, we cannot extend x to a word accepted by $\mathcal{A}$, thus x is a bad prefix. □

For a language L, the *in index* of L, denoted $inindex(L)$, is the minimal number of states that an NBW recognizing L has. Similarly, the *out index* of L, denoted $outindex(L)$, is the minimal number of states that an NBW recognizing *comp*(L) has.

Lemma 2. *A clopen ω-regular language $L \subseteq \Sigma^\omega$ is bounded by $inindex(L) \cdot outindex(L)$.*

Proof: Assume by way of contradiction that there is a word $x \in \Sigma^*$ such that $|x| \geq inindex(L) \cdot outindex(L)$ and x is undetermined with respect to L. Thus, there are suffixes y and z such that $x \cdot y \in L$ and $x \cdot z \notin L$. Let $\mathcal{A}_1$ and $\mathcal{A}_2$ be nondeterministic looping automata such that $\mathcal{L}(\mathcal{A}_1) = L$, $\mathcal{L}(\mathcal{A}_2) =$ *comp*(L), and $\mathcal{A}_1$ and $\mathcal{A}_2$ have $inindex(L)$ and $outindex(L)$ states, respectively. Consider two accepting runs r_1 and r_2 of $\mathcal{A}_1$ and $\mathcal{A}_2$ on $x \cdot y$ and $x \cdot z$, respectively. Since $|x| \geq inindex(L) \cdot outindex(L)$, there are two prefixes $x[1, \ldots, i]$ and $x[1, \ldots, j]$ of x such that $i < j$ and both runs repeat their state after these two prefixes; i.e., $r_1(i) = r_1(j)$ and $r_2(i) = r_2(j)$. Consider the word $x' = x[1, \ldots, i] \cdot x[i+1, \ldots, j]^\omega$. Since $\mathcal{A}_1$ is a looping automaton, the run r_1 induces an accepting run r'_1 of $\mathcal{A}_1$ on x'. Formally, for all $l \leq i$ we have $r'_1(l) = r_1(l)$ and for all $l > i$, we have $r'_1(l) = r_1(i + ((l - i) \text{mod} (j - i)))$. Similarly, the run r_2 induces an accepting run of $\mathcal{A}_2$ on x'. It follows that x' is accepted by both $\mathcal{A}_1$ and $\mathcal{A}_2$, contradicting the fact that $\mathcal{L}(\mathcal{A}_2) =$ *comp*$(\mathcal{L}(\mathcal{A}_1))$. □

3.2 Checkable Properties

For an integer $k \geq 1$, a language $L \subseteq \Sigma^\omega$ is *k-checkable* if there is a language $R \subseteq \Sigma^k$ (of "allowed subwords") such that a word w belongs to L iff all the subwords of w of length k belong to R. A property is locally checkable if its language is k-checkable for some k. Locally checkable properties, which are a special case of safety properties, are common in the specification of systems. In particular, one can often bound an eventuality constraint in a property by a fixed time frame, which results in a checkable property.

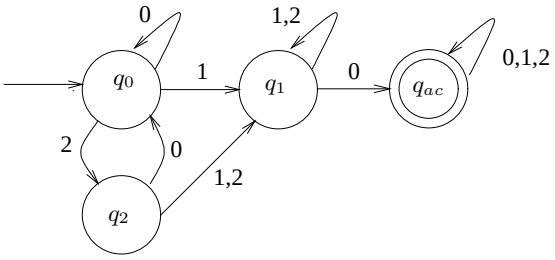
The practical importance of locally checkable properties lies in the low memory demand for their run-time verification. Indeed, k-checkable properties can be verified with a bounded memory – one that has access only to the last k-computation cycles. Run-time verification of a property amounts to executing a monitor together with the system allowing the detection of errors in run time [20, 3, 9]. Run-time monitors for

checkable specifications have low memory demand. Furthermore, in the case of general ω-regular properties, when several properties are checked, we need a monitor for each property, and since the properties are independent of each other, so are the state spaces of the monitors. Thus, the memory demand (as well as the resources needed to maintain the memory) grow linearly with the number of properties monitored. Such a memory demand is a real problem in practice. In contrast, as shown in [21], a monitor for a k-checkable property needs only a record of the last k computation cycles. Furthermore, even if a large number of k-checkable properties are monitored, the monitors can share their memory, resulting in memory demand of $|\Sigma|^k$, which is independant of the number of properties monitored.

As in the case of clopen properties, our goal is to identify a bound for a checkable property given by an automaton. We first need some notations. For a word $w \in \Sigma^\omega$ and $k \geq 0$, we denote by $sub(w, k)$ the set of finite subwords of w of length k, formally, $sub(w, k) = \{y \in \Sigma^* : |y| = k$ and there exist $x \in \Sigma^*$ and $z \in \Sigma^\omega$ such that $w = xyz\}$. A language $L \subseteq \Sigma^\omega$ is *k-checkable* if there exists a finite language $R \subseteq \Sigma^k$ such that $w \in L$ iff all the k-long subwords of w are in R. That is, $L = \{w \in \Sigma^\omega : sub(w, k) \subseteq R\}$. A language $L \subseteq \Sigma^\omega$ is *k-co-checkable* if there exists a finite language $R \subseteq \Sigma^k$ such that $w \in L$ iff there exists a k-long subword of w that is in R. That is, $L = \{w \in \Sigma^\omega : sub(w, k) \cap R \neq \emptyset\}$. A language is *checkable* (*co-checkable*) if it is k-checkable (k-co-checkable, respectively) for some k. We refer to k as the *width* of L. It is easy to to see that all checkable languages are safety, and similarly for co-checkable and co-safety. In particular, L is a checkable language induced by R iff *comp*(L) is co-checkable and induced by *comp*(L).

In order to demonstrate the the subtlety of the width question, consider the following example.

Example 1. Let $\Sigma = \{0, 1, 2\}$. The DBW $\mathcal{A}$ below recognizes the language L of all the words that contain 10, 120 or 220 as subwords. Note that L is the 3-co-checkable language L co-induced by $R = \{010, 110, 210, 100, 101, 102, 120, 220\}$. Indeed, a word w is in L iff $sub(w, 3) \cap R \neq \emptyset$.



At first sight, it seems that the same considerations applied in Lemma 1 can be used in order to prove that the width of a checkable language is bounded by the diameter of the smallest DBW recognizing the language. Indeed, it appears that in an accepting run, the traversal through the minimal good prefix should not contain a cycle. This

impression, however, is misleading, as demonstrated in the DBW $\mathcal{A}$ from Example 1, where a traversal through the subword 120 contains a cycle, and similarly for 010. The diameter of the DBW $\mathcal{A}$ is 3, so it does not constitute a counterexample to the conjecture that the diameter bounds the width, but the problem remains open in [21], and the tightest bound proven there depends on the size of $\mathcal{A}$ and not only on its diameter, and is even not linear. Intuitively, it follows form an upper-bound on the size of a DBW that recognizes minimal bad prefixes of L. Formally, we have the following.

Theorem 1. *If a checkable (or co-checkable) language L is recognized by a DBW with n states, then the width of L is bounded by $O(n^2)$.*

As noted above, the bound in Theorem 1 is not tight and the best known lower bound is only the diameter of a DBW for L. For the nondeterministic setting the bound is tighter:

Theorem 2. *If a checkable language L is recognized by an NBW with n states, then the width of L is bounded by $2^{O(n)}$. Also, There exist an NBW $\mathcal{A}$ with $O(n)$ states such that $L(\mathcal{A})$ is k-checkable but not $(k-1)$-checkable, for $k = (n+1)2^n + 2$.*

4 Safety in a Reactive Setting

Recall that safety is defined with respect to languages over an alphabet Σ. Typically, $\Sigma = 2^{AP}$, where AP is the set of the system's atomic propositions. Thus, the definition and studies of safety treat all the atomic propositions as equal and do not distinguish between input and output signals. As such, they are suited for closed systems – ones that do not maintain an interaction with their environment. In open (also called *reactive*) systems [19, 31], the system interacts with the environment, and a correct system should satisfy the specification with respect to all environments. A good way to think about the open setting is to consider the situation as a game between the system and the environment. The interaction between the players in this game generates a computation, and the goal of the system is that only computations that satisfy the specification will be generated.

Technically, one has to partition the set AP of atomic propositions to a set I of input signals, which the environment controls, and a set O of output signals, which the system controls. An open system is then an *I/O-transducer* – a deterministic automaton over the alphabet 2^I in which each state is labeled by an output in 2^O. Given a sequence of assignments to the input signals (each assignment is a letter in 2^I), the run of the transducer on it induces a sequence of assignments to the output signals (that is, letters in 2^O). Together these sequences form a computation, and the transducer *realizes* a specification ψ if all its computations satisfy ψ [31].

The transition from the closed to the open setting modifies the questions we typically ask about systems. Most notably, the *synthesis* challenge, of generating a system that satisfies the specification, corresponds to the satisfiability problem in the closed setting and to the realizability problem in the open setting. As another example, the equivalence problem between LTL specifications is different in the closed and open settings [16]. That is, two specifications may not be equivalent when compared with

respect to arbitrary systems on $I \cup O$, but be *open equivalent*; that is, equivalent when compared with respect to I/O-transducers. To see this, note for example that a satisfiable yet non-realizable specification is equivalent to false in the open but not in the closed setting.

As mentioned above, the classical definition of safety does not distinguish between input and output signals. The definition can still be applied to open systems, as a special case of closed systems with $\Sigma = 2^{I \cup O}$. In [11], Ehlers and Finkbeiner introduced *reactive safety* – a definition of safety for the setting of open systems. Essentially, reactive safety properties require the system to stay in a region of states that is both allowed and from which the environment cannot force it out. The definition in [11] is by means of sets of trees with directions in 2^I and labels in 2^O. The use of trees naturally locate reactive safety between linear and branching safety. In [25], we suggested an equivalent yet differently presented definition, which explicitly use realizability, and study the theoretical aspects of receive safety and other reactive fragments of specifications. In this section, we review the definition and results from [25].

4.1 Definitions

We model open systems by *transducers*. Let I and O be finite sets of input and output signals, respectively. Given $x = i_0 \cdot i_1 \cdot i_2 \cdots \in (2^I)^\omega$ and $y = o_0 \cdot o_1 \cdot o_2 \cdots \in (2^O)^\omega$, we denote their composition by $x \oplus y = (i_0, o_0) \cdot (i_1, o_1) \cdot (i_2, o_2) \cdots \in (2^{I \cup O})^\omega$. An I/O-transducer is a tuple $\mathcal{T} = \langle I, O, S, s_0, \eta, L \rangle$, where S is a set of states, $s_0 \in S$ is an initial state, $\eta : S \times 2^I \to S$ is a transition function, and $L : S \to 2^O$ is a labeling function. The *run* of $\mathcal{T}$ on a (finite or infinite) input sequence $x = i_0 \cdot i_1 \cdot i_2 \cdots$, with $i_j \in 2^I$, is the sequence $s_0, s_1, s_2, \ldots$ of states such that $s_{j+1} = \eta(s_j, i_{j+1})$ for all $j \geq 0$. The *computation* of $\mathcal{T}$ on x is then $x \oplus y$, for $y = L(s_0) \cdot L(s_1) \cdot L(s_2) \cdots$ Note that $\mathcal{T}$ is responsive and deterministic (that is, it suggests exactly one successor state for each input letter), and thus $\mathcal{T}$ has a single run, generating a single computation, on each input sequence. We extend η to finite words over 2^I in the expected way. In particular, $\eta(s_0, x)$, for $x \in (2^I)^*$ is the $|x|$-th state in the run on x. A transducer $\mathcal{T}$ induces a *strategy* $f : (2^I)^* \to 2^O$ such that for all $x \in (2^I)^*$, we have that $f(x) = L(\eta(s_0, x))$. Given an LTL formula ψ over $I \cup O$, we say that ψ is *I/O-realizable* if there is a finite-state I/O-transducer $\mathcal{T}$ such that all the computations of $\mathcal{T}$ satisfy ψ [31]. We then say that $\mathcal{T}$ realizes ψ. When it is clear from the context, we refer to I/O-realizability as *realizability*, or talk about realizability of languages over the alphabet $2^{I \cup O}$.

Since the realizability problem corresponds to deciding a game between the system and the environment, and the game is determined [15], realizability is determined too, in the sense that either there is an I/O-transducer that realizes ψ (that is, the system wins) or there is an O/I-transducer that realizes $\neg\psi$ (that is, the environment wins). Note that in an O/I-transducer the system and the environment "switch roles" and the system is the one that provides the inputs to the transducer. A technical detail is that in order for the setting of O/I-realizability to be dual to the one in I/O-realizability we need, in addition to switching the roles and negating the specification, to switch the player that moves first and consider transducers in which the environment initiates the interaction and moves first. Since we are not going to delve into constructions, we ignore this point, which is easy to handle.

Let I and O be sets of input and output signals, respectively. Consider a language $L \subseteq (2^{I \cup O})^\omega$. For a finite word $u \in (2^{I \cup O})^*$, let $L^u = \{s : u \cdot s \in L\}$ be the set of all infinite words s such that $u \cdot s \in L$. Thus, if L describes a set of allowed computations, then L^u describes the set of allowed suffixes of computations starting with u.

We say that a finite word $u \in (2^{I \cup O})^*$ is a *system bad prefix* for L iff L^u is not realizable. Thus, a system bad prefix is a finite word u such that after traversing u, the system does not have a strategy to ensure that the interaction with the environment would generate a computation in L. We use $sbp(L)$ to denote the set of system bad prefixes for L. Note that by determinacy of games, whenever L^u is not realizable by the system, then its complement is realizable by the environment. Thus, once a bad prefix has been generated, the environment has a strategy to ensure that the entire generated behavior is not in L.

A language $L \subseteq (2^{I \cup O})^\omega$ is a *reactive safety language* if every word not in L has a system bad prefix. Below are two examples, demonstrating that a reactive safety language need not be safe. Note that the other direction does hold: Let L be a safe language. Consider a word $w \notin L$ and a bad prefix $u \in (2^{I \cup O})^*$ of w. Since u is a bad prefix, the set L^u is empty, and is therefore unrealizable, so u is also a system bad prefix. Thus, every word not in L has a system bad prefix, implying that L is reactively safe.

Example 2. Let $I = \{\mathit{fix}\}$, $O = \{\mathit{err}\}$, $\psi = G(\mathit{err} \rightarrow F\mathit{fix})$, and $L = \|\psi\|$. Note that ψ is realizable using the system strategy "never err". Also, L is clearly not safe, as every prefix can be extended to one that satisfies ψ. On the other hand, L is reactively safe. Indeed, every word not in L must have a prefix u that ends with $\{\mathit{err}\}$. Since $L^u = \|F\mathit{fix}\|$, which is not realizable, we have that u is a system bad prefix and L is reactively safe.

Example 3. Let $I = \{\mathit{fix}\}$, $O = \{\mathit{err}\}$, $\psi = G\neg\mathit{err} \vee FG\mathit{fix}$, and $L = \|\psi\|$. Note that ψ is realizable using the system strategy "never err". Also, L is clearly not safe. We show L is reactively safe. Consider a word $w \notin L$. Since w does not satisfy $G\neg\mathit{err}$, there must be a prefix u of w such that u contains a position satisfying err. Since words with prefix u do not satisfy $G\neg\mathit{err}$, we have that $L^u = \|FG\mathit{fix}\|$, which is not realizable. Thus, u is a system bad prefix and L is reactively safe.

4.2 Properties of Reactive Safety

In the closed settings, the set *bad-pref*(L) is closed under finite extensions for all languages $L \subseteq \Sigma^\omega$. That is, for every finite word $u \in$ *bad-pref*(L) and finite extension $v \in \Sigma^*$, we have that $u \cdot v \in$ *bad-pref*(L). This is not the case in the reactive setting:

Theorem 3. *System bad prefixes are not closed under finite extension.*

Proof: Let $I = \{\mathit{fix}\}$, $O = \{\mathit{err}\}$, and $\psi = G(\mathit{err} \rightarrow X\mathit{fix}) \wedge FG\neg\mathit{err}$. Thus, ψ states that every error the system makes is fixed by the environment in the following step, and that there is a finite number of errors. Let $L = \|\psi\|$. Clearly, ψ is realizable, as the strategy "never err" is a winning strategy for the system. Also, L is reactively safe, as a word $w \notin L$ must have a prefix u that ends in a position satisfying err, and u is

a system bad prefix. We show that $sbp(L)$ is not closed under finite extensions. To see this, consider the word $w = (\{err, \mathit{fix}\} \cdot \{\mathit{fix}\})^\omega$. That is, the system makes an error on every odd position, and the environment always fixes errors. Since there are infinitely many errors in w, it does not satisfy ψ. The prefix $u = \{err, \mathit{fix}\}$ of w is a system bad prefix. Indeed, an environment strategy that starts with $\neg \mathit{fix}$ is a winning strategy. On the other hand, u's extension $v = \{err, \mathit{fix}\} \cdot \{\mathit{fix}\}$ is not a system bad prefix. Indeed, L^v is realizable using the winning system strategy "never err". □

Recall that reasoning about safety properties is easier than reasoning about general properties. In particular, rather than working with automata on infinite words, one can model check safety properties using automata (on finite words) for bad prefixes. The question is whether and how we can take advantage of reactive safety when the specification is not safe (but is reactively safe). In [11], the authors answered this question to the positive and described a transition from reactively safe to safe formulas. The translation is by means of nodes in the tree in which a violation starts. The translation from [25] we are going to describe here uses realizability explicitly, which we find simpler.

For a language $L \subseteq (2^{I \cup O})^\omega$, we define $close(L) = L \cap \{w \ : \ w$ has no system bad prefix for $L\}$. Equivalently, $close(L) = L \setminus \{w : w$ has a system bad prefix for $L\}$. Intuitively, we obtain $close(L)$ by defining all the finite extensions of $sbp(L)$ as bad prefixes. It is thus easy to see that $sbp(L) \subseteq$ ***bad-pref***$(close(L))$.

As an example, consider again the specification $\psi = G(err \rightarrow X\mathit{fix}) \wedge FG\neg err$, with $I = \{\mathit{fix}\}$, $O = \{err\}$. An infinite word contains a system bad prefix for ψ iff it has a position that satisfies err. Accordingly, $close(\psi) = G\neg err$. As another example, let us add to O the signal ack, and let $\psi = G(err \rightarrow X(\mathit{fix} \wedge F\mathit{ack}))$, with $I = \{\mathit{fix}\}$, $O = \{err, ack\}$. Again, ψ is reactively safe and an infinite word contains a system bad prefix for ψ iff it has a position that satisfies err. Accordingly, $close(\psi) = G\neg err$.

Our definition of $close(L)$ is sound, in the following sense:

Theorem 4. *A language $L \subseteq (2^{I \cup O})^\omega$ is reactively safe iff $close(L)$ is safe.*

While L and $close(L)$ are not equivalent, they are *open equivalent* [16]. Formally, we have the following.

Theorem 5. *For every language $L \subseteq (2^{I \cup O})^\omega$ and I/O-transducer $\mathcal{T}$, we have that $\mathcal{T}$ realizes L iff $\mathcal{T}$ realizes $close(L)$.*

It is shown in [11] that given an LTL formula ψ, it is possible to construct a deterministic looping word automaton for $close(\psi)$ with doubly-exponential number of states. In fact, as suggested in [23], it is then possible to generate also a deterministic automaton for the bad prefixes of $close(\psi)$. Note that when L is not realizable, we have that $\epsilon \in sbp(L)$, implying that $close(L) = \emptyset$. It follows that we cannot expect to construct small automata for $close(L)$, even nondeterministic ones, as the realizability problem for LTL can be reduced to easy questions about them.

Theorem 5 implies that a reactive safety language L is open equivalent to a safe language, namely $close(L)$. Conversely, open equivalence to a safe language implies reactive safety. This follows from the fact that if L and L' are open-equivalent languages, then a prefix x is a minimal system bad prefix in L iff x is a minimal system bad prefix in L'. We can thus conclude with the following.

Theorem 6. *A language L is reactively safe iff L is open equivalent to a safe language.*

In the setting of open systems, dualization of specifications is more involved, as one has not only to complement the language but to also dualizes the roles of the system and the environment. Accordingly, we actually have four fragments of languages that are induced by dualization of the reactive safety definition. We define them by means of bad and good prefixes.

Consider a language $L \subseteq (2^{I \cup O})^\omega$ and a prefix $u \in (2^{I \cup O})^*$. We say that:

- u is a *system bad prefix* if L^u is not I/O-realizable.
- u is a *system good prefix* if L^u is I/O-realizable.
- u is an *environment bad prefix* if L^u is not O/I-realizable.
- u is an *environment good prefix* if L^u is O/I-realizable.

Now, a language $L \subseteq (2^{I \cup O})^\omega$ is a *system (environment) safety language* if every word not in L has a system (environment, respectively) bad prefix. The language L is a *system (environment) co-safety language* if every word in L has a system (environment, respectively) good prefix. System safety and environment co-safety dualize each other: For every language $L \subseteq (2^{I \cup O})^\omega$, we have that L is system safe iff *comp*(L) is environment co-safe.

Since each language L^u is either I/O-realizable or not I/O-realizable, and the same for O/I-realizability, all finite words are determined, in the following sense.

Theorem 7. *Consider a language $L \subseteq (2^{I \cup O})^\omega$. All finite words in $(2^{I \cup O})^*$ are determined with respect to L. That is, every prefix is either system good or system bad, and either environment good or environment bad, with respect to L.*

Note that while every prefix is determined, a word may have both system bad and system good prefixes, and similarly for the environment, which is not the case in the setting of closed systems. For example, recall the language $L = \|G(err \rightarrow Xfix) \wedge FG\neg err\|$, for $I = \{fix\}$ and $O = \{err\}$. As noted above, the word $(\{err, fix\} \cdot \{fix\})^\omega$ has both a system bad prefix $\{err, fix\}$, and a system good prefix $\{err, fix\} \cdot \{fix\}$.

In Section 3.1 we showed that in the closed setting, the intersection of safe and co-safe properties induces the fragment of *bounded* properties. It is shown in [25] that boundedness in the open setting is more involved, as a computation may have both infinitely many good and infinitely many bad prefixes. It is still possible, however, to define reactive bounded properties and use their appealing practical advantages.

5 A Spectrum Between Safety and Co-Safety

Safety is a binary notion. A property may or may not satisfy the definition of safety. In this section we describe a probability-based approach for defining different levels of safety. The origin of the definition is a study of *vacuity* in model checking [4, 23]. Vacuity detection is a method for finding errors in the model-checking process when the specification is found to hold in the model. Most vacuity algorithms are based on checking the effect of applying mutations on the specification. It has been recognized

that vacuity results differ in their significance. While in many cases vacuity results are valued as highly informative, there are also cases in which the results are viewed as meaningless by users. In [10], we suggested a method for an automatic ranking of vacuity results according to their level of importance. Our method is based on the *probability* of the mutated specification to hold in a random computation. For example, two natural mutations of the specification $G(req \rightarrow Fready)$ are $G(\neg req)$, obtained by mutating the subformula $ready$ to **false**, and $GFready$, obtained by mutating the subformula req to **true**. It is agreed that vacuity information about satisfying the first mutation is more alarming than information about satisfying the second. The framework in [10] formally explains this, as the probability of $G(\neg req)$ to hold in a random computation is 0, whereas the probability of $GFready$ is 1. In this section we suggest to use probability also for defining levels of safety.

5.1 The Probabilistic Setting

Given a set S of elements, a *probability distribution* on S is a function $\mu : S \rightarrow [0,1]$ such that $\Sigma_{s \in S}\ \mu(s) = 1$. Consider an alphabet Σ. A random word over Σ is a word in which for all indices i, the i-th letter is drown uniformly at random. In particular, when $\Sigma = 2^{AP}$, then a random computation π is such that for each atomic proposition q and for each position in π, the probability of q to hold in the position is $\frac{1}{2}$. An equivalent definition of this probabilistic model is by means of the probabilistic labeled structure $\mathcal{U}_\Sigma$, which generates computations in a uniform distribution. Formally, $\mathcal{U}_\Sigma$ is a clique with $|\Sigma|$ states in which a state $\sigma \in \Sigma$ is labeled σ, is initial with probability $\frac{1}{|\Sigma|}$, and the probability to move from a state σ to a state σ' is $\frac{1}{|\Sigma|}$.

We define the probability of a language $\mathcal{L} \subseteq \Sigma^\omega$, denoted $Pr(\mathcal{L})$, as the probability of the event $\{\pi : \pi \text{ is a path in } \mathcal{U}_\Sigma \text{ that is labeled by a word in } \mathcal{L}\}$. Accordingly, for an LTL formula φ, we define $Pr(\varphi)$ as the probability of the event $\{\pi : \pi \text{ is a path in } \mathcal{U}_{2^{AP}} \text{ that satisfies } \varphi\}$. For example, the probabilities of Xp, Gp, and Fp are $\frac{1}{2}, 0$, and 1, respectively. Using $\mathcal{U}_\Sigma$ we can reduce the problem of finding $Pr(\varphi)$ to φ's model checking. Results on probabilistic LTL model checking [8] then imply that the problem of finding the probability of LTL formulas is PSPACE-complete.

First-order logic respects a $0/1$-law: the probability of a formula to be satisfied in a random model is either 0 or 1 [13]. It is easy to see that a $0/1$-law does not hold for LTL. For example, for an atomic proposition p, we have that $Pr(p) = \frac{1}{2}$. Back to our safety story, it is not hard to see that $Pr(G\xi)$, for a formula ξ with $Pr(\xi) \neq 1$, is 0. Dually, $Pr(F\xi)$, for a formula ξ with $Pr(\xi) \neq 0$ is 1. Can we relate this to the fact that Gp is a safety property whereas Fp is a co-safety property? Or perhaps it has to do with Fp being a liveness property?[2] This is not clear, as, for example, the probability of clopen formulas depends on finitely many events and can vary between 0 to 1. As another example, consider the two possible semantics of the Until temporal operator. For the standard, strong, Until, which is not a safe, we have $Pr(pUq) = \frac{2}{3}$. By changing the semantics of the Until to a weak one, we get the safety formula pWq, with $pWq = pUq \vee Gp$. Still, $Pr(pWq) = Pr(pUq)$. Thus, the standard probabilistic setting does not suggest a clear relation between probability and different levels of safety.

[2] A language L is a liveness language if $L = \Sigma^* \cdot L$ [1].

We argue that we can still use the probabilistic approach in order to measure safety. The definition of $Pr(\varphi)$ in [10] assumes that the probability of an atomic proposition to hold in each position is $\frac{1}{2}$. This corresponds to computations in an infinite-state system and is the standard approach taken in studies of $0/1$-laws. Alternatively, one can also study the probability of formulas to hold in computations of finite-state systems. Formally, for an integer $l \geq 1$, let $Pr_l(\varphi)$ denote the probability that φ holds in a random cycle of length l. Here too, the probability of each atomic proposition to hold in a state is $\frac{1}{2}$, yet we have only l states to fix an assignment to. So, for example, while $Pr(Gp) = 0$, we have that $Pr_1(Gp) = \frac{1}{2}$, $Pr_2(Gp) = \frac{1}{4}$, and in general $Pr_j(Gp) = \frac{1}{2^j}$. Indeed, an l-cycle satisfies Gp iff all its states satisfy p.

There are several interesting issues in the finite-state approach. First, it may seem obvious that the bigger l is, the closer $Pr_l(\varphi)$ gets to $Pr(\varphi)$. This is, however, not so simple. For example, issues like cycles in φ can cause $Pr_l(\varphi)$ to be non-monotonic. For example, when φ requires p to hold in exactly all even positions, then $Pr_1(\varphi) = 0, Pr_2(\varphi) = \frac{1}{4}, Pr_3(\varphi) = 0, Pr_4(\varphi) = \frac{1}{16}$, and so on.

Assume now that we have cleaned the cycle-based issue (for example by restricting attention to formulas without Xs, or by restricting attention to cycles of "the right" length). Can we characterize safety properties by means of the asymptotic behavior of $Pr_l(\varphi)$? Can we define different levels of safety according to the rate the probability decreases or increases? For example, clearly $Pr_l(Gp)$ tends to 0 as l increases, whereas $Pr_l(Fp)$ tends to 1. Also, now, for a given l, we have that $Pr_l(pWq) > Pr_l(pUq)$. In addition, for a clopen property φ, we have that $Pr_l(\varphi)$ stablizes once l is bigger than the bound of φ. Still, the picture is not clean. For example, FGp is a liveness formula, but $Pr_l(FGp)$ decreases as l increases. Finding a characterization of properties that is based on the analysis of Pr_l is an interesting question, and our initial research suggests a connection between the level of safety of φ and the behavior of $Pr_l(\varphi)$.

References

1. B. Alpern and F.B. Schneider. Defining liveness. *IPL*, 21:181–185, 1985.
2. B. Alpern and F.B. Schneider. Recognizing safety and liveness. *Distributed computing*, 2:117–126, 1987.
3. H. Barringer, A. Goldberg, K. Havelund, and K. Sen. Rule-based runtime verification. In *Proc. 5th VMCAI*, LNCS 2937, pages 44–57. Springer, 2004.
4. I. Beer, S. Ben-David, C. Eisner, and Y. Rodeh. Efficient detection of vacuity in ACTL formulas. In *Proc. 9th CAV*, LNCS 1254, pages 279–290, 1997.
5. A. Biere, A. Cimatti, E.M. Clarke, and Y. Zhu. Symbolic model checking without BDDs. In *Proc. 5th TACAS*, LNCS 1579. Springer, 1999.
6. R. Bloem, H.N. Gabow, and F. Somenzi. An algorithm for strongly connected component analysis in $n \log n$ symbolic steps. In *Proc. 3rd FMCAD*, LNCS 1954, pages 37–54. Springer, 2000.
7. C. Courcoubetis, M.Y. Vardi, P. Wolper, and M. Yannakakis. Memory efficient algorithms for the verification of temporal properties. *FMSD*, 1:275–288, 1992.
8. C. Courcoubetis and M. Yannakakis. The complexity of probabilistic verification. *J. ACM*, 42:857–907, 1995.
9. M. d'Amorim and G. Rosu. Efficient monitoring of omega-languages. In *Proc. 17th CAV*, LNCS 3576. Springer, 2005.

10. S. Ben David and O. Kupferman. A framework for ranking vacuity results. In *11th ATVA*, LNCS 8172, pages 148–162. Springer, 2013.
11. R. Ehlers and B. Finkbeiner. Reactive safety. In *Proc. 2nd GANDALF*, volume 54 of *Electronic Proceedings in TCS*, pages 178–191, 2011.
12. E.A. Emerson. Alternative semantics for temporal logics. *TCS*, 26:121–130, 1983.
13. R. Fagin. Probabilities in finite models. *Journal of Symb. Logic*, 41(1):50–5, 1976.
14. E. Filiot, N. Jin, and J.-F. Raskin. An antichain algorithm for LTL realizability. In *Proc. 21st CAV*, LNCS 5643, pages 263–277, 2009.
15. D. Gale and F. M. Stewart. Infinite games of perfect information. *Ann. Math. Studies*, 28:245–266, 1953.
16. K. Greimel, R. Bloem, B. Jobstmann, and M. Vardi. Open implication. In *Proc. 35th ICALP*, LNCS 5126, pages 361–372. Springer, 2008.
17. H.P. Gumm. Another glance at the Alpern-Schneider characterization of safety and liveness in concurrent executions. *IPL*, 47:291–294, 1993.
18. D. Harel, G. Katz, A. Marron, and G. Weiss. Non-intrusive repair of reactive programs. In *ICECCS*, pages 3–12, 2012.
19. D. Harel and A. Pnueli. On the development of reactive systems. In *Logics and Models of Concurrent Systems*, *NATO ASI*, vol. F-13, pages 477–498. Springer, 1985.
20. K. Havelund and G. Rosu. Synthesizing monitors for safety properties. In *Proc. 8th TACAS*, LNCS 2280, pages 342–356. Springer, 2002.
21. O. Kupferman, Y. Lustig, and M.Y. Vardi. On locally checkable properties. In *Proc. 13th LPAR*, LNCS 4246, pages 302–316. Springer, 2006.
22. O. Kupferman and M.Y. Vardi. Model checking of safety properties. In *Proc. 11th CAV*, LNCS 1633, pages 172–183. Springer, 1999.
23. O. Kupferman and M.Y. Vardi. Model checking of safety properties. *FMSD*, 19(3):291–314, 2001.
24. O. Kupferman and M.Y. Vardi. On bounded specifications. In *Proc. 8th LPAR*, LNCS 2250, pages 24–38. Springer, 2001.
25. O. Kupferman and S. Weiner. Environment-friendly safety. In *8th HVC*, LNCS 7857, pages 227–242. Springer, 2012.
26. L. Lamport. Logical foundation. In *Distributed systems - methods and tools for specification*, LNCS 190. Springer, 1985.
27. Z. Manna and A. Pnueli. The anchored version of the temporal framework. In *Linear time, branching time, and partial order in logics and models for concurrency*, LNCS 345, pages 201–284. Springer, 1989.
28. Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems: Specification*. Springer, 1992.
29. Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems: Safety*. Springer, 1995.
30. A. Pnueli. The temporal semantics of concurrent programs. *TCS*, 13:45–60, 1981.
31. A. Pnueli and R. Rosner. On the synthesis of a reactive module. In *Proc. 16th POPL*, pages 179–190, 1989.
32. A. Pnueli and E. Shahar. Liveness and acceleration in parameterized verification. In *Proc. 12th CAV*, LNCS 1855, pages 328–343. Springer, 2000.
33. A.P. Sistla. Safety, liveness and fairness in temporal logic. *Formal Aspects of Computing*, 6:495–511, 1994.
34. A.P. Sistla and E.M. Clarke. The complexity of propositional linear temporal logic. *Journal of the ACM*, 32:733–749, 1985.
35. H.J. Touati, R.K. Brayton, and R. Kurshan. Testing language containment for ω-automata using BDD's. *I& C*, 118(1):101–109, 1995.
36. M.Y. Vardi and P. Wolper. Reasoning about infinite computations. *I& C*, 115(1):1–37, 1994.