Polytechnic
UNIVERSITY
Brooklyn · Long Island · Westchester

Classifying Properties: An Alternative to the Safety-Liveness Classification

Gleb Naumovich

Lori A. Clarke

Department of Computer and Information Science

Technical Report
TR-CIS-2000-03
04/14/2000

Classifying Properties: An Alternative to the Safety-Liveness Classification

Gleb Naumovich
Polytechnic University, Brooklyn
Department of Computer and Information Science
Brooklyn, NY 11201
(718) 260-3554
gleb@poly.edu

Lori A. Clarke
Computer Science Department
University of Massachusetts
Amherst, Massachusetts 01003
(413) 545-2013
clarke@cs.umass.edu

Abstract

Traditionally, verification properties have been classified into *safety* and *liveness* properties. While this taxonomy has an attractive simplicity and is useful for identifying the appropriate analysis algorithm to use for checking a property, determining whether a property is safety, liveness, or neither can require significant mathematical insight on the part of the analyst. In this paper, we present an alternative property taxonomy. We argue that this taxonomy is a more natural classification of the kinds of questions that analysts want to ask. Moreover, most classes in our taxonomy have a known, direct mapping to the safety-liveness classification, and thus the appropriate analysis algorithm can be automatically determined.

1. Introduction

A number of finite state verification approaches are being developed. Some approaches are designed to check fixed, general properties of software systems, such as freedom from deadlock. Other approaches offer the flexibility of specifying and checking application-specific properties. Traditionally, properties have been classified into *safety* and *liveness* properties. In practice, it is often important to know whether a given property is a safety or liveness property, because several finite state verification approaches use different analysis algorithms for checking safety and liveness properties [6, 7, 11, 13, 18]. Determining whether a property is safety, liveness, or neither, however, can require significant mathematical insight on the part of the analyst. In this paper, we present an alternative property taxonomy. We argue that this taxonomy is a more natural classification of the kinds of questions that analysts want to ask (i.e., express as properties). Moreover, most classes in our taxonomy have a known, direct mapping to the safety-liveness classification, and thus the appropriate analysis algorithm within a verification approach of choice can be automatically determined.

Intuitively, a *safety* property specifies that "bad things" do not happen on all executions of a system and a *liveness* property specifies that "good things" eventually happen on all executions of a system [14]. Unfortunately, these intuitive definitions are often difficult to use for distinguishing between safety and liveness properties. For example, a property specifying that a communication socket must be created on all executions of a system is a liveness property, which agrees with the informal definition where the "good thing" is the creation of a communication socket. However, a similar property stating that a communication socket must be created before any disk access happens is a safety property, even though we can still view creation of a communication socket as a "good thing".

Because informal definitions of safety and liveness are unreliable, one has to use the precise definitions, given in section 2.2. Using these definitions requires constructing proofs, which are often non-trivial. Carrying out such proofs is human-intensive and error-prone. In this paper, we propose a new property taxonomy for which it is easy to assign a given property to one of several property classes. Furthermore, many of these property classes include only safety or only liveness properties, so if the given property belongs to one of these classes, no proofs are required for determining whether it is a safety or a liveness property.

Our property classification is described in terms of sequences of recognizable *events* in the system under analysis. We distinguish between property specifications that describe finite, infinite, or both kinds of sequences of events. We also distinguish between property specifications that must be checked on only finite, only infinite, or all system executions. At present, it is usually assumed that a property has to be checked on all system executions. In some cases, however, an analyst wants to specify sequences of events that hold on only finite or only infinite executions. For example, it does not make sense to check the property "all files must be closed by the time a system terminates" on infinite executions of a system or to check the property "an unbounded number of files must not be open at the same time" on finite executions. In cases where the analyst is interested in checking a property on only finite executions, the property has to be modified in such a way that it always holds on all infinite executions and vice versa. In contrast, with our proposed classification scheme the analyst explicitily indicates if the property is specifying finite, infinite or both finite and infinite sequences as well as indicates if the property is to be checked on finite, infinite, or all executions of the system.

In the next section, we give some background on reasoning about software systems and give formal definitions of safety and liveness properties. In Section 3 we offer a critique of the safety-liveness taxonomy. Section 4 describes our proposed property classification. In Section 5 we describe the relationship between our classification and the safety-liveness taxonomy by indicating whether a class in our taxonomy contains only safety, only liveness, or both kinds of properties. Finally, Section 6 summarizes.

2. Background

In this section, we briefly introduce the two alternative ways of representing a software system's behaviors, state-based and event-based and then introduce the traditional classification of event-based properties into safety and liveness.

2.1. Event-based and State-based Properties

The two popular ways of modeling systems are *state-based* and *event-based*. With the former, the model encodes all possible states the system might be in during execution. For a concurrent system, a system state may include the program counters for each of the threads of execution and the values for all program variables. Properties for systems with state-based models usually can be represented simply as sequences of state predicates that must hold along sequences of transitions between system states. We call such properties *state-based*. With the event-based approach to modeling systems, the model encodes all event sequences that can be observed during executions of the system. The events used in these sequences represent some actions of the system, where an action may represent some arbitrary level of granularity. For example, both a variable assignment and a function call could be events. Actions that are not of interest to the analyst usually are not assigned corresponding events. Properties for systems with event-based models are given in the form of sets of sequences of events that characterize executions of the model of the system. We call such properties *event-based*.

Theoretically, translations between state-based and event-based representations of systems and properties are not difficult. For example, if a system is specified as a set of sequences of its states, any such sequence can be translated into a sequence of events, where each event represents a transition from one state to another. In the rest of this paper we only deal with event-based models of systems and property specifications.

We assume that event-based properties use a subset of the events that could occur along an execution of the system under analysis. Throughout this paper, we use the term *event sequence* or just *sequence* to refer to any sequence of events and *execution* to refer to a sequence of events observed on an execution of the software system under analysis.[1]

The alphabet of property P is denoted $\Sigma(P)$ and represents the union of all events that could possibly occur on a sequence accepted by P: A projection of an event sequence s on an alphabet Σ is the event sequence s' obtained from s by removing all events not present in Σ. We use the notation $s|_\Sigma$ to denote projection of sequence s on alphabet Σ. For example, if a property P only has events $\mathtt{a}$ and $\mathtt{b}$ in its alphabet, then for a given system execution sequence $(\mathtt{a}, \mathtt{c}, \mathtt{d}, \mathtt{b}, \mathtt{c}, \mathtt{a})$ we find its projection on $\Sigma(P)$: $(\mathtt{a}, \mathtt{c}, \mathtt{d}, \mathtt{b}, \mathtt{c}, \mathtt{a})|_{\Sigma(P)} = (\mathtt{a}, \mathtt{b}, \mathtt{a})$ and then check if the resulting sequence $(\mathtt{a}, \mathtt{b}, \mathtt{a})$ would be accepted by P. We write $s \in P$ to represent the fact that sequence s is in the set of event sequences of P. We say that a sequence s' is *accepted* by property P, denoting this $s' \vdash P$, if the projection of s' on the alphabet of P is in the set of sequences represented by P. Thus, for the example above, $(\mathtt{a}, \mathtt{b}, \mathtt{a}) \in P$ and $(\mathtt{a}, \mathtt{c}, \mathtt{d}, \mathtt{b}, \mathtt{c}, \mathtt{a}) \vdash P$.

Let E be the union of the alphabets for a set of properties for a particular software system. We use E^* to denote the set of all finite sequences of events from E

[1] "Execution of the software system under analysis" is actually a trace or path through the event-based model of the system. Each event sequence that could be observed during execution of the system is represented by a trace through the model of that system. For brevity, we refer to this simply as an execution.

and E^ω to denote the set of all infinite sequences of events from E. We assume that the empty sequence $\lambda \in E^*$. For convenience, we introduce a function $\mathit{prefixes} : E^\omega \rightarrow 2^{E^*}$ that, given an infinite sequence σ, returns all finite prefixes of σ, including the empty sequence λ.

To use a uniform notation, many finite state verification approaches replace each finite execution v of a system with an infinite execution σ by adding infinitely many instances of an empty event τ to the end of v: $\sigma = v\tau\tau....$ For convenience, we assume that no non-empty events can follow event τ, which means that τ is used only for representing system termination.

A property can be represented as a set of event sequences that must hold on all system executions. For brevity, we use $\mathcal{P}_E$ to denote the set of all possible properties for set E. A property P holds for a system if and only if all system executions satisfy P.

2.2. The Safety-Liveness Property Taxonomy

While the informal definition of safety and liveness properties, given in the introduction, has intuitive appeal, generally it is not precise enough to be used for determining whether a given property is a safety or a liveness property.

A concise definition of safety and liveness properties based on topology says that a property is a safety property if and only if it is closed and a liveness property if and only if it is dense [3]. Equivalently, a safety property is the one that is finitely refutable and a liveness property is the one that is never finitely refutable [1,3].

In this section, we describe the safety-liveness taxonomy proposed by Alpern and Schneider [3][2]. The formal definition of a safety property [3] is

$$\begin{aligned} &P \text{ is a safety property iff} \\ &\quad \forall \sigma \in E^\omega, \sigma \not\vdash P \Rightarrow \\ &\quad (\exists v \in \mathit{prefixes}(\sigma) : (\forall \sigma' \in E^\omega, v\sigma' \not\vdash P)) \end{aligned} \qquad (1)$$

This definition means that P is a safety property if every infinite sequence of events that does not satisfy this property contains a finite prefix such that no infinite sequence obtained by adding an infinite suffix to this finite prefix satisfies this property.

The formal definition of a liveness property [3] is

$$\begin{aligned} &\text{Set } P \text{ is a liveness property iff} \\ &\quad \forall v \in E^*, (\exists \sigma \in E^\omega : v\sigma \vdash P) \end{aligned} \qquad (2)$$

This definition means

from the native property representation to the Büchi automaton representation, to perform the split into two Büchi automata, one for a safety property and another for a liveness property, and to translate these two automata back to the native property representation. In addition, this approach does not solve the problem of specifying whether the property must hold on finite, infinite, or all executions of the system.

3. Critique of the Safety-Liveness Taxonomy

The safety-liveness taxonomy has an attractive simplicity, providing two fundamental classes of properties, so that any property can be represented as a combination of two properties, one from each class. While it is an elegant and theoretically useful classification, in our opinion it has several important problems.

The first problem is terminology. Intuitively, the term *safety property* implies that if an execution of a system does not satisfy such property, it represents an unsafe behavior of a system. While this may be true, safety properties are not the only kind of property that specifies what it means for a software system to operate safely. aCxecirplf twos app errvtcetuf ter tem events τ to the end of all finite executions.

Our first classification criterion is based on what kinds of execution sequences are represented by the property. There are three obvious cases:

- **property P contains only finite event sequences**: $\forall s \in P \Rightarrow s \in E^*$,
- **property P contains only infinite event sequences**: $\forall s \in P \Rightarrow s \in E^\omega$, and
- **property P contains both finite and infinite event sequences**: $\exists s_1, s_2 \in P \Rightarrow s_1 \in E^* \wedge s_2 \in E^\omega$.

Our second classification criterion is based on whether the property refers to only finite, only infinite, or all executions of the system. We use S to denote the set of all execution sequences in the system. We recognize three cases:

- **The property is for finite execution sequences only**: the property holds if $\forall s \in S, s \in E^* \Rightarrow s \vdash P$.
- **The property is for infinite execution sequences only**: the property holds if $\forall s \in S, s \in E^\omega \Rightarrow s \vdash P$.
- **The property is for both finite and infinite execution sequences**: the property holds if $\forall s \in S \Rightarrow s \vdash P$.

Intersecting the two criteria, we obtain nine property classes. We refer to each property class by a tuple (A, B), where A refers to the first criterion and B refers to the second criterion. Thus, $A \in \{inf, fin, both\}$ and $B \in \{inf, fin, all\}$. Of these classes, class (inf, fin) is empty, since it does not make sense to specify a finite behavior with an infinite event sequence. For the same reason, class (inf, all) is equivalent to class (inf, inf), in the sense that any property from (inf, all) holds for a system if and only if all executions in the system are infinite. Also, class $(both, fin)$ is equivalent to class (fin, fin) in the sense that for any property P_1 from class $(both, fin)$ there exists property P_2 from class (fin, fin) (obtained from P_1 by discarding all infinite sequences) such that a system execution satisfies P_1 if and only if it satisfies P_2. Thus, we exclude classes (inf, fin), (inf, all), and $(both, fin)$ from our classification as redundant. In the rest of this paper we refer to the remaining six classes (fin, fin), (fin, inf), (fin, all), (inf, inf), $(both, inf)$, and $(both, all)$ as our property classification.

It is obvious that this property specification is complete in the sense that any property specifying a behavior that must hold on all executions of a system belongs to one of the three classes (fin, all), (inf, all), and $(both, all)$. The additional granularity provided by our second classification category is for added convenience of specifying properties. The reader might wonder how this trichotomy compares to the traditional safety, liveness, or both trichotomy used in the safety-liveness classification scheme. In the next section, we will explicitly describe this relationship.

In the following, we briefly describe each of the six categories. For each category, we give an example property that deals with opening and closing of files in a program. For this example, the events of interest to the properties correspond to calls to `open` and `close` file primitives.

(fin, fin)

A property P from class (fin, fin) specifies a set of event sequences of finite length and requires that all finite executions of the system are present in this set. This means that we can construct property P' that refers to all system executions by including in the set of event sequences of P' all event sequences of P and in addition all infinite event sequences: $P' = P \cup E^\omega$. Property P' holds on all executions of a system if and only if property P holds on all finite executions of this system. An example from this category is a property specifying that if a file is open at any point during the program execution, it is closed by the time the program terminates.

(fin, inf)

This is an interesting case, because in order for an infinite execution sequence σ to satisfy property P containing only finite event sequences, the projection of σ on the alphabet of P must be finite. In other words, σ must have a representation $v\sigma'$, where v is a finite prefix of σ and $\Sigma(\sigma') \cap \Sigma(P) = \emptyset$. For example, if the property specifies that on infinite executions of the system, events `a` and `b` alternate (but not infinitely often), then an infinite system execution `abababababab...` does not satisfy this property, because its projection on the alphabet of the property `{a, b}` is infinite. On the other hand, an infinite execution `ababccccc...` does satisfy this property, because `ababccccc...`$|_{\{a,\ b\}}$ = `abab`, which is a finite sequence on which each event `a` is followed by a `b`. Since properties from set (fin, inf) are not concerned with finite executions of the system, for each such property P we can construct property $P' \in (fin, all)$ by including in the set of event sequences of P' all event sequences of P and in addition all finite event sequences: $P' = P \cup E^*$. Property P' holds on all execution of a system if and only if property P holds on all infinite executions of this system. An example from this category is a property specifying that a file is never open for an infinite period of time.

(*fin*, *all*)

A property P from this class can also be represented as a conjunction of two properties P_1, P_2, where $P_1 \in$ (*fin*, *fin*) and $P_2 \in$ (*fin*, *inf*), where both P_1 and P_2 contain the same event sequences as P: $P_1 = P_2 = P$. If v is a finite execution of the system, then it satisfies P if it satisfies P_1. If σ is an infinite execution, then it satisfies P if it satisfies P_2. An example from this category is a property specifying that a file is opened and then closed exactly once on any execution of the program.

(*inf*, *inf*)

A property P from class (*inf*, *inf*) specifies a set of event sequences of infinite length and requires that all infinite executions of the system are present in this set. We can construct property $P' \in$ (*both*, *all*) by including in the set of event sequences of P' all event sequences of P and in addition all finite event sequences: $P' = P \cup E^*$. Property P' holds on all executions of a system if and only if property P holds on all infinite executions of this system. An example from this category is a property specifying that a file is opened infinitely many times on all infinite executions of the system.

(*both*, *inf*)

Any property P from this class can also be represented as a disjunction of two properties P_1 and P_2, where $P_1 \in$ (*fin*, *inf*) and $P_2 \in$ (*inf*, *inf*). In addition, we can construct $P' \in$ (*both*, *all*) by including in the set of event sequences of P' all event sequences of P and in addition all finite event sequences: $P' = P \cup E^*$. Property P' holds on all execution of a system if and only if property P holds on all infinite executions of this system. An example from this category is a property specifying that on all infinite executions, a file has to be opened and then closed at least once, but could be opened and then closed an infinite number of times.

(*both*, *all*)

This is the most general of all classes. Any property from this class can be represented as a disjunction of two properties P_1 and P_2, where $P_1 \in$ (*fin*, *all*) and $P_2 \in$ (*inf*, *inf*), by setting $P_1 = P \cap E^*$ and $P_2 = P \cap E^\omega$. An example from this category is a property specifying that on all executions, the operations of opening and closing a file strictly alternate, whether they occur a finite or an infinite number of times.

4.1. QRE Property Specification Language

As an example of a property specification language that supports our property classification scheme, we describe an extension we are developing for the *Quantified Regular Expressions (QRE)* language [10, 19]. The QRE language uses regular and ω-regular expressions and represents a convenient approach for specifying event sequencing properties.

A QRE specification consists of three parts: *alphabet*, *regular expressions*, and *modifier*. The alphabet simply lists all events of interest to this property. Regular expressions describe sequences of events of interest to this property. Modifier specify if the event sequences described by the regular expressions must hold on *all* system executions or on *no* system executions. In our extension, *modifier* is replaced by *modifiers*, which in addition to the quantification, also indicate whether this property must be checked for finite, for infinite, or for both kinds of system executions.

At present, the alphabet is specified simply by listing all events of interest to the property (in future, parameterization and aliases will be supported). The alphabet must contain all events explicitly used in the regular expressions but may also contain additional events. We explain the need for such additional events below when discussing the regular expressions part of the QRE specifications.

If multiple regular expressions are present in a QRE, the property is represented by a union of the sets of event sequences that each of these regular expressions specifies. Regular expressions are specified using an assortment of traditional syntactic features for supporting regular languages. Because of space constraints, we do not describe all these features here. Importantly, one of the features used in this language is *complement*. For example, "any event in the alphabet, except events `a` and `b`" may be represented in a QRE as `[-a,b]`. Thus, an event `c` from the property alphabet may not appear in the regular expression explicitly, although it is represented implicitly.

Regular expressions in the extended QRE notation may be ω-regular expressions, to indicate that a certain pattern of events repeats infinitely often. We use the symbol `@` to represent such an infinite repetition. For example, `a@` specifies an infinite sequence `aaaa...` In some cases, it is convenient to specify a certain pattern of events that may or may not repeat infinitely often. For example, an analyst may want to specify that events `a` and `b` alternate, without restricting whether this repetition is finite or infinite.. We use the symbol `#` to specify that the regular expression to which this symbol refers repeats either 0 or more times or infinitely: <expr>#

```
{open_F, close_F}

(open_F; close_F); (open_F; close_F)#

all infinite
```

Figure 1: An example extended QRE specification

is equivalent to `<expr>*|<expr>@`, where `<expr>` is an arbitrary regular expression and `|` is a logical "or" operator.

Finally, modifiers in the extended QRE language are preceded by one of two kinds of keywords. The first specifies the quantification by indicating whether the event sequences described by the regular expressions must hold on all executions (keyword `all`) or no executions (keyword `no`) of the system. The second specifies the execution by indicating by if only finite executions (keyword `finite`), only infinite executions (keyword `infinite`), or all possible executions (keyword `both`) have to be compared to the event sequences described by the regular expressions. Figure 1 shows a property specifying that that file `F` has to be open and then closed at least once, but could be open and then closed an infinite number of times, with `open` and `close` operations strictly alternating.

A property specified in this extended QRE language can be automatically classified into one of the categories of our classification scheme. Whether the property should be checked on finite, infinite, or all executions in the system is specified explicitly and information about whether only finite, only infinite, or both kinds of event sequences are present in the property specification can be derived from the regular expressions. If none of the regular expressions contain symbols `@` or `#` then the property represents only finite event sequences. If some regular expressions contain symbols `@` or `#`, then the structure of the regular expressions can be analyzed to determine whether or not they may encode finite sequences in addition to infinite sequences. For example, it is easy to see that the property in Figure 1 belongs to class $(both, inf)$. The use of symbol `#` in the regular expression part indicates (in this case) that the property contains both infinite and finite sequences. The modifier `infinite` indicates that the property refers to only infinite executions.

5. Relationship between the Proposed Taxonomy and the Safety-Liveness Taxonomy

In this section for each of the six classes from our classification we describe the part of the safety-liveness universe that it describes. For convenience, we denote the set of all safety properties $\mathcal{S}$ and the set of all liveness properties $\mathcal{L}$. The definitions of safety and liveness in Equations (1) and (2) assume that all execution sequences in the system are infinite (with all finite executions extended by an infinite number of empty events τ). To be able to use these definitions, we define a mapping *InfProp* that, given a property P specified in our classification (e.g., where some event sequences may be finite), returns a property P' that deals with infinite event sequences. P' is equivalent to P in the following sense:

$$\begin{gathered}\forall \sigma \in E^{\omega} : \sigma \in P \Rightarrow \sigma \in P' \\ \forall v \in E^{*} : \sigma \in P \Rightarrow v\tau\tau ... \in P' \\ P \in (fin, fin) \Rightarrow (E^{\omega} \setminus \{\sigma | \tau \in \sigma\} \subseteq P' \\ P \in ((fin, inf) \cup (inf, inf) \cup (both, inf)) \Rightarrow E^{*} \subseteq P'\end{gathered} \tag{3}$$

This means that for any property from our class (fin, fin), the corresponding property in the safety-liveness classification will accept any infinite sequence of events (without added τ events) and for any property in our classification scheme that specifies an infinite execution, the corresponding property in the safety-liveness classification scheme will accept all finite execution sequences.

Given a class C from our classification, for simplicity we use notation like $C \subset \mathcal{S}$ to show the relationship of this class with the safety-liveness taxonomy. In reality, it is the relationship of class $C' = \{P' | \exists P \in C : P' = InfProp(P)\}$ that is considered, since classes C and C' are equivalent in the sense described by Equation (3).

$(fin, fin) \subset \mathcal{S}$

First we prove $(fin, fin) \subseteq \mathcal{S}$. Take any $P \in (fin, fin)$ and let $P' = InfProp(P)$. The only event sequences σ that may not satisfy P' are finite (see Equation (3)) and thus contain the empty event τ. According to the definition of safety in Equation (1), we can pick a finite sequence v to be the prefix of σ that ends with τ and so no infinite suffix σ can make the sequence $v\sigma'$ into a sequence that does not contain τ. $v\sigma'$ also does not satisfy P' since it contains prefix v that caused σ not to satisfy P'. Thus, P is a safety property.

Now we prove $\mathcal{S} \not\subseteq (fin, fin)$. To do that, it is sufficient to show that there is a safety property that is not in (fin, fin). For example, any safety property that does not accept at least one infinite sequence is such a property.

$(fin, inf) \subset \mathcal{L}$

First, we prove $(fin, inf) \subseteq \mathcal{L}$. Take any $P \in (fin, inf)$ and let $P' = InfProp(P)$. Take any $v \in E^{*}$ and let $\sigma = \tau ...$. Then $v\sigma \in P'$, because $v\sigma$ is a representation of a finite sequence, and any finite sequence satisfies a property from (*fin*, *inf*). By the definition in Equation (2), P' is a liveness property.

To prove that $\mathcal{L} \nsubseteq$ (*fin*, *inf*), it is sufficient to show that there is a liveness property that is not in (*fin*, *inf*). For example, a liveness property requiring that on all executions, event a happens infinitely often is not in (*fin*, *inf*), because such a property contains infinite event sequences.

$\exists P_1, P_2, P_3 \in$ (***fin***, ***all***) $: P_1 \in \mathcal{S}, P_2 \in \mathcal{L}, P_3 \notin (\mathcal{S} \cup \mathcal{L})$

An example of P_1 is a property that accepts all sequences except those containing event a. $\forall \sigma \in E^\omega$: $\sigma \notin \mathit{InfProp}(P_1)$ means that σ contains event a. We can write σ in the form $\sigma = v\sigma'$, where v is a sequence ending with a. This is the v from the definition of safety in Equation (1).

An example of P_2 is a property that accepts all sequences except those that do not contain a. Take any $v \in E^*$. If v contains a, then infinite sequence σ from the definition of liveness in Equation (2) can be any infinite sequence. If v does not contain a, then σ can be any infinite sequence that contains a.

An example of P_3 is a property that accepts all sequences that contain exactly one event a. $P_3 \notin \mathcal{S}$ because we can pick σ to be any infinite sequence that does not contain a (and thus does not satisfy P_3). For any finite prefix v of this sequence σ we can take σ' to be any infinite sequence that contains one event a. Then $v\sigma'$ satisfies P_3 and so P_3 is not a safety property by the definition in Equation (1). P_3 is not a liveness property either, because we can pick v from the definition in Equation (2) to be a finite sequence that contains two events a. No infinite sequence σ exists such that $v\sigma \in P_3$.

(***inf***, ***inf***) $\subset \mathcal{L}$

First we prove (*inf*, *inf*) $\subseteq \mathcal{L}$. Take any $P \in$ (*inf*, *inf*) and let $P' = \mathit{InfProp}(P)$. Take any $v \in E^*$. Let σ be any infinite sequence that starts with τ. Then $v\sigma \in P'$ because all finite sequences satisfy P'. By the definition in Equation (2), P' is a liveness property.

To prove $\mathcal{L} \nsubseteq$ (*inf*, *inf*), let P be the property that specifies that an infinite number of events a must happen. No finite sequence can satisfy this liveness property, while any finite execution satisfies any property from (*inf*, *inf*).

(***both***, ***inf***) $\subset \mathcal{L}$

The proof is identical to the proof for (*inf*, *inf*).

$\exists P_1, P_2, P_3 \in$ (***both***, ***all***) $: P_1 \in \mathcal{S}, P_2 \in \mathcal{L}, P_3 \notin (\mathcal{S} \cup \mathcal{L})$

The proofs are identical to those for (*fin*, *all*).

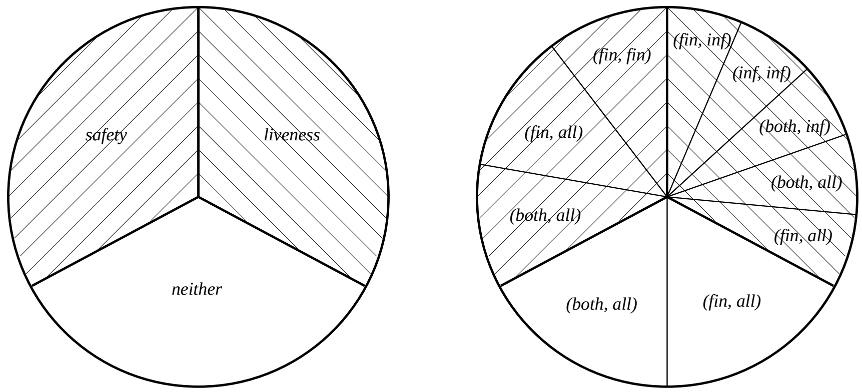
Figure 2 represents the correspondence between our property classification and the safety-liveness taxonomy visually.

6. Conclusion

As shown in Section 5, class (*fin*, *fin*) contains only safety properties and classes (*fin*, *inf*), (*inf*, *inf*), and (*both*, *inf*) contain only liveness properties. This means that if a given property falls into one of these four classes, it is immediately clear whether it is a safety or a liveness property. Classes (*fin*, *all*) and (*both*, *all*) contain safety properties, liveness properties, and also properties that are neither safety nor liveness. In general, any property from class (*fin*, *all*) can be represented as a conjunction of two properties, one from class (*fin*, *fin*), and another from class (*fin*, *inf*). For example, a property specifying that event a happens exactly once on all system executions (which is neither a safety nor liveness property), can be represented as a conjunction of two properties, one checking that a happens exactly once on finite executions and another checking that a happens exactly once on infinite executions. The first of these two properties belongs to class (*fin*, *fin*) and the second belongs to class (*fin*, *inf*).

The case with class (*both*, *all*) is not as simple. Whether or not such a property can be decomposed successfully depends on whether we can decompose the representation of event sequences in the property into finite sequences and infinite sequences. If we can, then the property P is represented as the conjunction of two properties, P_1 from class (*fin*, *fin*), and P_2 from class (*both*, *inf*). (Thus, P_1 is a safety property and P_2 is a liveness property.) For example, a property specifying that on all executions of a system events a and b alternate (but not specifying whether a finite or infinite number of such events must be observed) is in class (*both*, *all*). It can trivially be decomposed into a property P_1 specifying that on all finite executions events a and b alternate and a property P_2 specifying that on all infinite executions events a and b alternate and either a finite or an infinite number of such events is observed. We believe that in practice, most properties from class (*both*, *all*) are decomposable in a similar way.

Our classification scheme complements the property specification patterns work [9]. Property specification patterns map naturally occurring sequences of events, such as "event a must follow event b, but only after event c happens" to formal specifications in a variety of property specification formalisms. Each pattern can be categorized as containing only finite, only infinite, or



(a) The safety-liveness taxonomy

(b) Our classification, superimposed on the safety-liveness taxonomy

Figure 2: The correspondence between our property classification and the safety-liveness taxonomy

both kinds of event sequences, based on its formal representation. Specification patterns could benefit, however, from an additional modifier that indicates whether they should be checked on finite, infinite, or all executions of the system[3]

To summarize, we have described a new property classification based on two simple characteristics of properties. One characteristic is whether the sequences of events used in the property specification is finite, infinite, or both; and the other is whether the property specifies behaviors that must (or must not) hold only on finite, only on infinite, or on all executions of the system. The proposed classification has a number of advantages over the safety-liveness taxonomy. First, it is relatively natural. In fact, we are extending the FLAVERS finite state verification system to accept an extended property specification notation that supports our classification. Second, deciding which of the six classes in our classification a given property belongs is trivially derived from the specification of the property. Finally, and most importantly, four out of six classes in our classification contain properties that are either only safety or only liveness properties, so there is no need for proofs to determine which analysis algorithm to apply. Another class contains properties that can easily be decomposed into two properties, one a safety and another a liveness property. For the final class, the existence of a "nice" decomposition is not guaranteed, but likely.

[3]Note that although the *global absence* pattern [9] "event τ does not happen" can be used to represent infinite executions, it cannot be used in conjunction with patterns representing the actual property that has to be checked on only infinite executions, because at present the rules of composing several patterns are very restricted.

References

[1] M. Abadi and L. Lamport. Composing specifications. *ACM Transactions on Programming Languages and Systems*, 15(1):73–132, Jan. 1993.

[2] S. Aggarwal, C. Courcoubetis, and P. Wolper. Adding liveness properties to coupled finite-state machines. *ACM Transactions of Programming Languages and Systems*, 12(2):303–339, Apr. 1990.

[3] B. Alpern and F. B. Schneider. Defining liveness. *Information Processing Letters*, 21(4):181–185, Oct. 1985.

[4] B. Alpern and F. B. Schneider. Recognizing safety and liveness. *Distributed Computing*, 2:117–126, 1987.

[5] B. Alpern and F. B. Schneider. Verifying temporal properties without temporal logic. *ACM Transactions of Programming Languages and Systems*, 11(1):147–167, Jan. 1989.

[6] S. C. Cheung, D. Giannakopoulou, and J. Kramer. Verification of liveness properties using compositional reachability analysis. In *Proceedings of the 6th European Software Engineering Conference and 5th ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 227–243, Sept. 1997.

[7] S. C. Cheung and J. Kramer. Checking safety properties using compositional reachability analysis. *ACM Transactions on Software Engineering and Methodology*, 8(1):49–78, Jan. 1999.

[8] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions of Programming Languages and Systems*, 8(2):244–263, Apr. 1986.

[9] M. B. Dwyer, G. S. Avrunin, and J. C. Corbett. Patterns in property specifications for finite-state verification. In *Proceedings of the 21st International Conference on Software Engineering*, pages 411–421, May 1999.

[10] M. B. Dwyer and L. A. Clarke. Data flow analysis for verifying properties of concurrent programs. In *Proceedings of the 2nd ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 62–75, Dec. 1994.

[11] M. B. Dwyer and L. A. Clarke. Flow analysis for verifying specifications of concurrent and distributed software. Technical Report 1999-52, University of Massachusetts, Amherst, Aug. 1999. `ftp://ftp.cs.umass.edu/pub/techrept/techreport/1999/UM-CS-1999-052.ps`.

[12] P. Godefroid and G. J. Holzmann. On the verification of temporal properties. In *Proceedings of the 13th International Conference on Protocol Specification, Testing, and Verification, INWG/IFIP*, pages 109–124, May 1993.

[13] G. J. Holzmann. The model checking SPIN. *IEEE Transactions on Software Engineering*, 23(5):279–295, May 1997.

[14] L. Lamport. Proving the correctness of multiprocess programs. *IEEE Transactions on Software Engineering*, SE-3(2):125–143, 1977.

[15] Leslie Lamport. What good is temporal logic? In *Proceedings of the IFIP Congress on Information Processing*, pages 657–667, 1983.

[16] Z. Manna and A. Pnueli. *Verification of Concurrent Programs: The Temporal Framework*, pages 141–154. Academic Press, 1981.

[17] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems*. Springer, New York, 1992.

[18] G. Naumovich and L. A. Clarke. Extending FLAVERS to check properties on infinite executions of concurrent software systems. Technical Report 2000-02, Polytechnic University, New York, 2000. `http://cis.poly.edu/tr/tr-cis-2000-02.pdf`.

[19] K. M. Olender and L. J. Osterweil. Cecil: A sequencing constraint language for automatic static analysis generation. *IEEE Transactions on Software Engineering*, 16(3):268–280, Mar. 1990.

[20] M. Y. Vardi. Verification of concurrent programs: The automata-theoretic framework. In *Proceedings of the 2nd Annual Symposium on Logic in Computer Science*, pages 167–176, June 1987.

[21] P. Wolper. Temporal logic can be more expressive. *Information and Control*, 56(1/2):72–99, Jan./Feb. 1983.