Runtime Verification

Grigore Roşu

University of Illinois at Urbana-Champaign

Contents

Topics to cover:

- Safety properties and their monitoring. How many safety properties are there? Can they all be monitored? Complexity of monitoring in different formalism: LTL, RE, ERE, CFG, SRS. Both dynamic properties, like above, and static properties (e.g., complex heap patterns).
- Event/Trace observation. How to observe the execution of a program? Instrumentation vs. runtime environment.
- Monitor synthesis. Generating optimal monitors for several formalisms: LTL, FT/PT-LTL, RE, ERE, CFG, SRS, PT-CaRet, Allen TL.
- Parametric property monitoring. How to deal with multiple instances of monitors?
- Predictive runtime analysis. Vector clock vs SMT-based techniques.
- Static analysis to improve runtime verification. Improve runtime overhead by not instrumenting what is unnecessary. Improve prediction capability by looking beyond the trace.
- Semantics-based Runtime Verification. Defining a formal language semantics. Using a semantics to do symbolic execution and runtime verify properties. Ultimate goal: verify programs by exhaustive runtime verification.

Chapter 1

Introduction

Begin of intro stuff for chapter on safety

From SACS: ***Abstract sacs:*** *This paper addresses the problem of runtime verification from a foundational perspective, answering questions like "Is there a consensus among the various definitions of a safety property?" (Answer: Yes), "How many safety properties exist?" (Answer: As many as real numbers), "How difficult is the problem of monitoring a safety property?" (Answer: Arbitrarily complex), "Is there any formalism that can express all safety properties?" (Answer: No), etc. Various definitions of safety properties as sets of execution traces have been proposed in the literature, some over finite traces, others over infinite traces, yet others over both finite and infinite traces. By employing cardinality arguments and a novel notion of* persistence*, this paper first establishes the existence of bijective correspondences between the various notions of safety property. It then shows that safety properties can be characterized as "always past" properties. Finally, it proposes a general notion of* monitor*, which allows to show that safety properties correspond precisely to the monitorable properties, and then to establish that monitoring a safety property is arbitrarily hard.*

> **From safety:** ***Abstract safety:*** *Various definitions of safety properties as sets of execution traces have been introduced in the literature, some over finite traces, others over infinite traces, yet others over both finite and infinite traces. By employing cardinality arguments, this paper first shows that these notions of safety are ultimately equivalent, by showing each of them to have the cardinal of the continuum. It is then shown that all safety properties can be characterized as "always past" properties, and then that the problem of monitoring a safety property can be arbitrarily hard. Finally, two decidable specification formalisms for safety properties are discussed, namely extended regular expressions and past time LTL. It is shown that monitoring the former requires non-elementary space. An optimal monitor synthesis algorithm is given for the latter; the generated monitors run in space linear with the number of temporal operators and in time linear with the size of the formula.*

A *safety property* is a behavioral property which, once violated, cannot be satisfied anymore. For example, a property "always $x > 0$" is violated when $x \leq 0$ is observed for the first time; this safety property remains violated even though eventually $x > 0$ might hold. That means that one can identify each safety property with a set of "bad" finite execution traces, with the intuition that once one of those is reached the safety property is violated.

There are several apparently different ways to formalize safety. Perhaps the most immediate one is to complement the "bad traces" above and thus to define a safety property as a prefix-closed property over finite traces (containing the "good traces") – by "property" in this paper we mean a set of finite or infinite traces. Inspired by Lamport [36], Alpern and Schneider [4] define safety properties over infinite traces as ones with the property that if an infinite trace is unacceptable then there must be some finite prefix of it which is already unacceptable, in the sense that there is no acceptable infinite completion of it. Is there any relationship between these two definitions of safety? We show rather indirectly that there is, by showing that their corresponding sets of safety properties have the cardinal c of the continuum (i.e., the cardinal of $\mathbb{R}$, the set of real numbers), so there exists some bijective mapping between the two. Unfortunately, the existence of such a bijection is

as little informative as the existence of a bijection between the real numbers and the irrational numbers. To capture the relationship between finite- and infinite-trace safety properties in a meaningful way, we introduce a subset of finite-trace safety properties, called *persistent*, and then construct an explicit bijection between that subset and the infinite-trace safety properties. Interestingly, over finite traces there are as many safety properties as unrestricted properties (finite-traces are enumerable and $\mathcal{P}(\mathbb{N})$ is in bijection with $\mathbb{R}$), while over infinite traces there are c safety properties versus 2^c unrestricted properties (infinite traces are in bijection with $\mathbb{R}$).

It is also common to define safety properties as properties over both finite and infinite traces, the intuition for the finite traces being that of unfinished computations. For example, Lamport [37] extends the notion of infinite-trace safety properties to properties over both finite and infinite traces, while Schneider et al. [50, 19] give an alternative definition of safety over finite and infinite traces, called "execution monitoring". One immediate technical advantage of allowing both finite and infinite traces is that one can define prefix-closed properties. We indirectly show that prefix-closeness is not a sufficient condition to define safety properties when infinite traces are also allowed, by showing that there are 2^c prefix-closed properties versus, as expected, "only" c safety properties.

Another common way to specify safety properties is as "always past" properties, that is, as properties containing only words whose finite prefixes satisfy a given property. If P is a property on finite prefixes, then we write $\Box P$ for the "always P" safety property containing the words with prefixes in P. We show that specifying safety properties as "always past" properties is fully justified by showing that, for each of the three types of traces (finite, infinite, and both), the "always past" properties are precisely the safety properties as defined above. It is common to specify P using some logical formalism, for example past time linear temporal logic (past LTL) [39]; for example, one can specify "a before b" in past LTL as the formula $b \rightarrow \diamond a$.

The problem of monitoring safety properties is also investigated in this paper. Since there are as many safety properties as real numbers, it is not unexpected that some of them can be very hard to monitor. We show that the problem of monitoring a safety property is arbitrarily hard, by showing that it reduces to deciding membership of natural

numbers to a set of natural numbers. In particular, we can associate a safety property to any degree in the arithmetic hierarchy as well as to any complexity class in the decidable universe, whose monitoring is as hard as that degree or complexity class.

From SACS: *This paper makes three novel contributions, two technical and another pedagogical. On the technical side, it first introduces the notion of a* persistent *safety property, which appears to be the right finite-trace correspondent of an infinite-trace safety property, and uses it to show the cardinal equivalence of the various notions of safety property encountered in the literature. Also on the technical side, it rigorously defines the problem of monitoring a safety property, and it shows that it can be arbitrarily hard. On the pedagogical side, this paper offers the first comprehensive study and uniform presentation of safety properties and of their monitoring.*

From safety: *In practice not all ($c = |\mathbb{R}|$) safety properties are meaningful, but only those ($\aleph_0 = |\mathbb{N}|$) which are specifiable using formal specification languages or logics of interest. We also investigate the problem of monitoring safety properties expressed using two common formalisms, namely regular expressions extended with complement, also called extended regular expressions (ERE), and LTL. It is known that both formalisms allow polynomial finite-trace membership checking algorithms [27, 45] if one has random access to the trace, but that both require exponential space if the trace can only be analyzed online [44, 33]. It is also known that LTL can indeed be monitored in exponential space [13] and so is claimed*[1] *for EREs in [44]. We show that the claim in [44] is, unfortunately, wrong, by showing that ERE monitoring requires non-elementary space. To do so, we propose for any $n \in \mathbb{N}$ a safety property P_n whose monitoring requires space non-elementary in n, as well as an ERE of size $O(n^3)$. Since the known monitoring algorithms for LTL in its full generality are asymptotically optimal, what is left to do is to consider important fragments of LTL. We focus on the "always past" fragment and give a monitor synthesis algorithm that takes formulae φ and generate monitors for them that need $O(k)$ total space and $O(|\varphi|)$ time to process each event, where k is the number of past operators in φ. This improves over the best known algorithm that needs space $O(|\varphi|$ (and same time).*

End of intro stuff for chapter on safety

Chapter 2

Background, Preliminaries, Notations

Add some structure to this chapter

We let $\mathbb{N}$ denote the set of natural numbers including 0 but excluding the infinity symbol ∞ and let $\mathbb{N}_\infty$ denote the set $\mathbb{N} \cup \{\infty\}$. We also let $\mathbb{Q}$ denote the set of rational numbers and $\mathbb{R}$ the set of real numbers; as for natural numbers, the "∞" subscript can also be added to $\mathbb{Q}$ and $\mathbb{R}$ for the corresponding extensions of these sets. $\mathbb{Q}^+$ and $\mathbb{R}^+$ denote the sets of strictly positive (0 not included) rational and real numbers, respectively.

We fix a set Σ of elements called *events* or *states*. We call words in $\Sigma^\star$ *finite traces* and those in Σ^ω *infinite traces*. If $u \in \Sigma^\star \cup \Sigma^\omega$ then u_i is the i-th state or event that appears in u. We call *finite-trace properties* sets $P \subseteq \Sigma^*$ of finite traces, *infinite-trace properties* sets $P \subseteq \Sigma^\omega$ of infinite traces, and just *properties* sets $P \subseteq \Sigma^* \cup \Sigma^\omega$ of finite or infinite traces. If the finite or infinite aspect of traces is understood from context, then we may call any of the types or properties above just *properties*. We may write $P(w)$ for a property P and a (finite or infinite) trace w whenever $w \in P$. Traces and properties are more commonly called *words* and *languages*, respectively, in the literature; we prefer to call them traces and properties to better reflect the intuition that our target application is monitoring and system observance, not formal languages. We take, however, the liberty to also call them words and languages whenever that terminology seems more appropriate.

In some cases states can be simply identified with their names, or labels, and specifications of properties on traces may just refer to those labels. For

example, the regular expression $(s_1 \cdot s_2)^\star$ specifies all those finite traces starting with state s_1 and in which states s_1 and s_2 alternate. In other cases, one can think of states as sets of atomic predicates, that is, predicates that hold in those states: if s is a state and a is an atomic predicate, then we say that $a(s)$ is true iff a "holds" in s; thus, if all it matters with respect to states is which predicates hold and which do not hold in each state, then states can be faithfully identified with sets of predicates. We prefer to stay loose with respect to what "holds" means, because, depending on the context, it can mean anything. In conventional software situations, atomic predicates can be: boolean expressions over variables of the program, their satisfaction being decided by evaluating them in the current state of the program; or whether a function is being called or returned from; or whether a particular variable is being written to; or whether a particular lock is being held by a particular thread; and so on. In the presence of atomic predicates, specifications of properties on traces typically only refer to the atomic predicates. For example, the property "always a before b", that is, those traces containing no state in which b holds that is not preceded by some state in which a holds (for example, a can stand for "authentication" and b for "resource access"), can be expressed in LTL as the formula $\Box(b \rightarrow \diamond a)$.

Let us recall some basic notions and notations from formal languages, temporarily using the consecrated terminology of "words" and "languages" instead of traces and properties. For an alphabet Σ, let $\mathcal{L}_\Sigma$ be the set of languages over Σ, i.e., the powerset $\mathcal{P}(\Sigma^\star)$. By abuse of language and notation, let $\emptyset$ be the empty language $\{\}$ and ϵ the language containing only the empty word, $\{\epsilon\}$. If $L_1, L_2 \in \mathcal{L}_\Sigma$ then $L_1 \cdot L_2$ is the language $\{\alpha_1\alpha_2 \mid \alpha_1 \in L_1 \text{ and } \alpha_2 \in L_2\}$. Note that $L \cdot \emptyset = \emptyset \cdot L = \emptyset$ and $L \cdot \epsilon = \epsilon \cdot L = L$. If $L \in \mathcal{L}_\Sigma$ then $L^\star$ is $\{\alpha_1\alpha_2 \cdots \alpha_n \mid n \geq 0 \text{ and } \alpha_1, \alpha_2, \ldots, \alpha_n \in L\}$ and $\neg L$ is $\Sigma^\star - L$.

We next recall some notions related to cardinality. If A is any set, we let $|A|$ denote the *cardinal* of A, which expresses the size of A. When A is finite, $|A|$ is precisely the number of elements of A and we call it a *finite cardinal*. Infinite sets can have different cardinals, called *transfinite* or even *infinite*. For example, natural numbers $\mathbb{N}$ have the cardinal $\aleph_0$ (pronounced "aleph zero") and real numbers $\mathbb{R}$ have the cardinal c, also called the *cardinal of the continuum*. Two sets A and B are said to have the same cardinal, written $|A| = |B|$, iff there is some bijective mapping between the two. We write $|A| \leq |B|$ iff there is some injective mapping from A to B.

The famous *Cantor-Bernstein-Schroeder theorem* states that if $|A| \leq |B|$

and $|B| \leq |A|$ then $|A| = |B|$. In other words, to show that there is some bijection between sets A and B, it suffices to find an injection from A to B and an injection from B to A. The two injections need not be bijections. For example, the inclusion of the interval $(0, 1)$ in $\mathbb{R}^+$ is obviously an injection, so $|(0, 1)| \leq |\mathbb{R}^+|$. On the other hand, the function $x \rightsquigarrow x/(2x + 1)$ from $\mathbb{R}^+$ to $(0, 1)$ (in fact its codomain is the interval $(0, 1/2)$) is also injective, so $|\mathbb{R}^+| \leq |(0, 1)|$. Neither of the two injective functions is bijective, yet by the Cantor-Bernstein-Schroeder theorem there is some bijection between $(0, 1)$ and $\mathbb{R}^+$, that is, $|(0, 1)| = |\mathbb{R}^+|$. We will use this theorem to relate the various types of safety properties; for example, we will show that there is an injective function from safety properties over finite traces to safety properties over infinite traces and another injective function in the opposite direction. Unfortunately, the Cantor-Bernstein-Schroeder theorem is existential: it only says that some bijection exists between the two sets, but it does not give us an explicit bijection. Since the visualization of a concrete bijection between different sets of safety properties can be very meaningful, we will avoid using the Cantor-Bernstein-Schroeder theorem when we can find an explicit bijection between two sets of safety properties.

If A is a set of cardinal α, then 2^α is the cardinal of $\mathcal{P}(A)$, the power set of A (the set of subsets of A). It is known that $2^{\aleph_0} = c$, that is, there are as many sets of natural numbers as real numbers. The famous, still unanswered *continuum hypothesis*, states that there is no set whose size is strictly between $\aleph_0$ and c; more generally, it states that, for any transfinite cardinal α, there is no proper cardinal between α and 2^α. If A and B are infinite sets, then $|A| + |B|$ and $|A| \cdot |B|$ are the cardinals of the sets $A \cup B$ and $A \times B$, respectively. An important property of transfinite cardinals is that of *absorption* – the larger cardinal absorbs the smaller one: if α and β are transfinite cardinals such that $\alpha \leq \beta$, then $\alpha + \beta = \alpha \cdot \beta = \beta$; in particular, $c \cdot 2^c = 2^c$. Besides sets of natural numbers, there are several other important sets that have cardinal c: streams (i.e., infinite sequences) of Booleans, streams of reals, non-empty closed or open intervals of reals, as well as the sets of all open or closed sets of reals, respectively (Exercise 1).

For our purposes, if Σ is an enumerable set of states, then $\Sigma^\star$ is also enumerable, so it has cardinal $\aleph_0$. Also, if $|\Sigma| \leq c$, in particular if it is finite, then Σ^ω has the cardinal c, because it is equivalent to streams of states. We can then immediately infer that the set of finite-trace properties over Σ has cardinal $2^{\aleph_0} = c$, while the set of infinite-trace properties has cardinal 2^c.

Exercises

Exercise 1 *Show that each of the following sets have cardinal c: streams (i.e., infinite sequences) of Booleans; streams of natural numbers; streams of real numbers; closed intervals of real numbers; open intervals of real numbers; closed sets of real numbers; open sets of real numbers.*

Chapter 3

Safety Properties

Intuitively, a safety property of a system is one stating that the system cannot "go wrong", or, as Lamport [36] put it, that the "bad thing" never happens. In other words, in order for a system to violate a safety property, it should eventually "go wrong" or the "bad thing" should eventually happen. There is a very strong relationship between safety properties and runtime monitoring: if a safety property is violated by a running system, then the violation should happen *during* the execution of the system, in a finite amount of time, so a monitor for that property observing the running system should be able to detect the violation; an additional point in the favor of monitoring is that, if a system violates a safety property at some moment during its execution, then there is no way for the system to continue its execution to eventually satisfy the property, so a monitor needs not wait for a better future once it detects a bad present/past.

State properties or assertions that need only the current state of the running system to check whether they are violated or not, such as "no division by 0", or "x positive", or no deadlock, are common safety properties; once violated, one can stop the computation or take corrective measures. However, there are also interesting safety properties that involve more than one state of the system, such as "if one uses resource x then one must have authenticated at some moment in the past", or "any start of a process must be followed by a stop within 10 units of time", or "take command from user only if the user has logged in at some moment in the past and has not logged out since then", etc. Needless to say that the atomic events, or states, which form execution traces on which safety properties are defined, can be quite abstract: not all the details of a system execution are relevant

for the particular safety property of interest. In the context of monitoring, these relevant events or states can be extracted by means of appropriate instrumentation of the system. For example, runtime monitoring systems such as Tracematches [3] and MOP [10] use aspect-oriented technology to "hook" relevant observation points and appropriate event filters in a system.

It is customary to define safety properties as properties over *infinite traces*, to capture the intuition that they are defined for systems that can potentially run forever, such as reactive systems. A point in favor of infinite traces is that finite traces can be regarded as special cases of infinite traces, namely ones that "stutter" indefinitely in their last state (see, for example, Abadi and Lamport [1, 2]). Infinite traces are particularly desirable when one specifies safety properties using formalisms that have infinite-trace semantics, such as linear temporal logics or corresponding automata.

While "infinity" is a convenient abstraction that is relatively broadly-accepted nowadays in mathematics and in theoretical foundations of computer science, there is no evidence so far that a system can have an infinite-trace behavior (we have not seen any). A disclaimer is in place here: we do *not* advocate finite-traces as a foundation for safety properties; all we try to do is to argue that, just because they can be seen as a special case of infinite traces, finite traces are not entirely uninteresting. For example, a safety property associated to a one-time-access key issued to a client can be "activate, then use at most once, then close". Using regular patterns over the alphabet of relevant events $\Sigma = \{activate, use, close\}$, this safety property can be expressed as "$activate \cdot (\epsilon + use) \cdot close$"; any trace that is not a prefix of the language of this regular expression violates the property, including any other activation or use of the key after it was closed. While these finite-trace safety properties can easily be expressed as infinite-trace safety properties, we believe that that would be more artificial than simply accepting that in practice we deal with many finite-trace safety properties.

In this section we discuss various approaches to formalize safety properties and show that they are ultimately directly or indirectly equivalent. We categorize them into finite-trace safety properties, infinite-trace safety properties, and finite- and infinite-trace safety properties:

1. Section 3.1 defines safety properties over finite traces as prefix closed properties. A subset of finite-trace safety properties, that we call *persistent*, contain only traces that "have a future" within the property, that is, finite traces that can be continued into other finite traces that are also in the safety property. Persistent safety properties appear to

be the right finite-trace variant that corresponds faithfully to the more conventional infinite-trace safety properties. Even though persistent safety properties form a proper subset of finite-trace safety properties and each finite-trace safety property has a largest persistent safety property included in it, we show that there is in fact a bijection between safety properties and persistent safety properties by showing them both to have the cardinal of the continuum c.

2. In Section 3.2, we consider two standard infinite-trace definitions of a safety property, one based on the intuition that violating behaviors must manifest so after a finite number of events and the other based on the intuition of a safety property as a closed set in an appropriate topology over infinite-traces. We show them both equivalent to persistent safety properties over finite traces, by constructing an explicit bijection (as opposed to using cardinality arguments and infer the existence of a bijection); consequently, infinite-trace safety properties also have the cardinal of the continuum c. Since closed sets of real numbers are in a bijective correspondence with the real numbers, we indirectly rediscover Alpern and Schneider's result [4] stating that infinite-trace safety properties correspond to closed sets in infinite-trace topology.

3. Section 3.3 considers safety properties defined over both finite and infinite traces. We discuss two definitions of such safety properties encountered in the literature, and, using cardinality arguments, we show their equivalence with safety properties over only finite traces. In particular, safety properties over finite and infinite traces also have the cardinality of the continuum c. We also show that prefix-closeness is not a sufficient condition to characterize (not even bijectively) such safety properties, by showing that there are significantly more (2^c) prefix-closed properties over finite and infinite traces than safety properties.

Therefore, each of the classes of safety properties is in bijection with the real numbers. Since there are so many safety properties, we can also insightfully conclude that there is *no* enumerable mechanism to define all the safety properties, because $\aleph_0 \lneq c$. Therefore, particular logical or syntactic recursive formalisms can only define *some* of the safety properties, but not all of them.

3.1 Finite Traces

One of the most common intuitions for a safety property is as a prefix-closed set of finite traces. This captures best the intuition that once something bad happened, there is no way to recover: if $w \notin P$ then there is no u such that $P(wu)$, which is equivalent to saying that if $P(wu)$ then $P(w)$, which is equivalent to saying that P is prefix closed. From a monitoring perspective, a prefix closed property can be regarded as one containing all the good (complete or partial) behaviors of the observed system: once a state is encountered that does not form a good behavior together with the previously observed states, then a violation can be reported.

Definition 1 *Let* $\mathit{prefixes} : \Sigma^\star \to \mathcal{P}(\Sigma^\star)$ *be the prefix function returning for any finite trace all its prefixes, and let* $\mathit{prefixes} : \mathcal{P}(\Sigma^\star) \to \mathcal{P}(\Sigma^\star)$ *be its corresponding closure operator that takes sets of finite traces and closes them under prefixes.*

Note that $\mathsf{prefixes} : \mathcal{P}(\Sigma^\star) \to \mathcal{P}(\Sigma^\star)$ is indeed a closure operator (Exercise 2): it is extensive ($P \subseteq \mathsf{prefixes}(P)$), monotone ($P \subseteq P'$ implies $\mathsf{prefixes}(P) \subseteq \mathsf{prefixes}(P')$), and idempotent ($\mathsf{prefixes}(\mathsf{prefixes}(P)) = \mathsf{prefixes}(P)$).

Definition 2 *Let* $\mathit{Safety}^\star$ *be the set of finite-trace prefix-closed properties, that is, the set* $\{P \in \mathcal{P}(\Sigma^\star) \mid P = \mathit{prefixes}(P)\}$. *In other words,* $\mathit{Safety}^\star$ *is the set of fixed points of the prefix operator* $\mathit{prefixes} : \mathcal{P}(\Sigma^\star) \to \mathcal{P}(\Sigma^\star)$.

The star superscript in $\mathsf{Safety}^\star$ reflects that its traces are finite; in the next section we will define a set Safety^ω of infinite-trace safety properties. Since $\mathsf{prefixes}(P) \in \mathsf{Safety}^\star$ for any $P \in \mathcal{P}(\Sigma^*)$, we can assume from here on that $\mathsf{prefixes} : \mathcal{P}(\Sigma^\star) \to \mathcal{P}(\Sigma^\star)$ is actually a function $\mathcal{P}(\Sigma^*) \to \mathsf{Safety}^\star$.

Example 1 Consider the one-time-access key safety property discussed above, saying that a client can "activate, then use at most once, and then close" the key. If $\Sigma = \{\mathit{activate}, \mathit{use}, \mathit{close}\}$, then this safety property can be expressed as the finite set of finite words

$$\{\epsilon, \mathit{activate}, \mathit{activate\ close}, \mathit{activate\ use}, \mathit{activate\ use\ close}\}$$

No other behavior is allowed. Now suppose that the safety policy is extended to allow multiple uses of the key once activated, but still no further events

once it is closed. The extended safety property has now infinitely many finite-traces:

$$\{\epsilon\} \cup \{activate\} \cdot \{use^n \mid n \in \mathbb{N}\} \cdot \{\epsilon, close\}.$$

Note that this property is indeed prefix-closed. A monitor in charge of online checking this safety property would report a violation if the first event is not *activate*, or if it encounters any second *activate* event, or if it encounters any event after a *close* event is observed, including another *close* event.

It is interesting to note that this finite-trace safety property encompasses both finite and infinite aspects. For example, it does not preclude behaviors in which one sees an *activate* event and then an arbitrary number of *use* events; *use* events can persist indefinitely after an *activate* event without violating the property. On the other hand, once a *close* event is encountered, no other event can be further seen. We will shortly see that the safety property above properly includes the *persistent* safety property $\{\epsilon\} \cup \{activate\, use^n \mid n \in \mathbb{N}\}$, which corresponds to the infinite-trace safety property $\{activate\, use^\omega\}$. □

While prefix closeness seems to be the right requirement for a safety property, one can argue that it is not sufficient. For example, in the context of reactive systems that supposedly run forever, one may think of a safety property as one containing safe finite traces, that is, ones for which the reactive system can always find a way to continue its execution safely. The definition of safety properties above includes, among other safety properties, the empty set of traces as well as all prefix-closed *finite* sets of finite traces; any reactive system will eventually violate such safety properties, so one can say that the definition of safety property above is too generous.

We next define *persistent safety properties* as ones that always allow a future; intuitively, an observed reactive system that is in a safe state can always (if persistent enough) find a way to continue its execution to a next safe state. This notion is reminiscent of "feasibility", a semantic characterization of fairness in [7], and of "machine closeness" [1, 49], also used in the context of fairness.

Definition 3 *Let* $\mathsf{PersistentSafety}^\star$ *be the set of finite-trace* persistent safety properties*, that is, safety properties* $P \in \mathsf{Safety}^\star$ *such that if* $P(w)$ *for some* $w \in \Sigma^\star$ *then there is some* $a \in \Sigma$ *such that* $P(wa)$*.*

If a persistent safety property is non-empty, then note that it must contain an infinite number of words. The persistency aspect of a finite-trace safety

property can be regarded, in some sense, as a liveness argument. Indeed, assuming that it is a "good thing" for a trace to be indefinitely continued, then a persistent safety property is one in which the "good thing" always eventually happens. If one takes the liberty to regard "stuck" computations as unfair, then the persistency aspect above can also be regarded as a fairness argument.

Another way to think of persistent safety properties is as a means to refer to infinite behaviors by means of finite traces. This view is, in some sense, dual to the more common approach to regard finite behaviors as infinite behaviors that stutter infinitely in a "last" state (see, for example, Abadi and Lamport [1, 2] for a formalization of such last-state infinite stuttering).

Note that if Σ is a degenerate set of events containing only one element, that is, if $|\Sigma| = 1$, then $|\mathsf{Safety}^\star| = \aleph_0$ and $|\mathsf{PersistentSafety}^\star| = 2$; indeed, if $\Sigma = \{a\}$ then $\mathsf{Safety}^\star$ contains precisely the finite properties $a^{\leq n} = \{a^i \mid 0 \leq i \leq n\}$ for each $n \in \mathbb{N}$ plus the infinite property $\{a^n \mid n \in \mathbb{N}\}$, so a total of $\aleph_0 + 1 = \aleph_0$ properties, while $\mathsf{PersistentSafety}^\star$ contains only two properties, namely $\emptyset$ and $\{a^n \mid n \in \mathbb{N}\}$. The case when there is only one event or state in Σ is neither interesting nor practical. Therefore, from here on in this paper we take the liberty to assume that $|\Sigma| \geq 2$. Since in practice Σ contains states or events generated by a computer, for simplicity in stating some of the subsequent results, we also take the liberty to assume that $|\Sigma| \leq \aleph_0$; therefore, Σ can be any finite or recursively enumerable set, including $\mathbb{N}$, $\mathbb{N}_\infty$, $\mathbb{Q}$, etc., but cannot be $\mathbb{R}$ or any set "larger" than $\mathbb{R}$. With these assumptions, it follows that $|\Sigma^\star| = \aleph_0$ (finite words are recursively enumerable) and $|\Sigma^\omega| = c$ (infinite streams have the cardinality of the continuum).

Proposition 1 *$\mathsf{Safety}^\star$ and $\mathsf{PersistentSafety}^\star$ are closed under union; $\mathsf{Safety}^\star$ is also closed under intersection.*

Proof: The union and the intersection of prefix-closed properties is also prefix-closed. Also, the union of persistent prefix-closed properties is also persistent. □

The intersection of persistent safety properties may not be persistent:

Example 2 Let Σ be the set $\{0, 1\}$. Let $P = \{1^m \mid m \in \mathbb{N}\}$ and $P' = \{\epsilon\} \cup \{10^m \mid m \in \mathbb{N}\}$ be two persistent safety properties, where ϵ is the empty word (the word containing no letters). Then $P \cap P'$ is the finite safety property $\{\epsilon, 1\}$, which is not persistent. If one thinks that this happened because $P \cap P'$ does not contain any proper (i.e., non-empty)

persistent property, then one can take instead the persistent safety properties $P = \{0^n \mid n \in \mathbb{N}\} \cdot \{1^m \mid m \in \mathbb{N}\}$ and $P' = \{0^n \mid n \in \mathbb{N}\} \cdot (\{\epsilon\} \cup \{10^m \mid m \in \mathbb{N}\}$, whose intersection is the safety property $\{0^n \mid n \in \mathbb{N}\} \cup \{0^n 1 \mid n \in \mathbb{N}\}$. This safety property is not persistent because its words ending in 1 cannot persist, but it contains the proper persistent safety property $\{0^n \mid n \in \mathbb{N}\}$. □

Therefore, we can associate to any safety property in $\mathsf{Safety}^\star$ a largest persistent safety property in $\mathsf{PersistentSafety}^\star$, by simply taking the union of all persistent safety properties that are included in the original safety property (the empty property is one of them, the smallest):

Definition 4 *For a safety property $P \in \mathsf{Safety}^\star$, let $P^\circ \in \mathsf{PersistentSafety}^\star$ be the largest persistent safety property with $P^\circ \subseteq P$.*

The following example shows that one may need to eliminate infinitely many words from a safety property in order to obtain a persistent safety property:

Example 3 Let $\Sigma = \{0, 1\}$ and let P be the safety property $\{0^n \mid n \in \mathbb{N}\} \cup \{0^n 1 \mid n \in \mathbb{N}\}$. Then P° can contain no word ending with a 1 and can contain all the words of 0's. Therefore, $P^\circ = \{0^n \mid n \in \mathbb{N}\}$. □

Finite safety properties obviously cannot contain any non-empty persistent safety property, that is, $P^\circ = \emptyset$ if P is finite. But what if P is infinite? Is it always the case that it contains a non-empty persistent safety property? Interestingly, it turns out that this is true if and only if Σ is finite:

Proposition 2 *If Σ is finite and P is a safety property containing infinitely many words, then $P^\circ \neq \emptyset$.*

Proof: For each letter $a \in \Sigma$, let us define the *derivative of P wrt a*, written $\delta_a(P)$, as the language $\{w \in \Sigma^\star \mid aw \in P\}$. Since

$$P = \{\epsilon\} \cup \bigcup_{a \in \Sigma} \{a\} \cdot \delta_a(P)$$

since Σ is finite, and since P is infinite, it follows that there is some $a_1 \in \Sigma$ such that $\delta_{a_1}(P)$ is infinite; note that $a_1 \in P$ since P is prefix closed. Similarly, since $\delta_{a_1}(P)$ is infinite, there is some $a_2 \in \Sigma$ such that $\delta_{a_2}(\delta_{a_1}(P))$ is infinite and $a_1 a_2 \in P$. Iterating this reasoning, we can find some $a_n \in \Sigma$

for each $n \in \mathbb{N}$, such that $a_1 a_2 \dots a_n \in P$ and $\delta_{a_n}(\cdots(\delta_{a_2}(\delta_{a_1}(P)))\cdots)$ is infinite, that is, the set $\{w \in \Sigma^\star \mid a_1 a_2 \dots a_n w \in P\}$ is infinite. It is now easy to see that the set $\{a_1 a_2 \dots a_n \mid n \in \mathbb{N}\} \subseteq P$ is persistent. Therefore, $P^\circ \neq \emptyset$. □

The following example shows that Σ must indeed be finite in order for the result above to hold:

Example 4 Consider some infinite set of events or states Σ. Then we can label distinct elements in Σ with distinct labels in $\mathbb{N} \cup \{\infty\}$. We only need these elements from Σ; therefore, without loss of generality, we can assume that $\Sigma = \mathbb{N} \cup \{\infty\}$. Let P be the safety property

$$\{\epsilon\} \cup \{\infty\, n\, (n-1) \dots (m+1)\, m \mid 0 \leq m \leq n+1\},$$

where ϵ is the empty word (the word containing no letters) and $n \dots (n+1)$ is also the empty word for any $n \in \mathbb{N}$. Then P° is the empty property. Indeed, note that any persistent safety property P' included in P cannot have traces ending in 0, because those cannot be continued into other traces in P; since P' cannot contain traces ending in 0, it cannot contain traces ending in 1 either, because such traces can only be continued with a 0 letter into traces in P, but those traces have already been decided that cannot be part of P'; inductively, one can show that P' can contain no words ending in letters that are natural numbers in $\mathbb{N}$. Since the only trace in P ending in ∞ is ∞ itself and since ∞ can only be continued with a natural number letter into a trace in P but such trace cannot belong to P', we deduce that P' can contain no word with letters in Σ. In particular, P° must be empty. □

Even though we know that the largest persistent safety property P° included into a safety property P always exists because $\mathsf{PersistentSafety}^\star$ is closed under union, we would like to have a more constructive way to obtain it. A first and obvious thing to do is to eliminate from P all the "stuck" computations, that is, those which cannot be added any new state to obtain a trace that is also in P. This removal step does not destroy the prefix-closeness of P, but it may reveal new computations which are stuck. By iteratively eliminating all the computations that get stuck in a finite number of steps, one would expect to obtain a persistent safety property, namely precisely P°. It turns out that this is indeed true only if Σ is finite. If that is the case, then the following can also be used as an alternative definition of P°:

Proposition 3 *Given safety property $P \in \mathsf{Safety}^\star$, then let P^- be the property $\{w \in P \mid (\exists a \in \Sigma)\, wa \in P\}$. Also, let $\{P_i \mid i \in \mathbb{N}\}$ be properties defined as $P_0 = P$ and $P_{i+1} = P_i^-$ for all $i \geq 0$. Then $P^\circ = \bigcap_{i \geq 0} P_i$ whenever Σ is finite.*

Proof: It is easy to see that if P is prefix-closed then $P^- \subseteq P$ is also prefix-closed, so P^- is also a property in $\mathsf{Safety}^\star$. Therefore, the properties P_i form a sequence $P = P_0 \supseteq P_1 \supseteq P_2 \supseteq \cdots$ of increasingly smaller safety properties.

Let us first prove that $\bigcap_{i \geq 0} P_i$ is a persistent safety property. Assume by contradiction that for some $w \in \bigcap_{i \geq 0} P_n$ there is no $a \in \Sigma$ such that $wa \in \bigcap_{i \geq 0} P_i$. In other words, we can find for each $a \in \Sigma$ some $i_a \geq 0$ such that $wa \notin P_{i_a}$. Since Σ is finite, we can let i be the largest among the natural numbers $i_a \in \mathbb{N}$ for all $a \in \Sigma$. Since $P_i \subseteq P_{i_a}$ for all $a \in \Sigma$, it should be clear that there is no $a \in \Sigma$ such that $wa \in P_i$, which means that $w \notin P_{i+1}$. This contradicts the fact that $w \in \bigcap_{i \geq 0} P_i$. Therefore, $\bigcap_{i \geq 0} P_i \in \mathsf{PersistentSafety}^\star$.

Let us now prove that $\bigcap_{i \geq 0} P_i$ is the largest persistent safety property included in P. Let P' be any persistent safety property included in P. We show by induction on i that $P' \subseteq P_i$ for all $i \in \mathbb{N}$. The base case, $P' \subseteq P_0$, is obvious. Suppose that $P' \subseteq P_i$ for some $i \in \mathbb{N}$ and let $w \in P'$. Since P' is persistent, there is some $a \in \Sigma$ such that $wa \in P' \subseteq P_i$, which means that $w \in P_{i+1}$. Since w was chosen arbitrarily, it follows that $P' \subseteq P_{i+1}$. Therefore, $P' \subseteq \bigcap_{i \geq 0} P_i$. □

We next show that the finiteness of Σ was a necessary requirement in order for the result above to hold. In other words, we show that if Σ is allowed to be infinite then we can find a safety property $P \in \mathsf{Safety}^\star$ over Σ such that $P^\circ \in \mathsf{PersistentSafety}^\star$ and $\bigcap_{i \geq 0} P_i \in \mathsf{Safety}^\star$ are distinct. Since we showed in the proof of Proposition 3 that any persistent safety property P' is included in $\bigcap_{i \geq 0} P_i$, it follows that $P^\circ \subseteq \bigcap_{i \geq 0} P_i$. Since P° is the largest persistent safety property included in P, one can easily show that $P^\circ = (\bigcap_{i \geq 0} P_i)^\circ$. Therefore, it suffices to find a safety property P such that $\bigcap_{i \geq 0} P_i$ is not persistent, which is what we do in the next example:

Example 5 Consider the safety property P over infinite $\Sigma = \mathbb{N} \cup \{\infty\}$ discussed in Example 4, namely $\{\epsilon\} \cup \{\infty\, n\, (n-1) \ldots (m+1)\, m \mid 0 \leq m \leq n+1\}$. Then one can easily show by induction on $i \in \mathbb{N}$ that the properties P_i defined in Proposition 3 are the sets $\{\epsilon\} \cup \{\infty\, n\, (n-1) \ldots (m+1)\, m \mid i \leq m \leq n+1\}$; in other words, each P_i excludes from P all the words whose

last letters are smaller than i when regarded as natural numbers. Then the intersection $\bigcap_{i\geq 0} P_i$ contains no trace ending in a natural number; the only possibility left is then $\bigcap_{i\geq 0} P_i = \{\epsilon, \infty\}$, which is different from $P^\circ = \emptyset$ (see Example 4).

One may argue that $P^\circ \neq \bigcap_{i\geq 0} P_i$ above happened precisely because P° was empty. One can instead pick the safety property $Q = \{0^n \mid n \in \mathbb{N}\} \cdot P$. Then one can show following the same idea as in Example 4 that $Q^\circ = \{0^n \mid n \in \mathbb{N}\}$. Further, one can show that $Q_i = \{0^n \mid n \in \mathbb{N}\} \cdot P_i$, so $\bigcap_{i\geq 0} Q_i = \{0^n \mid n \in \mathbb{N}\} \cup \{0^n\infty \mid n \in \mathbb{N}\}$, which is different from Q°. $\square$

Persistency is reminiscent of "feasibility" introduced by Apt et al. [7] in the context of fairness, and of "machine closeness" introduced by Abadi and Lamport [1, 2] (see also Schneider [49]) in the context of refinement. Let us use the terminology "machine closeness": a property L (typically a liveness or a fairness property) is *machine closed* for a property M (typically given as the language of some state machine) iff L does not prohibit any of the observable runtime behaviors of M, that is, iff $\mathsf{prefixes}(M) = \mathsf{prefixes}(M \cap L)$; for example, if M is the total property (i.e., every event is possible at any moment, i.e., $M = \Sigma^\star$) and L is the property stating that "always eventually event a", then any prefix of M can be continued to obtain a property satisfying L. Persistency is related to machine closeness in that a safety property P is persistent if and only if P° is machine closed for P. In other words, there is nothing P can do in a finite amount of time that P° cannot do. However, there is a caveat here: since liveness and fairness are inherently infinite-trace notions, machine closeness (or feasibility) have been introduced in the context of infinite-traces. On the other hand, persistency makes sense only in the context of finite traces.

It is clear that $\mathsf{PersistentSafety}^\star$ is properly included in $\mathsf{Safety}^\star$. Yet, we next show that, surprisingly, there is a bijective correspondence between $\mathsf{Safety}^\star$ and $\mathsf{PersistentSafety}^\star$, both having the cardinal of the continuum:

Theorem 1 $|\mathit{PersistentSafety}^\star| = |\mathit{Safety}^\star| = c$.

Proof: Since $\Sigma^\star$ is recursively enumerable and since $2^{\aleph_0} = c$, we can readily infer that $|\mathsf{PersistentSafety}^\star| \leq |\mathsf{Safety}^\star| \leq |\mathcal{P}(\Sigma^\star)| = c$.

Let us now define an injective function φ from the open interval of real numbers $(0, 1)$ to $\mathsf{PersistentSafety}^\star$. Since $|\Sigma| \geq 2$, let us distinguish two different elements in Σ and let us label them $\overline{0}$ and $\overline{1}$. For a real $r \in (0, 1)$, let $\varphi(r)$ be the set $\{\overline{\alpha} \mid \alpha \in \{0, 1\}^\star \text{ and } 0.\alpha < r\}$, where $0.\alpha$

is the (rational) number in $(0,1)$ whose decimals in binary representation are α, and where $\overline{\alpha}$ is the word in $\Sigma^\star$ corresponding to α. Note that the set $\varphi(r) \in \mathcal{P}(\Sigma^\star)$ is prefix-closed for any $r \in (0,1)$, and that if $w \in \varphi(r)$ then also $w\overline{0} \in \varphi(r)$ (the latter holds since, by real numbers conventions, $0.\alpha = 0.\alpha 0$), so $\varphi(r) \in \mathsf{PersistentSafety}^\star$. Since the set of rationals with finite number of decimals in binary representation is dense in $\mathbb{R}$ (i.e., it intersects any open interval in $\mathbb{R}$) and in particular in the interval $(0,1)$, it follows that the function $\varphi : (0,1) \to \mathsf{PersistentSafety}^\star$ is injective: indeed, if $r_1 \neq r_2 \in (0,1)$, say $r_1 < r_2$, then there is some $\alpha \in \{0,1\}^\star$ such that $r_1 < 0.\alpha < r_2$, so $\varphi(r_1) \neq \varphi(r_2)$. Since the interval $(0,1)$ has the cardinal of the continuum c, the existence of the injective function φ implies that $c \leq |\mathsf{PersistentSafety}^\star|$. By the Cantor-Bernstein-Schroeder theorem it follows that $|\mathsf{PersistentSafety}^\star| = |\mathsf{Safety}^\star| = c$. □

From safety: *The proof above could have been rearranged to avoid the need to use the set PersistentSafety*. However, we prefer to keep it for two reasons:*

1. *For finite-traces, persistent safety properties appear to be more natural in the context of reactive systems than just prefix closed properties;*
2. *Persistent safety properties play a technical bridge role in the next section to show that the infinite-trace safety properties also have the cardinal c.*

With regards to finite-traces, persistent safety properties appear to be more natural in the context of reactive systems than just prefix-closed properties. Also, persistent safety properties play a technical bridge role in the next section to show that the infinite-trace safety properties also have the cardinal c.

3.2 Infinite Traces

The finite-trace safety properties defined above, persistent or not, rely on the intuition of a correct prefix: a safety property is identified with the set of all its finite prefixes. In the case of a persistent safety property, each "informal" infinite acceptable behavior is captured by its infinite set of finite prefixes. Even though persistent safety properties appear to capture well in a finite-trace setting the intuition of safety in the context of (infinite-trace) reactive

systems, one could argue that it does not say anything about unacceptable infinite traces. Indeed, one may think that persistent safety properties do not capture the intuition that if an infinite trace is unacceptable then there must be some finite prefix of it which is already unacceptable. In this section we show that there is in fact a bijection between safety properties over infinite traces and persistent safety properties over finite traces as we defined them in the previous section.

We start by extending the $\mathsf{prefixes}$ function to infinite traces:

Definition 5 *Let* $\mathit{prefixes} : \Sigma^\omega \to \mathcal{P}(\Sigma^\star)$ *be the function returning for any infinite trace* u *all its finite prefixes* $\mathit{prefixes}(u)$*, and let* $\mathit{prefixes} : \mathcal{P}(\Sigma^\omega) \to \mathcal{P}(\Sigma^\star)$ *be its corresponding extension to sets of infinite traces.*

Note that $\mathsf{prefixes}(S) \in \mathsf{PersistentSafety}^\star$ for any $S \in \mathcal{P}(\Sigma^\omega)$, so $\mathsf{prefixes}$ is in fact a function $\mathcal{P}(\Sigma^\omega) \to \mathsf{PersistentSafety}^\star$.

The definition of safety properties over infinite traces below appears to be the most used definition of a safety property in the literature; at our knowledge, it was formally introduced by Alpern and Schneider [4], but they credit the insights of their definition to Lamport [36].

Definition 6 *Let* Safety^ω *be the set of infinite-trace properties* $Q \in \mathcal{P}(\Sigma^\omega)$ *s.t.: if* $u \notin Q$ *then there is a finite trace* $w \in \mathit{prefixes}(u)$ *s.t.* $wv \notin Q$ *for any* $v \in \Sigma^\omega$*.*

In other words, if an infinite behavior violates the safety property then there is some finite-trace "violation threshold"; once the violation threshold is reached, there is no chance to recover.

The following proposition can serve as an alternative and more compact definition of Safety^ω:

Proposition 4 $\mathit{Safety}^\omega = \{Q \in \mathcal{P}(\Sigma^\omega) \mid u \in Q \text{ iff } \mathit{prefixes}(u) \subseteq \mathit{prefixes}(Q)\}$.

Proof: Since $u \in Q$ implies $\mathsf{prefixes}(u) \subseteq \mathsf{prefixes}(Q)$, the only thing left to show is that $Q \in \mathsf{Safety}^\omega$ iff "$\mathsf{prefixes}(u) \subseteq \mathsf{prefixes}(Q)$ implies $u \in Q$"; the latter is equivalent to "$u \notin Q$ implies $\mathsf{prefixes}(u) \not\subseteq \mathsf{prefixes}(Q)$", which is further equivalent to "$u \notin Q$ implies there is some $w \in \mathsf{prefixes}(u)$ s.t. $w \notin \mathsf{prefixes}(Q)$", which is indeed equivalent to $Q \in \mathsf{Safety}^\omega$. □

Another common intuition for safety properties over infinite traces is as *closed* sets in the topology corresponding to Σ^ω. Alpern and Schneider captured formally this intuition for the first time in [4]; then it was used as a convenient definition of safety by Abadi and Lamport [1, 2] among others:

Definition 7 *An infinite sequence $u^{(1)}$, $u^{(2)}$, ..., of infinite traces in Σ^ω* converges to $u \in \Sigma^\omega$, *or u is a* limit of $u^{(1)}$, $u^{(2)}$, ...,, *written $u = \lim_i u^{(i)}$, iff for all $m \geq 0$ there is an $n \geq 0$ such that $u_1^{(i)} u_2^{(i)} \dots u_m^{(i)} = u_1 u_2 \dots u_m$ for all $i \geq n$. If $Q \in \mathcal{P}(\Sigma^\omega)$ then $\overline{Q}$, the* closure *of Q, is the set $\{\lim_i u^{(i)} \mid u^{(i)} \in Q \text{ for all } i \in \mathbb{N}\}$.*

It can be easily shown that the overline closure above is indeed a closure operator on Σ^ω, that is, it is extensive ($Q \subseteq \overline{Q}$), monotone ($Q \subseteq Q'$ implies $\overline{Q} \subseteq \overline{Q'}$), and idempotent ($\overline{\overline{Q}} = \overline{Q}$); see Exercise 6.

Definition 8 *Let $\mathsf{Safety}^\omega_{\lim}$ be the set of properties $\{Q \in \mathcal{P}(\Sigma^\omega) \mid Q = \overline{Q}\}$.*

As expected, the two infinite-trace safety property definitions are equivalent; we have not found any formal proof in the literature, so for the sake of completeness we give a simple proof here:

Proposition 5 $\mathsf{Safety}^\omega_{\lim} = \mathsf{Safety}^\omega$.

Proof: All we need to prove is that for any $Q \in \mathcal{P}(\Sigma^\omega)$ and any $u \in \Sigma^\omega$, $\mathsf{prefixes}(u) \subseteq \mathsf{prefixes}(Q)$ iff $u = \lim_i u^{(i)}$ for some infinite sequence of infinite traces $u^{(1)}$, $u^{(2)}$, ... in Q. If $\mathsf{prefixes}(u) \subseteq \mathsf{prefixes}(Q)$ then one can find for each $i \geq 0$ some $u^{(i)} \in Q$ such that $u_1 u_2 \dots u_i = u_1^{(i)} u_2^{(i)} \dots u_i^{(i)}$, so for each $m \geq 0$ one can pick $n = m$ such that $u_1 u_2 \dots u_m = u_1^{(i)} u_2^{(i)} \dots u_m^{(i)}$ for all $i \geq n$, so $u = \lim_i u^{(i)}$. Conversely, if $u = \lim_i u^{(i)}$ for some infinite sequence of infinite traces $u^{(1)}$, $u^{(2)}$, ... in Q, then for any $m \geq 0$ there is some $n \geq 0$ such that $u_1 u_2 \dots u_m = u_1^{(n)} u_2^{(n)} \dots u_m^{(n)}$, that is, for any prefix of u there is some $u' \in Q$ having the same prefix, that is, $\mathsf{prefixes}(u) \subseteq \mathsf{prefixes}(Q)$. □

The next result establishes the relationship between infinite-trace safety properties and finite-trace persistent safety properties, by proposing a concrete bijective mapping relating the two (as opposed to using cardinality arguments to indirectly show only the existence of such a mapping). Therefore, there is also a bijective correspondence between safety properties over infinite traces and the real numbers:

Note there are 2^c properties over Σ^ω

Theorem 2 $|\mathsf{Safety}^\omega| = |\mathsf{PersistentSafety}^\star| = c$.

Proof: We show that there is a bijective function between the two sets of safety properties. Recall that $\mathsf{prefixes}(S) \in \mathsf{PersistentSafety}^\star$ for any $S \in \mathcal{P}(\Sigma^\omega)$, that is, that $\mathsf{prefixes}$ is a function $\mathcal{P}(\Sigma^\omega) \to \mathsf{PersistentSafety}^\star$. Let $\mathsf{prefixes} : \mathsf{Safety}^\omega \to \mathsf{PersistentSafety}^\star$ be the restriction of this prefix function to Safety^ω. Let us also define a function $\omega : \mathsf{PersistentSafety}^\star \to \mathsf{Safety}^\omega$ as follows: $\omega(P) = \{u \in \Sigma^\omega \mid \mathsf{prefixes}(u) \subseteq P\}$. This function is well-defined: if $u \notin \omega(P)$ then by the definition of $\omega(P)$ there is some $w \in \mathsf{prefixes}(u)$ such that $w \notin P$; since $w \in \mathsf{prefixes}(wv)$ for any $v \in \Sigma^\omega$, it follows that $wv \notin \omega(P)$ for any $v \in \Sigma^\omega$.

We next show that $\mathsf{prefixes}$ and ω are inverse to each other. Let us first show that $\mathsf{prefixes}(\omega(P)) = P$ for any $P \in \mathsf{PersistentSafety}^\star$. The inclusion $\mathsf{prefixes}(\omega(P)) \subseteq P$ follows by the definition of $\omega(P)$: $\mathsf{prefixes}(u) \subseteq P$ for any $u \in \omega(P)$. The inclusion $P \subseteq \mathsf{prefixes}(\omega(P))$ follows from the fact that P is a persistent safety property: for any $w \in P$ one can iteratively build an infinite sequence v_1, v_2, ..., such that $wv_1,\ wv_1v_2,\ ... \in P$, so $wv_1v_2... \in \omega(P)$. Let us now show that $\omega(\mathsf{prefixes}(Q)) = Q$ for any $Q \in \mathsf{Safety}^\omega$. The inclusion $Q \subseteq \omega(\mathsf{prefixes}(Q))$ is immediate. For the other inclusion, let $u \in \omega(\mathsf{prefixes}(Q))$, that is, $\mathsf{prefixes}(u) \subseteq \mathsf{prefixes}(Q)$. Suppose by contradiction that $u \notin Q$. Then there is some $w \in \mathsf{prefixes}(u)$ such that $wv \notin Q$ for any $v \in \Sigma^\omega$. Since $w \in \mathsf{prefixes}(u)$ and $\mathsf{prefixes}(u) \subseteq \mathsf{prefixes}(Q)$, it follows that $w \in \mathsf{prefixes}(Q)$, that is, that there is some $u' \in Q$ such that $u' = wv$ for some $v \in \Sigma^\omega$. This contradicts the fact that $wv \notin Q$ for any $v \in \Sigma^\omega$. Consequently, $u \in Q$.

The second part follows by Theorem 1. □

3.3 Finite and Infinite Traces

It is also common to define safety properties as properties over both finite and infinite traces, the intuition for the finite traces being that of unfinished computations. For example, Lamport [37] extends the notion of safety in Definition 6 to properties over both finite and infinite traces, while Schneider et al [50, 19] give an alternative definition of safety over finite and infinite traces. We define both approaches shortly and then show their equivalence and their bijective correspondence with real numbers. Before that, we argue that the mix of finite and infinite traces is less trivial than it may appear, by showing that there are significantly more prefix closed properties than in the case when only finite traces were considered.

Definition 9 *Let $\mathsf{PrefixClosed}^{\star,\omega}$ be the set of prefix-closed sets of finite and infinite traces: for $Q \subseteq \Sigma^\star \cup \Sigma^\omega$, $Q \in \mathsf{PrefixClosed}^{\star,\omega}$ iff $\mathsf{prefixes}(Q) \subseteq Q$.*

Also, let $\mathsf{PersistentPrefixClosed}^{\star,\omega}$ *be the set of* persistent *prefix-closed sets of finite and infinite traces: for* $Q \in \mathsf{PrefixClosed}^{\star,\omega}$*, it is the case that* $Q \in \mathsf{PersistentPrefixClosed}^{\star,\omega} \iff$ *if* $Q(w)$ *for some* $w \in \Sigma^\star$ *then that there is some* $a \in \Sigma$ *such that* $Q(wa)$*.*

The next result says that there is a bijective correspondence between prefix-closed and persistent prefix-closed properties also in the case of finite and infinite traces, but that there are exponentially more such properties than in the case of just finite traces:

Proposition 6 $|\mathsf{PersistentPrefixClosed}^{\star,\omega}| = |\mathsf{PrefixClosed}^{\star,\omega}| = 2^c$*.*

Proof: We show $2^c \leq |\mathsf{PersistentPrefixClosed}^{\star,\omega}| \leq |\mathsf{PrefixClosed}^{\star,\omega}| \leq 2^c$, where the middle inequality is immediate. For $2^c \leq |\mathsf{PersistentPrefixClosed}^{\star,\omega}|$, let us define $\varphi : \mathcal{P}((0,1)) \to \mathsf{PersistentPrefixClosed}^{\star,\omega}$ as

$$\varphi(R) = \bigcup_{0.\alpha \in R} \{\overline{\alpha}\} \cup \mathsf{prefixes}(\overline{\alpha})$$

where we assume for any real number in the interval $(0,1)$ its decimal binary representation $0.\alpha$ with $\alpha \in \{0,1\}^\omega$ (if the number is rational then α may contain infinitely many ending 0's), and $\overline{\alpha}$ is the infinite trace in Σ^ω replacing each 0 and 1 in α by $\overline{0}$ and $\overline{1}$, respectively, where $\overline{0}$ and $\overline{1}$ are two arbitrary but fixed distinct elements in Σ (recall that $|\Sigma| \geq 2$. Note that $\varphi(R)$ is well-defined: it is clearly prefix-closed and it is also persistent because its finite traces are exactly prefixes of infinite traces, so they admit continuations in $\varphi(R)$. It is easy to see that φ is injective. Since $|(0,1)| = c$, we conclude that $2^c \leq |\mathsf{PersistentPrefixClosed}^{\star,\omega}|$.

To show $|\mathsf{PrefixClosed}^{\star,\omega}| \leq 2^c$, note that any property in $\mathsf{PrefixClosed}^{\star,\omega}$ is a union of a subset in $\Sigma^\star$ and a subset in Σ^ω, so $|\mathsf{PrefixClosed}^{\star,\omega}| \leq 2^{|\Sigma^\star|} \cdot 2^{|\Sigma^\omega|}$. Since $|\Sigma^\star| = \aleph_0$, $|\Sigma^\omega| = c$, $2^{\aleph_0} = c$, and $c \cdot 2^c = 2^c$ (by absorption of transfinite cardinals), we get that $|\mathsf{PrefixClosed}^{\star,\omega}| \leq 2^c$. □

The fact that properties in $\mathsf{PersistentPrefixClosed}^{\star,\omega}$ contain also infinite traces was crucial in showing the injectivity of φ in the proof above. A similar construction for the finite trace setting does *not* work. Indeed, if one tries to define a function $\varphi : \mathcal{P}((0,1)) \to \mathsf{PersistentSafety}^\star$ as $\varphi(R) = \bigcup_{0.\alpha \in R} \mathsf{prefixes}(\overline{\alpha})$, then one can show it well-defined but cannot show it injective: e.g., $\varphi((0,0.5)) = \varphi((0,0.5])$.

Since safety properties over finite and infinite traces are governed by the same intuitions as safety properties over only finite or over only infinite

traces, the result above tells us that prefix closeness is not a sufficient condition to properly capture the safety properties. Schneider [50] proposes an additional condition in the context of his EM (execution monitoring) framework, namely that if an infinite trace is not in the property, then there is a finite prefix of it which is not in the property either. It is easy to see that this additional condition is equivalent to saying that an infinite trace is in the property whenever all its finite prefixes are in the property, which allows us to compactly define safety properties over finite and infinite traces in the EM style as follows:

Definition 10 $\mathit{Safety}^{\star,\omega}_{\mathsf{EM}} = \{Q \subseteq \Sigma^\star \cup \Sigma^\omega \mid u \in Q \textit{ iff } \mathit{prefixes}(u) \subseteq Q\}$.

Note that $\mathsf{Safety}^{\star,\omega}_{\mathsf{EM}} \subset \mathsf{PrefixClosed}^{\star,\omega}$. We will shortly show that $\mathsf{Safety}^{\star,\omega}_{\mathsf{EM}}$ is in fact much smaller than $\mathsf{PrefixClosed}^{\star,\omega}$, by showing that $|\mathsf{Safety}^{\star,\omega}_{\mathsf{EM}}| = c$.

The consecrated definition of a safety property in the context of both finite and infinite traces is perhaps the one proposed by Lamport in [37], which relaxes the one in Definition 6 by allowing u to range over both finite and infinite traces:

Definition 11 *Let* $\mathit{Safety}^{\star,\omega}$ *be the set of finite- and infinite-trace properties* $\{Q \subseteq \Sigma^\star \cup \Sigma^\omega \mid u \notin Q \Rightarrow (\exists w \in \mathit{prefixes}(u))\,(\forall v \in \Sigma^\star \cup \Sigma^\omega)\; wv \notin Q\}$

Schneider informally stated in [50] that the two definitions of safety above are equivalent. It is not hard to show it formally:

Proposition 7 $\mathit{Safety}^{\star,\omega}_{\mathsf{EM}} = \mathit{Safety}^{\star,\omega}$.

Proof: First note that $\mathsf{Safety}^{\star,\omega} \subseteq \mathsf{PrefixClosed}^{\star,\omega}$: if $wu \in Q \in \mathsf{Safety}^{\star,\omega}$ and $w \notin Q$ then there is some $w' \in \mathsf{prefixes}(w)$, say $w = w'w''$, such that $w'v \notin Q$ for any v, in particular $w'w''u \notin Q$, which contradicts $wu \in Q$.

$\mathsf{Safety}^{\star,\omega} \subseteq \mathsf{Safety}^{\star,\omega}_{\mathsf{EM}}$: let $Q \in \mathsf{Safety}^{\star,\omega}$ and $u \in \Sigma^\star \cup \Sigma^\omega$ s.t. $\mathsf{prefixes}(u) \subseteq Q$; if $u \notin Q$ then there is some $w \in \mathsf{prefixes}(u)$ s.t. $wv \notin Q$ for any v, in particular for v the empty word, that is, $w \notin Q$, which contradicts $\mathsf{prefixes}(u) \subseteq Q$.

$\mathsf{Safety}^{\star,\omega}_{\mathsf{EM}} \subseteq \mathsf{Safety}^{\star,\omega}$: let $u \notin Q \in \mathsf{Safety}^{\star,\omega}_{\mathsf{EM}}$; then $\mathsf{prefixes}(u) \not\subseteq Q$, that is, there is some $w \in \mathsf{prefixes}(u)$ s.t. $w \notin Q$; since Q is prefix-closed, it follows that $wv \notin Q$ for any $v \in \Sigma^\star \cup \Sigma^\omega$. □

We next show that there is a bijective correspondence between the safety properties over finite or infinite traces above and the finite trace safety properties in Section 3.1:

Theorem 3 $|\mathit{Safety}^{\star,\omega}| = |\mathit{Safety}^{\star,\omega}_{\mathsf{EM}}| = |\mathit{Safety}^{\star}| = c$.

Proof: $\mathsf{Safety}^{\star} \subset \mathsf{Safety}^{\star,\omega}_{\mathsf{EM}}$ since the properties in $\mathsf{Safety}^{\star,\omega}_{\mathsf{EM}}$ are prefix-closed, so $|\mathsf{Safety}^{\star}| \leq |\mathsf{Safety}^{\star,\omega}_{\mathsf{EM}}|$.

Since the functions $\mathsf{prefixes} : \mathcal{P}(\Sigma^{\star}) \to \mathcal{P}(\Sigma^{\star})$ and $\mathsf{prefixes} : \mathcal{P}(\Sigma^{\omega}) \to \mathcal{P}(\Sigma^{\star})$ have actual co-domains $\mathsf{Safety}^{\star}$ and $\mathsf{PersistentSafety}^{\star}$, respectively, they can be organized as a function $\mathsf{prefixes} : \mathsf{Safety}^{\star,\omega}_{\mathsf{EM}} \to \mathsf{Safety}^{\star}$. Let us show that this function is injective. Let us assume $Q \neq Q' \in \mathsf{Safety}^{\star,\omega}_{\mathsf{EM}}$, say $u \in Q$ and $u \notin Q'$, s.t. $\mathsf{prefixes}(Q) = \mathsf{prefixes}(Q')$. Since $u \in Q \in \mathsf{Safety}^{\star,\omega}_{\mathsf{EM}}$ it follows that $\mathsf{prefixes}(u) \subseteq \mathsf{prefixes}(Q) \subseteq Q$, which implies that $\mathsf{prefixes}(u) \subseteq \mathsf{prefixes}(Q') \subseteq Q'$; since $Q' \in \mathsf{Safety}^{\star,\omega}_{\mathsf{EM}}$, it follows that $u \in Q'$, contradiction. Therefore, $\mathsf{prefixes} : \mathsf{Safety}^{\star,\omega}_{\mathsf{EM}} \to \mathsf{Safety}^{\star}$ is injective, which proves that $\mathsf{Safety}^{\star,\omega}_{\mathsf{EM}} \leq \mathsf{Safety}^{\star}$.

The rest follows by Proposition 7 and Theorem 1. □

3.4 "Always Past" Characterization

Another common way to specify safety properties is by giving an arbitrary property on finite traces, not necessarily prefix closed, and then to require that any acceptable behavior must have all its finite prefixes in the given property. A particularly frequent case is when one specifies the property of the finite-prefixes using the past-time fragment of linear temporal logics (LTL). For example, Manna and Pnueli [39] call the resulting "always (past LTL)" properties *safety formulae*; many other authors, including ourselves, adopted the terminology "safety formula" from Manna and Pnueli, although some qualify it as "LTL safety formula". An example of an LTL safety formula is "always (b implies eventually in the past a)", written using LTL notation as "$\Box(b \to \diamond a)$"; here the past time formula "$b \to \diamond a$" compactly specifies all the finite-traces

$$\{wsw's' \mid w, w' \in \Sigma^{\star},\ s, s' \in \Sigma,\ a(s) \text{ and } b(s') \text{ hold}\} \bigcup$$
$$\{ws \mid w \in \Sigma^{\star},\ s \in \Sigma,\ b(s) \text{ does not hold}\}.$$

From safety: *We will investigate the case when safety properties are expressed as LTL safety formulae, as well as optimal monitoring techniques for such safety properties, in Section 6.3.*

In the remainder of this section we assume that the past time prefix properties are given as ordinary sets of finite-traces (so we make abstraction of how

these properties are expressed) and show not only that the resulting "always past" properties are safety properties, but also that any safety properties can be expressed as an "always past" property. This holds for all the variants of safety properties (i.e., over finite traces, over infinite traces, or over both finite and infinite traces).

Definition 12 *Let $P \subseteq \Sigma^\star$ be any property over finite traces. Then we define the "always past" property $\Box P$ as follows:*

(finite traces) $\{w \in \Sigma^\star \mid \mathsf{prefixes}(w) \subseteq P\}$*; and*
(infinite traces) $\{u \in \Sigma^\omega \mid \mathsf{prefixes}(u) \subseteq P\}$*; and*
(finite and infinite traces) $\{u \in \Sigma^\star \cup \Sigma^\omega \mid \mathsf{prefixes}(u) \subseteq P\}$.

Let $\mathsf{Safety}^\star_\Box$, $\mathsf{Safety}^\omega_\Box$ and $\mathsf{Safety}^{\star,\omega}_\Box$ be the corresponding sets of properties.

From safety: *In Section 6.3 we show that the language $\mathcal{L}(\Box\varphi)$, that corresponds to the LTL safety formula $\Box\varphi$ for φ some past-time LTL formula, is a property in $\mathsf{Safety}^\omega_\Box$. If one was interested in a finite-trace or in a both finite and infinite trace semantics of LTL, then one could have shown that $\mathcal{L}(\Box\varphi) \in \mathsf{Safety}^\star_\Box$ or that $\mathcal{L}(\Box\varphi) \in \mathsf{Safety}^{\star,\omega}_\Box$.*

Intuitively, one can regard the square "$\Box$" as a closure operator. Technically, it is not precisely a closure operator because it does not operate on the same set: it takes finite-trace properties to any of the three types of properties considered. Since prefixes takes properties back to finite-trace properties, we can show the following result saying that the square is a "closure operator via prefixes", and that safety properties are precisely the sets of words which are closed this way:

Proposition 8 *The following hold for all three types of safety properties:*

- $\Box(\mathsf{prefixes}(\Box P)) = \Box P$ *for any* $P \subseteq \Sigma^\star$*;*
- Q *is a safety property iff* $\Box(\mathsf{prefixes}(Q)) = Q$.

Proof: Left as an exercise to the reader. See Exercise 5. □

We next show that the "always past" properties are all safety properties and, moreover, that any safety property can be expressed as an "always past" property:

Theorem 4 *The following hold:*

- $\mathit{Safety}^{\star}_{\Box} = \mathit{Safety}^{\star}$,
- $\mathit{Safety}^{\omega}_{\Box} = \mathit{Safety}^{\omega}$, *and*
- $\mathit{Safety}^{\star,\omega}_{\Box} = \mathit{Safety}^{\star,\omega}$.

Therefore, each of the "always past" safety properties have the cardinal c.

Proof: We prove each of the equalities by double inclusion.

$\mathsf{Safety}^{\star}_{\Box} \subseteq \mathsf{Safety}^{\star}$. It is true because any property $\Box P$ in $\mathsf{Safety}^{\star}_{\Box}$ is prefix-closed.

$\mathsf{Safety}^{\star} \subseteq \mathsf{Safety}^{\star}_{\Box}$. If $P \in \mathsf{Safety}^{\star}$ then we claim that $P = \Box P$, so $P \in \mathsf{Safety}^{\star}_{\Box}$. Indeed, since P is prefix-closed, $\mathsf{prefixes}(w) \subseteq P$ for any $w \in P$, so $w \in \Box P$; also, since $w \in \mathsf{prefixes}(w)$, it follows that for any $w \in \Box P$, $w \in P$.

$\mathsf{Safety}^{\omega}_{\Box} \subseteq \mathsf{Safety}^{\omega}$. Let $\Box P$ be an "always past" property in $\mathsf{Safety}^{\omega}_{\Box}$, and let u be an infinite trace in Σ^{ω} such that $u \notin \Box P$. Then it follows that $\mathsf{prefixes}(u) \not\subseteq P$, that is, there is some $w \in \mathsf{prefixes}(u)$ such that $w \notin P$. Since $w \in \mathsf{prefixes}(wv)$ for any $v \in \Sigma^{\omega}$, it means that there is no $v \in \Box P$ such that $\mathsf{prefixes}(wv) \subseteq P$, that is, there is no $v \in \Sigma^{\omega}$ such that $wv \in \Box P$. Therefore, $\Box P \in \mathsf{Safety}^{\omega}$.

$\mathsf{Safety}^{\omega} \subseteq \mathsf{Safety}^{\omega}_{\Box}$. If $Q \in \mathsf{Safety}^{\omega}$ then we claim that $Q = \Box\mathsf{prefixes}(Q)$. The inclusion $Q \subseteq \Box\mathsf{prefixes}(Q)$ is clear, because $u \in Q$ implies $\mathsf{prefixes}(u) \subseteq \mathsf{prefixes}(Q)$. For the other inclusion, note that if $\mathsf{prefixes}(u) \subseteq \mathsf{prefixes}(Q)$ for some $u \in \Sigma^{\omega}$, then u must be in Q: if $u \notin Q$ then by the definition of $Q \in \mathsf{Safety}^{\omega}$, there is some $w \in \mathsf{prefixes}(u)$ which cannot be completed into an infinite trace in Q, which contradicts $\mathsf{prefixes}(u) \subseteq \mathsf{prefixes}(Q)$.

$\mathsf{Safety}^{\star,\omega}_{\Box} \subseteq \mathsf{Safety}^{\star,\omega}$. By Proposition 7, it suffices to show that $\mathsf{Safety}^{\star,\omega}_{\Box} \subseteq \mathsf{Safety}^{\star,\omega}_{\mathsf{EM}}$. Let $\Box P$ be an "always past" property in $\mathsf{Safety}^{\star,\omega}_{\Box}$, and let $u \in \Sigma^{\star} \cup \Sigma^{\omega}$ such that $\mathsf{prefixes}(u) \subseteq \mathsf{prefixes}(\Box P)$. Since $\mathsf{prefixes}(\Box P) \subseteq P$, it follows that $u \in \Box P$; therefore, $\Box P \in \mathsf{Safety}^{\star,\omega}_{\mathsf{EM}}$.

$\mathsf{Safety}^{\star,\omega} \subseteq \mathsf{Safety}^{\star,\omega}_{\Box}$. It is straightforward to see that $Q \in \mathsf{Safety}^{\star,\omega}_{\mathsf{EM}}$ implies $Q = \Box\mathsf{prefixes}(Q)$.

The cardinality part follows by Theorems 1, 2, and 3. □

Proposition 8 and Theorem 4 give yet another characterization for safety properties over any of the three combinations of traces, namely one in the style of the equivalent formulation of safety over infinite traces in Proposition 4: Q is a safety property iff it contains precisely the words whose prefixes are in $\mathsf{prefixes}(Q)$.

Exercises

Exercise 2 *The* $\mathsf{prefixes}\colon \mathcal{P}(\Sigma^\star) \to \mathcal{P}(\Sigma^\star)$ *in Definition 1 is a closure operator: it is extensive (*$P \subseteq \mathsf{prefixes}(P)$*), monotone (*$P \subseteq P'$ *implies* $\mathsf{prefixes}(P) \subseteq \mathsf{prefixes}(P')$*), and idempotent (*$\mathsf{prefixes}(\mathsf{prefixes}(P)) = \mathsf{prefixes}(P)$*).*

Exercise 3 *(Counter-)Example 4 showed that the finiteness of* Σ *was necessary in order for Proposition 2 to hold, by defining a property* P *over* $\Sigma = \mathbb{N} \cup \{\infty\}$ *in which all non-empty words start with* ∞*. Can we remove* ∞ *from* Σ *and from all the words in* P*? Why, or why not?*

Exercise 4 *Same like Exercise 3, but for Example 5 instead of Example 4.*

Exercise 5 *Prove Proposition 8.*

Exercise 6 *The "closure under limits" operation in Definition 7 is indeed a closure operator on* Σ^ω*: it is extensive (*$Q \subseteq \overline{Q}$*), monotone (*$Q \subseteq Q'$ *implies* $\overline{Q} \subseteq \overline{Q'}$*), and idempotent (*$\overline{\overline{Q}} = \overline{Q}$*).*

Chapter 4

Monitoring

In this section we give yet another characterization of safety properties, namely as monitorable properties. Specifically, we formally define a monitor as a (possibly infinite) state machine without final states but with a partial transition function, and then we show that safety properties are precisely the properties that can be monitored with such monitors. We then elaborate on the problem of defining the complexity of monitoring a safety property, discussing some pitfalls and guiding principles, and show that monitoring a safety property can be an arbitrarily hard problem. Finally, we give a more compact and mathematical equivalent definition of a monitor, which may be useful in further foundational efforts in this area.

Relate our definition of a monitor with Schneider's security automata

---

4.1 Specifying Safety Properties as Monitors

Safety properties are difficult to work with as flat sets of finite or infinite words, not only because they can contain infinitely many words, but also because such a flat representation is inconvenient for further analysis. It is important therefore to *specify* safety properties using formalisms that are easier to represent and reason about.

From safety: *The next sections in this paper investigate several dedicated formalisms that proved to be convenient in specifying safety, such as finite state machines, regular expressions and temporal logics, together with corresponding limitations and efficient monitor synthesis techniques.*

Formalisms known to be useful for specifying safety properties include regular expressions and temporal logics, which can be efficiently translated into finite-state machines which can then be used as monitors. In this section we formalize the intuitive notion of a *monitor* as a special state machine and give yet another characterization of safety properties, namely as *monitorable properties*. Since monitorable properties are completely defined by their monitors, it follows that *all* safety properties can be specified by their corresponding monitors.

Recall that we work under the assumption that Σ is a set of events or program states such that $|\Sigma| \leq \aleph_0$.

Definition 13 *A* Σ-monitor*, or just a* monitor *(when* Σ *is understood), is a triple* $\mathcal{M} = (S, s_0, M : S \times \Sigma \rightharpoonup S)$*, where* S *is a set of* states*,* $s_0 \in S$ *is the* initial state*, and* M *is a deterministic partial* transition *function.*

Therefore, a monitor as defined above is nothing but a deterministic state machine without final states. Moreover, the set of states is allowed to be infinite, and the transition function has no complexity requirements (it can even be undecidable). We could have defined monitors to be standard state machines, but the subsequent technical developments would have been slightly more involved.

From safety: *In fact, we aim at shortly giving an even more compact definition of a monitor, that we will call* canonical monitor*, which appears to be sufficient to capture any safety property.*

The intuition for a monitor is the expected one: the monitor is driven by events generated by the observed program (the letters in Σ)—each newly received event drives the monitor from its current state to some other state, as indicated by the transition function M; if the monitor ever gets stuck, that is, if the transition function M is undefined on the current state and the current event, then the monitored property is declared violated at that point by the monitor.

For any partial function $M : S \times \Sigma \rightharpoonup S$, we obey the following common notational convention. If $s \in S$ and $w = w_1 w_2 \dots w_k \in \Sigma^\star$, we write "$M(s,w)\downarrow$" whenever $M(s,w)$ is defined, that is, whenever $M(s,w_1)$ and $M(M(s,w_1),w_2)$ and ... and $M(...(M(s,w_1),w_2)...,w_k)$ are all defined, which is nothing but only saying that $M(...(M(s,w_1),w_2)...,w_k)$ is defined. If we write $M(s,w) = s'$ for some $s' \in S$, then, as expected, we mean that $M(...(M(s,w_1),w_2)...,w_k)$ is defined and equal to s'.

A monitor specifies a finite-trace property, an infinite-trace property, as well as a finite- and infinite-trace property:

Definition 14 *Given a monitor $\mathcal{M} = (S, s_0, M : S \times \Sigma \rightharpoonup S)$, we define the following properties:*

- $\mathcal{L}^\star(\mathcal{M}) = \{w \in \Sigma^\star \mid M(s_0,w)\downarrow\}$,
- $\mathcal{L}^\omega(\mathcal{M}) = \{u \in \Sigma^\omega \mid M(s_0,w)\downarrow \text{ for all } w \in \mathsf{prefixes}(u)\}$, *and*
- $\mathcal{L}^{\star,\omega}(\mathcal{M}) = \mathcal{L}^\star(\mathcal{M}) \cup \mathcal{L}^\omega(\mathcal{M})$.

We call $\mathcal{L}^\star(\mathcal{M})$ the finite-trace property specified by $\mathcal{M}$, *call $\mathcal{L}^\omega(\mathcal{M})$ the* infinite-trace property specified by $\mathcal{M}$, *and call $\mathcal{L}^{\star,\omega}(\mathcal{M})$ the* finite- and infinite-trace property specified by $\mathcal{M}$. *Also, we let*

$$\mathcal{S}_\mathcal{M} = \{s \in S \mid (\exists w \in \Sigma^\star)\, M(s_0,w) = s\}$$

denote the set of reachable states *of $\mathcal{M}$.*

A *monitorable* property is a property which can be specified by a monitor. We next capture this intuitive notion formally:

Definition 15 *For a property $P \subseteq \Sigma^\star \cup \Sigma^\omega$, we let $\mathsf{Monitors}(P)$ be the set of monitors $\{\mathcal{M} \mid \mathcal{L}^{\star,\omega}(\mathcal{M}) = P\}$. If $\mathsf{Monitors}(P) \neq \emptyset$ then P is called* monitorable *and the elements of $\mathsf{Monitors}(P)$ are called* monitors of P. *We define the following classes of properties:*

- $\mathsf{Monitorable}^\star = \{P \subseteq \Sigma^\star \mid P \text{ monitorable}\}$,
- $\mathsf{Monitorable}^\omega = \{P \subseteq \Sigma^\omega \mid P \text{ monitorable}\}$, *and*
- $\mathsf{Monitorable}^{\star,\omega} = \{P \subseteq \Sigma^\star \cup \Sigma^\omega \mid P \text{ monitorable}\}$.

The notion of persistence can also be adapted to monitors:

Definition 16 *A monitor* $\mathcal{M} = (S, s_0, M : S \times \Sigma \rightharpoonup S)$ *is* persistent *iff for any reachable state* $s \in \mathcal{S}_{\mathcal{M}}$*, there is an* $a \in \Sigma$ *such that* $M(s, a)\downarrow$*. Let*

- $\mathsf{PersistentMonitorable}^\star = \{\mathcal{L}^\star(\mathcal{M}) \mid \mathcal{M}$ *persistent*$\}$

be the set of finite-trace properties monitorable by persistent monitors.

Our next goal is to show that each monitor admits a largest persistent "submonitor". To formalize it, we lift the conventional partial order relation on partial functions to monitors:

Definition 17 *If* $\mathcal{M}_1 = (S, s_0, M_1 : S \times \Sigma \rightharpoonup S)$ *and* $\mathcal{M}_2 = (S, s_0, M_2 : S \times \Sigma \rightharpoonup S)$ *are two monitors sharing the same states and initial state, then let* $\mathcal{M}_1 \sqsubseteq \mathcal{M}_2$*, read* $\mathcal{M}_1$ *a* submonitor *of* $\mathcal{M}_2$*, iff for any* $s \in S$ *and any* $a \in \Sigma$*, if* $M_1(s, a)$ *is defined then* $M_2(s, a)$ *is also defined and* $M_2(s, a) = M_1(s, a)$*.*

The above can be easily generalized to allow $\mathcal{M}_1$ to only have a subset of the states of $\mathcal{M}_2$, but we found that generalization unnecessary so far.

The above partial-order on monitors allows us to use conventional mathematics to obtain the largest persistent sub-monitor of a monitor:

Proposition 9 $(\{\mathcal{K} \mid \mathcal{K} \sqsubseteq \mathcal{M}$ *and* $\mathcal{K}$ *persistent*$\}, \sqsubseteq)$ *is a complete (join) semilattice for any monitor* $\mathcal{M}$*.*

Proof: If $\{\mathcal{K}_i = (S, s_0, K_i : S \times \Sigma \rightharpoonup S) \in \mathcal{M}\}_{i \in I}$ is a set of persistent monitors, then their supremum (or join) is the monitor $\mathcal{K} = (S, s_0, K : S \times \Sigma \rightharpoonup S)$ where $K(s, a) = s'$ iff there is some $i \in I$ such that $K_i(s, a) = s'$. It is easy to see that $\mathcal{K}$ is a well-defined monitor and that it is persistent. □

Since complete semilattices have maximum elements, the following definition is fully justified:

Definition 18 *For any monitor* $\mathcal{M} = (S, s_0, M : S \times \Sigma \rightharpoonup S)$*, we let* $\mathcal{M}^\circ = (S, s_0, M^\circ : S \times \Sigma \rightharpoonup S)$ *be the* $\sqsubseteq$*-maximal element of the complete lattice* $(\{\mathcal{K} \mid \mathcal{K} \sqsubseteq \mathcal{M}$ *and* $\mathcal{K}$ *persistent*$\}, \sqsubseteq)$*.*

We next show that, as expected, there is a tight relationship between persistent safety properties (Definition 3) and persistent canonical monitors.

Proposition 10 *Let* $\mathcal{M} = (S, s_0, M : S \times \Sigma \rightharpoonup S)$*. Then the following hold:*

- $\mathcal{L}^\omega(\mathcal{M}) = \mathcal{L}^\omega(\mathcal{M}^\circ)$,

- $\mathcal{L}^\star(\mathcal{M}^\circ) = \mathcal{L}^\star(\mathcal{M})^\circ$, *and*
- $\mathcal{M}$ *persistent iff* $\mathcal{L}^\star(\mathcal{M})$ *persistent.*

Proof: The first property can be shown by the following sequence of equivalences: $u \in \mathcal{L}^\omega(\mathcal{M})$ iff $M(s_0, w) \downarrow$ for all $w \in \mathsf{prefixes}(u)$, iff there is some persistent monitor $\mathcal{K} \sqsubseteq \mathcal{M}$ such as $K(s_0, w) \downarrow$ for all $w \in \mathsf{prefixes}(u)$, iff $M^\circ(s_0, w) \downarrow$ for all $w \in \mathsf{prefixes}(u)$, iff $u \in \mathcal{L}^\omega(\mathcal{M}^\circ)$.

The second property can be shown as follows: $w \in \mathcal{L}^\star(\mathcal{M}^\circ)$ iff $M^\circ(s_0, w) \downarrow$, iff there is some $u \in \mathcal{L}^\omega(\mathcal{M}^\circ)$ such that $w \in \mathsf{prefixes}(u)$ (because $\mathcal{M}^\circ$ is persistent), iff there is some $u \in \mathcal{L}^\omega(\mathcal{M})$ such that $w \in \mathsf{prefixes}(u)$ (by the first property), iff there is some $u \in \Sigma^\omega$ such that $w \in \mathsf{prefixes}(u) \subseteq \mathcal{L}^\star(\mathcal{M})$, iff there is some $u \in \Sigma^\omega$ such that $w \in \mathsf{prefixes}(u) \subseteq \mathcal{L}^\star(\mathcal{M})^\circ$ (because $\mathsf{prefixes}(u)$ is a persistent safety property), iff $w \in \mathcal{L}^\star(\mathcal{M})^\circ$.

Finally, the third property is an immediate consequence of the second, noticing that $\mathcal{M}$ is persistent iff it is equal to $\mathcal{M}^\circ$, and that $\mathcal{L}^\star(\mathcal{M})$ is persistent iff it is equal to $\mathcal{L}^\star(\mathcal{M})^\circ$. □

Theorem 5 *The following hold:*

- $\mathit{Monitorable}^\star = \mathit{Safety}^\star$,
- $\mathit{Monitorable}^\omega = \mathit{Safety}^\omega$,
- $\mathit{Monitorable}^{\star,\omega} = \mathit{Safety}^{\star,\omega}$, *and*
- $\mathit{PersistentMonitorable}^\star = \mathit{PersistentSafety}^\star$.

Proof: First, note that the following hold for any monitor $\mathcal{M}$:

- $\mathcal{L}^\star(\mathcal{M}) \in \mathsf{Safety}^\star$,
- $\mathcal{L}^\omega(\mathcal{M}) \in \mathsf{Safety}^\omega$, and
- $\mathcal{L}^{\star,\omega}(\mathcal{M}) \in \mathsf{Safety}^{\star,\omega}$.

These all follow by Theorem 4: taking P in Definition 12 to be the property $\{w \in \Sigma^\star \mid M(s_0, w) \downarrow\}$, then $\Box P$ over finite traces is precisely $\mathcal{L}^\star(\mathcal{M})$, over infinite traces is precisely $\mathcal{L}^\omega(\mathcal{M})$, and over finite and infinite traces is precisely $\mathcal{L}^{\star,\omega}(\mathcal{M})$, so the three languages are in $\mathsf{Safety}^\star_\Box$, $\mathsf{Safety}^\omega_\Box$, and $\mathsf{Safety}^{\star,\omega}_\Box$, respectively. Therefore, $\mathsf{Monitorable}^\star \subseteq \mathsf{Safety}^\star$, $\mathsf{Monitorable}^\omega \subseteq \mathsf{Safety}^\omega$, and $\mathsf{Monitorable}^{\star,\omega} \subseteq \mathsf{Safety}^{\star,\omega}$.

Second, note that we can associate a default monitor $\mathcal{M}_P$ to any finite-trace property $P \subseteq \Sigma^\star$, namely $(S_P, \epsilon, M_P : S_P \times \Sigma \rightharpoonup S_P)$, where $S_P = \mathsf{prefixes}(P)$, ϵ is the empty word, and $M_P(w, a)$ is defined iff $wa \in \mathsf{prefixes}(P)$, and in that case $M_P(w, a) = wa$. Moreover, it is easy to check that

- $\mathcal{L}^\star(\mathcal{M}_P) = \{w \in \Sigma^\star \mid \mathsf{prefixes}(w) \subseteq P\} = \Box P$ (over finite traces) ,
- $\mathcal{L}^\omega(\mathcal{M}_P) = \{u \in \Sigma^\omega \mid \mathsf{prefixes}(u) \subseteq P\} = \Box P$ (over infinite traces),
- $\mathcal{L}^{\star,\omega}(\mathcal{M}_P) = \{u \in \Sigma^\star \cup \Sigma^\omega \mid \mathsf{prefixes}(w) \subseteq P\} = \Box P$ (over both finite and infinite traces).

Since P was chosen arbitrarily, it follows then by Theorem 4 that $\mathsf{Safety}^\star \subseteq \mathsf{Monitorable}^\star$, $\mathsf{Safety}^\omega \subseteq \mathsf{Monitorable}^\omega$, and $\mathsf{Safety}^{\star,\omega} \subseteq \mathsf{Monitorable}^{\star,\omega}$.

Finally, the equality $\mathsf{PersistentMonitorable}^\star = \mathsf{PersistentSafety}^\star$ follows by the first fact and by Proposition 10. □

4.2 Complexity of Monitoring a Safety Property

We here address the problem of defining the complexity of monitoring. Before we give our definition, let us first discuss some pitfalls in defining this notion. Our definition for the complexity of monitoring resulted as a consequence of trying to avoid these pitfalls. Let P be a safety property.

> **Pitfall 1.**
>
> *The complexity of monitoring P is nothing but the complexity of checking, for an input word $w \in \Sigma^\star$, whether $w \in \mathsf{prefixes}(P)$.*

This would be an easy to formulate decision problem, but, unfortunately, does not capture well the intuition of monitoring, because it does not require that the word w be processed incrementally, as its letters become available from the observed system. Incremental processing of letters can make a huge difference in both how complex monitoring is and how monitoring complexity can be defined. For example, it is well-known that the membership problem of a finite word to the language of an extended regular expression (ERE), i.e., a regular expression extended with complement operators, is a polynomial problem (the classic algorithm by Hopcroft and Ullman [27] runs in space $O(m^2 \cdot n)$ and time $O(m^3 \cdot n)$, where m is the size of the word and n that of the expression). However,

---

From safety: *as shown in Section 6.1*

---

there are EREs defining safety properties whose monitoring requires non-elementary space and time. Of course, this non-elementary lower-bound is expressed only as a function of the size of the ERE representing the safety property; it does not take into account the size of the monitored trace. This leads us to our first guiding principle:

Principle 1.

The complexity of monitoring a safety property P should depend only upon P, not upon the trace being monitored.

Indeed, since monitoring is a process that involves potentially unbounded traces, if the complexity of monitoring a property P were expressed as a function of the execution trace as well, then that complexity measure would be close to meaningless in practice, because monitoring reactive systems would have unbounded complexity. For example, consider an operating system monitoring some safety property on how its resources are being used by the various running processes; what one would like to know here is what is the runtime overhead of monitoring that safety property at each relevant event, and not the obvious fact that the more the operating system runs the larger the total runtime overhead is.

Nevertheless, one can admittedly argue that it would still be useful to know how complex the monitoring of P against a given finite trace w is, in terms of both the size of (some representation of) P and the size of w; however, this is nothing but a conventional membership test decision problem, that has nothing to do with monitoring. If one picks some arbitrary off-the-shelf efficient algorithm for membership testing and uses that at each newly received event on the existing finite execution trace, then one may obtain a "monitoring" algorithm whose complexity to process each event grows in time, as events are processed. In the context of monitoring a reactive system, that means that eventually the monitoring process may become unfeasible, regardless of how many resources are initially available and regardless of how efficient the membership testing algorithm is. What one needs in order for the monitoring process to stay feasible regardless of how many events are observed, is a special membership algorithm that processes each event as received and whose state or processing time does not increase potentially unbounded as events are received. Therefore, one needs an algorithm which, if it takes resources R to check w, then it takes at most $R + \Delta$ to check a one-event continuation wa of w, where Δ *does not depend on* w. In other words, one needs a *monitor for* P *of complexity* Δ.

Pitfall 2.

P is typically infinite, so the complexity of monitoring P should be a function of the size of some finite specification, or representation, of P.

Indeed, since Principle 1 tells us that the complexity of monitoring P is a function of P only and not of the monitored trace, one may be tempted to conclude that it is a function of the *size* of some convenient encoding of P. There are at least two problems with this approach, that we discuss below.

- One problem is that the same property P can be specified in many different ways as a structure of finite size; for example, it can be specified as a regular expression, as an extended regular expression, as a temporal logic formula, as an ordinary automaton, as a push-down automaton, etc. These formalisms may represent P as specifications of quite different sizes. Which is the most appropriate? It is, nevertheless, interesting and important to study the complexity of monitoring safety properties expressed using different specification formalisms, as a function of the property representation size, because that can give us an idea of the amount of resources needed to monitor a particular specification. However, one should be aware that such a complexity measure is an attribute of the corresponding specification formalisms, not of the specified property itself. Indeed, the higher this complexity measure for a particular formalism, the higher the encoding strength of safety properties in that formalism: for example, the complexity of monitoring safety properties expressed as EREs is non-elementary in the size of the original ERE, while the complexity of monitoring the same property expressed as an ordinary regular expression is linear in the size of the regular expression. Does that mean that one can monitor safety properties expressed as regular expressions non-elementarily more efficiently than one can monitor safety properties expressed as EREs? Of course not, because EREs and regular expressions have the same expressiveness, so they specify exactly the same safety properties. All it means is that EREs can express safety-properties non-elementarily more compactly than ordinary regular expressions.

- Another problem with this approach is that apparently appropriate representations of P may be significantly larger than it takes to monitor P. One may say, for example, that, whenever possible, a natural way to specify a particular safety property is as a finite-state machine, e.g.,

as a monitor like in Definition 13 . To be more concrete, consider that the safety property P_n saying "every 2^n-th event is a" is specified as a monitor of 2^n states that transits with any event from each state to the next one, except for the 2^n-th state, which has only one transition, with event a, back to state 1. Therefore, the size of this representation of P_n is $\Omega(2^n)$. Assuming that each state takes n bits of storage (for example, assume that states are exactly the binary encodings of the numbers 1, 2, 3, ..., 2^n) and that the next state can be calculated from the current state in linear complexity with the size of the state (which is true in our scenario), then it is clear that the actual complexity of monitoring P_n is $O(n)$. If the complexity of monitoring P_n were a function of the size of the specification of P_n, then one could wrongly conclude that the complexity of monitoring "every 2^n-th event is a" is $O(2^n)$.

Therefore, a safety property P has an inherent complexity w.r.t. monitoring, complexity which has nothing to do with how P is represented, or encoded, or specified. It is that inherent complexity attribute of safety properties that we are after here. From the discussion above, we draw our second guiding principle:

Principle 2.

The monitoring complexity of a safety property P is an attribute of P alone, not a function of the size of some adhoc representation of P.

By Theorem 5, safety properties are precisely those properties that are monitorable, that is, those properties P for which there are (finite-state or not) monitors $\mathcal{M} = (S, s_0, M : S \times \Sigma \rightharpoonup S)$ whose (finite-trace, infinite-trace, or finite- and infinite-trace—this depends upon the type of P) language is precisely P. Any algorithm, program or system that one may come up with to be used as a monitor for P, can be organized as a monitor of the form $\mathcal{M} = (S, s_0, M : S \times \Sigma \rightharpoonup S)$ for P. Consequently, the complexity of monitoring P cannot be smaller than the functional complexity of the partial function $M : S \times \Sigma \rightharpoonup S$) corresponding to some "best" monitor $\mathcal{M}$ for P; if there are no additional restrictions, then by "best" monitor we mean the one whose functional complexity of M is smallest. In particular, if there is no monitor for P whose transition partial function M is decidable, then we can safety say that the problem of monitoring P is undecidable. This discussion leads to the following:

Pitfall 3.

The complexity of monitoring P is the functional complexity of function M, where $\mathcal{M} = (S, s_0, M : S \times \Sigma \rightharpoonup S)$ is the "best" monitor for P.

Since safety properties are precisely the monitorable properties, this appears to be a very natural definition for the complexity of monitoring. While the functional complexity of the monitor function is indeed important because it directly influences the efficiency of monitoring, it is *not* a sufficient measure for the complexity of monitoring. That is because the functional complexity of M only says how complex M is in terms of the size of *its input*; it does not say anything about how large the state of the monitor can grow in time. For example, the rewriting-based monitoring algorithm for EREs from [44], whose states are EREs and whose transition is a derivative operation of functional complexity $O(n^2)$ taking an ERE of size n into an ERE of size $O(n^2)$. It would be very misleading to say that the complexity of monitoring EREs is $O(n^2)$, because it may sound much better than it actually is: the n^2 factor accumulates as events are processed. Any monitor for EREs, including the one based on derivatives, eventually requires non-elementary resources (in the size of the ERE) to process a new event.

Therefore, while the complexity of the function M being executed at each newly received event by a monitor $\mathcal{M}$ is definitely a necessary and important factor to be considered when defining the complexity of monitoring using $\mathcal{M}$, it is not sufficient. One also needs to take into account the size of the input that is being passed to the monitoring function, that is, the size of the monitor state together with the size of the received event. In particular, a monitor storing all the observed trace has unbounded complexity, say ∞, even though its monitoring function has trivial complexity (e.g., the "event storing" function has linear complexity). More generally, if a property admits no finite-state monitor, than we'd like to say that its monitoring complexity is ∞: indeed, for any monitor for such a property and for any amount of resources R, there is some sequence of events that would lead the monitor to a state that needs more than R resources to be stored or computed. These observations lead us to the following:

Principle 3. The complexity of monitoring P is a function of both the functional complexity of M and of the size of the states in S, where $\mathcal{M} = (S, s_0, M : S \times \Sigma \rightharpoonup S)$ is an appropriately chosen ("best") monitor for P.

We next follow the three principles above and derive our definition for the complexity of monitoring a safety property P. Before that, let us first define the complexity of monitoring a safety property using a particular monitor for that property, or in other words, let us first define the complexity of a monitor.

During a monitoring session using a monitor, at any moment in time one needs to store at least one state, namely the state that the monitor is currently in. When receiving a new event, the monitor launches its transition function on the current state and the received input. Therefore, the (worst-case) complexity of monitoring with $\mathcal{M} = (S, s_0, M : S \times \Sigma \rightharpoonup S)$ could be defined as

$$\max\{FC(M(s,a)) \mid s \in S, a \in \Sigma\},$$

where $FC(M(s,a))$ is the functional complexity of evaluating M on state s and event a, as a function of the sizes of s and a. In other words, the worst-case monitoring complexity of a particular monitor is the maximal functional complexity that its transition function has on any state and any input; this functional complexity is expressed as a function of the size of the pair (state,event). In order for such a definition to make sense formally, one would need to define or axiomatize the size of monitor states and the size of events. Since in order to distinguish N elements one needs $\log(N)$ space, we deduce that one needs at least $\log(|S|)$ space to store the state of the monitor in its worst-case monitoring scenario (each state in S is reachable).

Definition 19 *Given a monitor $\mathcal{M} = (S, s_0, M : S \times \Sigma \rightharpoonup S)$, we define the complexity of monitoring $\mathcal{M}$, written $C_{Mon}(\mathcal{M})$, as the function*

$$FC(M)(\log|S|) : \mathbb{N} \to \mathbb{N},$$

which is the "uncurried" version applied on $\log|S|$ of the worst-case functional complexity $FC(M) : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$ of the partial function M as a function of the size of the pair (state,event) being passed to it.

We assume that the complexity of monitoring a safety property P is the worst-case complexity of monitoring it using some appropriate, "best" monitor for P:

$$\min\{\max\{FC(M(s,a)) \mid s \in S, a \in \Sigma\} \mid \mathcal{M} = (S, s_0, M) \in \mathsf{Monitors}(P)\},$$

> **From safety:** *where $FC(M(s,a))$ is the functional complexity of evaluating M on state s and event a, as a function of the sizes of s and a. In other words, we assume that the complexity of monitoring a safety property P is the worst-case complexity of monitoring it using some appropriate, "best" monitor for P. The worst-case monitoring complexity of a particular monitor is the maximal functional complexity that its transition function has on any state and any input; this functional complexity is expressed as a function of the size of the pair (state,event). Therefore, in order for such a definition to make sense formally, one would need to define or axiomatize the size of monitor states and the size of events.*

This gives us the following:

Definition 20 *We let*

$$C_{Mon}(P) = \min\{FC(M) \circ \langle \log(|S|), 1_\Sigma \rangle \mid \mathcal{M} = (S, s_0, M) \in \mathsf{Monitors}(P)\}$$

be the complexity of monitoring a safety property P.

4.3 Monitoring Safety Properties is Arbitrarily Hard

We show that the problem of monitoring a safety property can be arbitrarily complex. The previous section tells us that there are as many safety properties as real numbers. Therefore, it is not surprising that some of them can be very hard or impossible to monitor. In this section we formalize this intuitive argument. Our approach is to show that we can associate a safety property P_S to any set of natural numbers S, such that monitoring that safety property is as hard as checking membership of arbitrary natural numbers to S. The result then follows from the fact that checking memberships of natural numbers to sets of natural numbers is a problem that can be arbitrarily complex.

Theorem 1 indirectly says that we can associate a persistent safety property to any set of natural numbers (sets of natural numbers are in a bijective correspondence with the real numbers). However, it is not clear how that safety property looks and neither how to monitor it. We next give a more concrete mapping from sets of natural numbers to (persistent) safety properties and show that monitoring the property is equivalent to

testing membership to the set. It suffices to assume that Σ contains only two elements, say $\Sigma = \{0, 1\}$.

Definition 21 *Let $P\_ : \mathcal{P}(\mathbb{N}) \to \mathsf{PersistentSafety}^\star$ be the mapping defined as follows: for any $S \subseteq \mathbb{N}$, let P_S be the set $1^\star \cup \{1^k 0 \mid k \in S\} \cdot \{0, 1\}^\star$.*

It is easy to see that P_S is a persistent safety property over finite traces. Also, it is easy to see that the bijection in the proof of Theorem 2 associates to P_S the safety property over infinite traces $1^\omega \cup \{1^k 0 \mid k \in S\} \cdot \{0, 1\}^\omega$.

Let us now investigate the problem of monitoring P_S.

Proposition 11 *For any $S \subseteq \mathbb{N}$, monitoring P_S is equivalent to deciding membership of natural numbers to S.*

Proof: If M_S is an oracle deciding membership of natural numbers to S, that is, if $M_S(n)$ is true iff $n \in S$, then one can build a monitor for P_S as follows: for a given trace, incrementally read and count the number of prefix 1's; if no 0 is ever observed then monitor indefinitely without reporting any violation; when a first 0 is observed, if any, ask if $M(k)$, where k is the number of 1's observed; if $M(k)$ is false, then report violation; if $M(k)$ is true, then continue monitoring indefinitely and never report violation. It is clear that this is indeed a monitor for P_S.

Conversely, if we had any monitor for P_S then we could build a decision procedure for membership to S as follows: given $k \in \mathbb{N}$, send to the monitor a sequence of k ones followed by a 0; if the monitor reports violation then deduce that $k \notin S$; if the monitor does not report violation, then deduce that $k \in S$. It is clear that this is a decision procedure for membership to S.

The proof works for both persistent safety properties over finite traces and for safety properties over infinite traces. □

The claim in the title of this section follows now from the fact that the set S of natural numbers can be chosen so that its membership problem is arbitrarily complex. For example, since there are as many subsets of natural numbers as real numbers while there are only as many Turing machines as natural numbers, it follows that there are many (exponentially) more sets of natural numbers that are not recognized by Turing machines than those that are. In particular, there are sets of natural numbers corresponding to any degree in the arithmetic hierarchy, i.e., to predicates $A(k)$ of the form $(Q_1 k_1)(Q_2 k_2) \cdots (Q_n k_n)\ R(k, k_1, k_2, \cdots, k_n)$, where Q_1, Q_2, ..., Q_n are alternating (universal or existential) quantifiers and R is a recursive/decidable

relation: for A such a predicate, let S_A be the set of natural numbers $\{k \mid A(k)\}$. Recall that if Q_1 is $\forall$ then A is called a Π_n property, while if Q_1 is $\exists$ then A is called a Σ_n property. In particular, $\Sigma_0 = \Pi_0$ and they contain precisely the recursive/decidable properties, Σ_1 contains precisely the class of recursively enumerable problems, Π_1 contains precisely the co-recursively enumerable problems, etc.; a standard Π_2 problem is TOTALITY: given $k \in \mathbb{N}$, is it true that Turing machine with Gödel number k terminates on all inputs? Since each level in the arithmetic hierarchy contains problems strictly harder than problems on the previous layer (because $\Sigma_n \cup \Pi_n \subsetneq \Sigma_{n+1} \cap \Pi_{n+1}$), the arithmetic hierarchy gives us a universe of safety properties whose monitoring can be arbitrarily hard.

Within the decidable fragment, as expected, monitoring safety properties can also have any complexity. Indeed, pick for example any NP-complete problem and let S be the set of inputs (coded as natural numbers) for which the problem has a positive answer; then, as explained in the proof of Proposition 11, monitoring P_S against input 1^k0 is equivalent to deciding membership of k to S, which is further equivalent to answering the NP-complete problem on input k. Of course, in practice a particular (implementation of a) monitor can be more complex than the corresponding membership problem; for example, monitors corresponding to NP-complete problems are most likely exponential. Also, note that a monitor for P_S needs not necessarily do its complex computation on an input 1^k0 when it encounters the 0. It can perform intermediate computations as it reads the prefix 1's and thus pay a lesser computational price when the 0 is encountered. What Proposition 11 says is that the *total* complexity to process the input 1^k0 can be no lower than the complexity of checking whether $k \in S$.

4.4 Canonical Monitors

We conclude this section with an alternative definition of a monitor, called *canonical monitor*, which is more compact than our previous definition and which appears to be sufficient to capture any safety property. We do not make any use of this alternative definition in this paper, but it may serve as a basis for further foundational endeavors in this area.

The set of states S of a monitor $(S, s_0, M : S \times \Sigma \rightharpoonup S)$ are typically enumerable, so they can be very well replaced with natural numbers. Moreover, the initial state s_0 can be encoded, by convention, as the first natural number, 0. A monitor then becomes nothing but a partial function $\mathbb{N} \times \Sigma \rightharpoonup \mathbb{N}$. We

therefore rightfully call these particular monitors *canonical*:

Definition 22 *A* canonical Σ-monitor *is a partial function* $\mathcal{N} : \mathbb{N} \times \Sigma \rightharpoonup \mathbb{N}$. *Let* $\mathcal{S}_{\mathcal{N}} = \{n \mid (\exists w)\, \mathcal{N}(0, w) = n\}$ *be the* states *of* $\mathcal{N}$. *As before, let*

- $\mathcal{L}^{\star}(\mathcal{N}) = \{w \in \Sigma^{\star} \mid \mathcal{N}(0, w) \downarrow\}$,
- $\mathcal{L}^{\omega}(\mathcal{N}) = \{u \in \Sigma^{\omega} \mid \mathcal{N}(0, w) \downarrow \text{ for all } w \in \mathsf{prefixes}(u)\}$, *and*
- $\mathcal{L}^{\star,\omega}(\mathcal{N}) = \mathcal{L}^{\star}(\mathcal{N}) \cup \mathcal{L}^{\omega}(\mathcal{N})$.

Although the set of states S in a monitor $(S, s_0, M : S \times \Sigma \rightharpoonup S)$ is allowed to have any cardinal while the states in canonical monitors are restricted to natural numbers, it turns out that canonical monitors can in fact express all monitorable properties:

Proposition 12 *A property* $P \subseteq \Sigma^{\star}$ *(resp.* $P \subseteq \Sigma^{\omega}$, *resp.* $P \in \Sigma^{\star} \cup \Sigma^{\omega}$*) is monitorable iff there is some canonical monitor* $\mathcal{N}$ *such that* $P = \mathcal{L}^{\star}(\mathcal{N})$ *(resp.* $P = \mathcal{L}^{\omega}(\mathcal{N})$, *resp.* $P = \mathcal{L}^{\star,\omega}(\mathcal{N})$*).*

Proof: Since any canonical monitor is a monitor, it follows that any property specifiable by a canonical monitor is indeed monitrable. For the converse, let P be a property monitorable by some monitor $\mathcal{M} = (S, s_0, M : S \times \Sigma \rightharpoonup S)$. Since $|\Sigma| \leq \aleph_0$, we can enumerate all the states of $\mathcal{M}$ that can be reached from s_0 with its transition function M. There are many different ways to do this (e.g., in breadth-first order, in depth-first order, etc.), but these are all ultimately irrelevant. If we let $S^r = \{s_0, s_1, s_2, \ldots\}$ denote the resulting set of reachable states, then it is easy to first note that the monitor $\mathcal{M}^r = (S^r, s_0, M : S^r \times \Sigma \rightharpoonup S^r)$ specifies the same property P as $\mathcal{M}$, and second note that $\mathcal{M}^r$ specifies the same property as the canonical monitor $\mathcal{N} : \mathbb{N} \times \Sigma \rightharpoonup \mathbb{N}$ defined by $\mathcal{N}(i, a) = j$ iff $M(s_i, a) = s_j$. □

Exercises

Exercise 7 *Define a canonical monitor for the property*

> *"A file can only be accessed if it is open."*

That is, the file can only be accessed if it was opened at some moment in the past and it was not closed since then. Suppose Σ *consists of the events/actions* $\{o, a, c\}$, *where* o *stands for "file open",* a *for "file access", and* c *for "file close".*

Chapter 5

Event/Trace Observation

Chapter 6

Monitor Synthesis

6.1 Extended Regular Expressions (EREs)

6.1.1 Monitoring ERE Safety Needs Non-Elementary Space

Extended regular expressions (EREs) add complementation ($\neg$) to regular expressions (REs). Complementation can be handy when defining safety properties, because it allows one to say both what should never happen as well as what could happen. In particular, in the context of monitoring EREs, one can switch between the expression of bad prefixes of a safety property and that of good prefixes by just applying a complement operator.

In this section we show that any monitor for safety properties expressed as EREs requires non-elementary space. More precisely, for a given $n \in \mathbb{N}$, we build an ERE of size $O(n^3)$ such that its language is prefix closed and any monitor for its language requires $2^{2^{\cdot^{\cdot^{2^n}}}}$ space, with n nested power of 2 operations.

Discussion and Relevance of the Lower-Bound Result

Since regular expressions (REs) and deterministic (DFA) and non-deterministic (NFA) finite-state automata are enumerable structures while the set of safety properties is in bijection with the set of real numbers, there are many safety properties that cannot be expressed using REs or automata (or any other formalism whose objects are enumerable). Nevertheless, there are many safety properties of interest that can be expressed as REs or automata. Safety properties over finite or infinite traces can be expressed as REs in at

least two different ways:

1. Use an RE to express the language of its bad prefixes; or
2. Use an RE to express the language of its good prefixes.

In the first case, the RE captures the finite-trace behaviors that should never happen, while in the second the RE captures the ones that could possibly happen.

Obviously, not all REs correspond to safety properties in one or the other of the two cases above. For example, the RE $(0 \cdot 1)^+$ cannot express the bad prefixes of a safety property, because "01" is a bad prefix while "010" is not. In order for an RE to express the bad prefixes of a safety property, it should have the property that once w is in its language, then all ww' for any w' should also be in its language. The RE $(0 \cdot 1)^+$ cannot express the good prefixes of a safety property either: "0101" is a good prefix while 010 is not. The language of an RE must be prefix closed in order to express the good prefixes of a safety property.

In both cases above, monitoring the safety property can be done very efficiently (linearly in the size of the RE, both space-wise and time-wise) by first translating its corresponding RE into an NFA, for example using a technique such as Thompson's [54], and then doing one of the following:

1. In the first case, simulate the NFA-to-DFA construction on the fly as events are received. The state of the monitor is therefore a set of states of the NFA. At start, that set contains only the initial state of the NFA. For each new event, construct the next set by collecting all the NFA states that can be reached via the received event from any of the existing states in the set. If a final state is reached then report violation of the property: bad prefix found. Since the final states in the NFA symbolize the reach of a bad prefix and since bad prefixes have "no future" in a safety property, the NFA associated to the original RA can be optimized (in case it is not already optimal directly from its construction) by removing any edges out of its final states.
2. In the second case, the monitor works the same way as in the first case, but checking at each time that the monitor state (also a set of NFA states) contains at least one final state of the NFA; if that is not the case, then report violation: prefix found which is not good. If one is willing to pay the exponential price and determinize the NFA of good prefixes, then one can further optimize the resulting DFA (in case it is

not already optimized by construction) by collapsing all its non-final states into a "dead-end" state: indeed, the reachability of a non-final state signifies the reachability of a bad prefix, which has "no future". It is not clear whether or how to optimize the NFA of good prefixes using the additional info that it is a safety property.

Extended regular expressions (EREs) add complementation ($\neg$) to REs. Meyer and Stockmeyer [52, 53] showed that EREs can express languages non-elementarily more compactly than REs. In other words, for any constant $k \geq 1$, one can find EREs of large enough size $n \in \mathbb{N}$ for which there is no RE having the same language of size less than $2^{2^{\cdot^{\cdot^{2^n}}}}$, with k nested power operations. Meyer and Stockmeyer [52, 53] showed that several other problems concerning EREs are also non-elementary, including: the equivalence of EREs, the emptiness of the language of an ERE (and implicitly the emptiness of the complement of the language of an ERE), the automata generation (NFA or DFA) from an ERE, etc. Note that it is straightforward to generate potentially non-elementarily large automata from EREs. All one needs to do is to iteratively apply NFA-to-DFA transformations for EREs under complement operators and then complement the resulting DFAs (by complementing their final states). Since each NFA-to-DFA transformation may lead to an exponential explosion on the number of states and since the ERE can have arbitrarily many nested complement operators, the resulting NFA or DFA can be non-elementarily larger than the original ERE.

As already mentioned above, if we allow complementation then we can easily switch from an expression defining the bad prefixes of a safety property to one defining its good prefixes, and backwards, by applying a complement operator. Therefore, from here on, when we say that an ERE expresses a safety property, without any loss of generality we assume that it defines the good prefixes of the safety property; in particular, we assume that its language is prefix closed. Clearly, if one can afford to generate an automaton from the ERE expressing a safety property, then one can and probably should use that automaton as a monitor for the safety property. However, since such an automaton can be enormous compared to the the size of the original ERE, a natural question to ask is whether one can generate monitors for safety properties expressed as EREs that need less than non-elementary space in the size of the original ERE. We next give a negative answer to this question.

Let us first discuss our subsequent lower bound result from a more conceptual perspective. Notice that *synchronous monitoring*, that is, the

monitoring process where an error is detected as soon as it appears, is harder than checking for satisfiability (or non-emptiness); indeed, if a formula in a particular formalism is not satisfiable (or it has an empty language), then a synchronous monitor should detect that before the first event is observed. We refer the interested reader to [45] for a discussion on various types of monitoring, including synchronous versus asynchronous monitoring. Synchronous monitors need to either directly (e.g., by accumulating logical constraints while verifying their consistency) or indirectly (e.g., by generating statically an automaton or a structure containing all possible future behaviors) check for satisfiability (or non-emptiness) of the remaining requirements as events are observed online. Unfortunately, this is an expensive process that may be desired to be avoided, at the expense of delaying the detection of violations. For example, the rewriting based monitoring approach for LTL in [45] delays the detection of violations of LTL formulae of the form "(next φ) and (next $\neg\varphi$)" for one more event, to avoid invoking an expensive satisfiability checker for LTL but to instead invoke a propositional satisfiability checker which is less expensive in practice; this is closely related to the notion of "informative prefixes" in [35] that "tell all the story". Our subsequent lower-bound result states that any monitor for safety properties expressed using EREs, *synchronous or asynchronous*, requires non-elementary space.

Let us now clarify that our lower bound result is not a consequence of the lower-bound result by Meyer and Stockmeyer in [52] (see [53] for a proof of that result). A first reason is that neither the ERE constructed in [53, 52] nor its complement is prefix closed. Indeed, we here focus on a subset of EREs, rather than arbitrary EREs, namely those whose languages are prefix closed, so they express good prefixes of safety properties. Supposing that one could modify the "hard" ERE in [53, 52] whose complement non-emptiness requires non-elementary space into one whose language is prefix-closed and whose size is linear in the size of the original one, the fact that synchronous monitoring of EREs is harder than checking for emptiness does not necessarily imply that monitoring that hard ERE requires non-elementary space. In fact, monitoring that particular ERE requires constant time to process each event, because it is equivalent with an automaton of one state – it takes, however, non-elementary space to compute that one state automaton. What it says is therefore that the *initialization step of ERE-safety synchronous monitoring* requires, in the worst case, non-elementary space.

Interestingly, if one could modify the results in [53, 52] to hold for prefix-closed EREs, including especially the result stating that finding an RE

equivalent to an ERE is a non-elementary problem, then one could show that *synchronous monitoring* of safety properties expressed as EREs requires non-elementary space! Indeed, supposing that one had for any ERE a monitor that takes only elementary space in its worst-case monitoring scenario, then one could use that monitor to generate a DFA for the ERE as follows: start with the initial state of the monitor and then discover and store new states of the monitor by feeding arbitrary (but finite in number) events to each state of the monitor until no new state is discovered. This closure operation takes as much time and space as the number of states the monitor reach; since by assumption the monitor needs "only" elementary space to store its state in any scenario, we deduce that the obtained automaton has size elementary in the size of the ERE (and it also takes elementary time and space to generate it). In other words, we could find an elementary algorithm to associate to any (prefix closed) ERE an equivalent RE, contradicting the non-elementary lower bound in [53, 52] (again, supposing that the lower-bound results in [53, 52] could be modified for prefix-closed EREs).

Unfortunately, it is not that clear how to reduce the non-elementary problems in [53, 52] to *asynchronous monitoring*, and thus to conclude that asynchronous monitoring also requires non-elementary space. That is because a "smart" asynchronous monitor may in principle collapse states (when regarded as an automaton as in the construction above) in rather unexpected ways, just because it "knows" that eventually an error may be reported anyway if observation continues indefinitely from that state on; in other words, states with the property that "eventually violation detected in the future" may be collapsed as equivalent by an asynchronous monitor. This way, the DFA extracted from a monitor for a safety property expressed as an ERE may be significantly smaller than the DFA corresponding to the ERE (and obviously, it may have a different language).

Our next result shows that asynchronous monitoring of ERE-safety also requires non-elementary space, which is a more general lower-bound result than the space non-elementarity of synchronous monitoring. Moreover, it gives an alternative proof of the non-elementary lower-bounds by Stockmeyer and Meyer [53, 52], because automata corresponding to safety-defining EREs are just special cases of monitors, so they must also take non-elementary space. Moreover, we improve the results in [53, 52] by showing that their lower-bounds also hold for a *subset of EREs*, namely those corresponding to safety properties.

Summarizing the discussion above, we believe that the main contributions

of our subsequent lower-bound result are the following:

1. We show that asynchronous monitoring already requires non-elementary space, same as synchronous monitoring. For example, an ERE monitoring algorithm was presented by Roşu and Viswanathan in [44], which "rewrites" or "derives" the ERE by each letter in the input word; the derivation process consists of some straightforward rewrite rules, some for expanding the ERE others for contracting it via simplifications. No comprehensive and expensive check for emptiness on the resulting ERE is performed, except what is done by the simplification rules (for example $\emptyset \cdot R \to \emptyset$ and $\epsilon \cdot R \to R$, etc.). A check for emptiness can and should be eventually performed (for example, a check for emptiness can be done periodically, say every 10^x events for some convenient x). This gives us an asynchronous ERE monitoring algorithm, which, unlike the simplistic NFA/DFA generation algorithm, does not pay upfront the potentially non-elementary worst-case penalty! However, our subsequent lower bound result tells that there is a worst-case scenario in which one cannot avoid the non-elementary space required to store the continuously changing ERE if one wants to correctly eventually detect violations of the original ERE. And that is the case for any synchronous or asynchronous monitoring algorithm for safety properties expressed as EREs.

2. We propose a different technique to prove non-elementary lower-bounds, fundamentally different from the one in [53]. The technique in [53] is based on diagonalization arguments and encodings of accepting Turing machine computations as finite trace words. Our technique is inspired from an idea by Chandra, Kozen and Stockmeyer [9] introduced to show the power of alternation and then used by several authors to prove exponential lower bounds [32, 33, 44, 45]. At our knowledge, the use of such a technique to show non-elementary lower bounds is novel. The idea of the technique in [9] is to define expressions having as languages $L_n = \{w^{(1)}\#w^{(2)}\#\cdots\#w^{(k)}\$w \mid w^{(1)}, w^{(2)}, ..., w^{(k)}, w \in \{0,1\}^n,\ (\exists 1 \leq i \leq k)\ w^{(i)} = w\}$. Our idea is to define, using EREs, words of the form $X_n\$X'_n$, where X_n and X'_n are *n-deep nested sets* starting with elements in $\{0,1\}^n$ (i.e., sets of sets of ... of sets of elements in $\{0,1\}^n$, with n power set operations), such that X'_n is *n-nested included* in X_n, where i-nested inclusion is standard inclusion when $i = 1$ and, if $i > 1$, then it is defined inductively as: X'_i is i-nested

included in X_i iff for each $X'_{i-1} \in X'_i$, there is some $X_{i-1} \in X_i$ such that X'_{i-1} is $(i-1)$-nested included in X_{i-1}.

One more observation is in place before we move on to the technical details. It is known that the *membership problem* for EREs, testing whether a word w of size m is in the language of an ERE of size n, is polynomial in m and n. For example, the classic algorithm by Hopcroft and Ullman [27] runs in space $O(m^2 \cdot n)$ and time $O(m^3 \cdot n)$; slightly improved algorithms have been proposed by several authors [25, 55, 56, 57, 34, 28], reducing space requirements to $O(m^2 \cdot k + m \cdot n)$ and time to $O(m^3 \cdot k + m^2 \cdot n)$ or worse, where k is the number of complement operators in the ERE; a recent ERE membership algorithm was proposed by the author in [47], which runs in space $O(m \cdot \log m \cdot 2^n)$ and time $O(m^2 \cdot \log m \cdot 2^n)$ when $m > 2^n$. These algorithms appear to be efficient, because they are polynomial or simply exponential in the ERE, so one may think that one may device an elementary ERE-safety monitoring algorithm as follows: store the trace of events and at each newly received event invoke one of these "efficient" ERE membership algorithms. While this algorithm will indeed be elementary in the size of the ERE *and* the size of the trace, our lower bound result says that it will, in fact, be *non-elementary in the size of only the ERE*! In other words, for a carefully chosen "hard" ERE of size n, there are finite traces of large enough size m so that checking their membership is a problem which is non-elementary in n; this will indeed happen when m is non-elementarily larger than n.

The Lower-Bound Result

EREs define languages by inductively applying *union* ($+$), *concatenation* ($\cdot$), *Kleene Closure* ($\star$), *intersection* ($\cap$), and *complementation* ($\neg$). The language of an ERE R, say $\mathcal{L}(R)$, is defined inductively as follows, where $s \in \Sigma$:

- $\mathcal{L}(\emptyset) = \emptyset$,
- $\mathcal{L}(\epsilon) = \{\epsilon\}$,
- $\mathcal{L}(s) = \{s\}$,
- $\mathcal{L}(R_1 + R_2) = \mathcal{L}(R_1) \cup \mathcal{L}(R_2)$,
- $\mathcal{L}(R_1 \cdot R_2) = \mathcal{L}(R_1) \cdot \mathcal{L}(R_2)$,

- $\mathcal{L}(R^\star) = (\mathcal{L}(R))^\star$,
- $\mathcal{L}(R_1 \cap R_2) = \mathcal{L}(R_1) \cap \mathcal{L}(R_2)$,
- $\mathcal{L}(\neg R) = \neg\mathcal{L}(R)$.

If R does not contain $\neg$ and $\cap$ then it is a *regular expression* (RE). By applying De Morgan's law $R_1 \cap R_2 \equiv \neg(\neg R_1 + \neg R_2)$, EREs can be linearly (in both time and size) translated into equivalent EREs without intersection; therefore, intersection of EREs is just syntactic sugar. The *size* of an ERE is the total number of occurrences of letters and composition operators ($+$, $\cdot$, $\star$, and $\neg$) that it contains. In what follows we assume that Σ is finite. For notational simplicity, in what follows we let Σ also denote the RE $s_1 + s_2 + \cdots + s_n$ where $\Sigma = \{s_1, s_2, \ldots, s_n\}$ and let $\Sigma^\star$ denote both the language $\{s_1, s_2, \ldots, s_n\}^\star$ and the RE $(s_1 + s_2 + \cdots + s_n)^\star$.

For $n \in \mathbb{N}$, let us define inductively the following alphabets and languages:

- $\Sigma_0 = \{0, 1\}$ and $\Psi_0 = \{0, 1\}^n$, and
- $\Sigma_i = \Sigma_{i-1} \cup \{\#_i\}$ and $\Psi_i = \{\#_i\#_i\} \cup (\{\#_i\} \cdot \Psi_{i-1})^+ \cdot \{\#_i\}$, for all $1 \leq i \leq n$.

In the above, $\#_i$ are n fresh letters. The intuition for the languages Ψ_i above is to encode nested sets of depth i that contain sets of words of n bits at their deepest level. The symbols $\#_i$ play the role of markers separating the elements of such sets. For example, the word $\#_2\#_1\#_1\#_2\#_101\#_110\#_1\#_2\#_1\#_1\#_2$ encodes $\{\{\}, \{01, 10\}, \{\}\}$, that is, the set $\{\{\}, \{01, 10\}\}$; since the multiplicity and order of elements in sets are irrelevant, the same set can have (infinitely) many different encodings. Formally, let us define the following *set* functions associating to encodings in Ψ_i corresponding nested sets:

- $set_0 : \Psi_0 \to \{0, 1\}^n$ is the identity function on Ψ_0, i.e., $set_0(w) = w$;
- $set_i : \Psi_i \to \mathcal{P}^i(\{0, 1\}^n)$, where $\mathcal{P}^i$ is the power set operator applied i times, $set_i(\#_i\#_i) = \{\}$, $set_i(\#_i X_{i-1}\#_i) = \{set_{i-1}(X_{i-1})\}$, and $set_i(\#_i X_{i-1} X_i) = \{set_{i-1}(X_{i-1})\} \cup set_i(X_i)$, for all $1 \leq i \leq n$, $X_{i-1} \in \Psi_{i-1}$, and $X_i \in \Psi_i$.

Note that $|set_0(\Psi_0)| = 2^n$ and $set_i(\Psi_i) = \mathcal{P}(set_{i-1}(\Psi_{i-1}))$ for all $1 \leq i \leq n$; therefore, $|set_i(\Psi_i)| = 2^{2^{\cdot^{\cdot^{2^n}}}}$ for all $1 \leq i \leq n$, with $i + 1$ nested power operations.

Let us define *nested-inclusion* relations $\_ \Subset_i \_ : \mathcal{P}^i(\{0,1\}^n) \times \mathcal{P}^i(\{0,1\}^n)$ for $0 \leq i \leq n$ and *nested-membership* relations $\_ \Subset_i \_ : \mathcal{P}^{i-1}(\{0,1\}^n) \times \mathcal{P}^i(\{0,1\}^n)$ for $1 \leq i \leq n$ as follows:

- $\_ \Subset_0 \_$ is the identity on $\{0,1\}^n$ and $\_ \Subset_1 \_$ is $\_ \subseteq \_ : \mathcal{P}(\{0,1\}^n) \times \mathcal{P}(\{0,1\}^n)$,
- $\_ \Subset_1 \_$ is $\_ \in \_ : \{0,1\}^n \times \mathcal{P}(\{0,1\}^n)$, and for $1 < i$,
- $S_{i-1} \Subset S_i$ iff there is some $S'_{i-1} \in S_i$ such that $S_{i-1} \Subset S'_{i-1}$, and
- $S_i \Subset S'_i$ iff $S_{i-1} \Subset S'_i$ for each $S_{i-1} \in S_i$.

For example, if $n = 2$ then $\{\{00,01\},\{01,10\},\{11\}\} \Subset_2 \{\{00,01,10\},\{00,11\}\}$ because $\{00,01\},\{01,10\} \Subset_1 \{00,01,10\}$ and $\{11\} \Subset_1 \{00,11\}$.

We can now define Σ as $\Sigma_n \cup \{\$\}$ and the infinite trace property P_n^ω over Σ:

$$(\epsilon \cup (\Sigma_n^* \cup \{X_n \$ X'_n \mid X_n, X'_n \in \Psi_n \text{ and } set_n(X'_n) \Subset_n set_n(X_n)\}) \cdot \{\$\}) \cdot \Sigma_n^\omega.$$

An infinite trace in P_n^ω therefore contains at most two \$ letters and infinitely many letters in Σ_n. There are no restrictions on the appearance of the letters in Σ_n when there is no \$ letter or when there is precisely one \$ letter. However, if the infinite trace contains precisely two \$ letters, that is, if it has the form $w\$w'\$u$ for some $w, w' \in \Sigma_n^\star$ and some $u \in \Sigma_n^\omega$, then w and w' must be in Ψ_n and the nested set corresponding to w must nested-include the nested set corresponding to w'; there are no restrictions on u.

Proposition 13 $P_n^\omega \in \mathit{Safety}^\omega$.

Proof: There are two cases to analyze for an infinite trace that is not in P_n^ω: when it contains more than two \$ letter, or when it has the form $w\$w'\$u$ with $w, w' \in \Sigma_n^\star$ and $u \in \Sigma_n^\omega$, but it is not the case that $w, w' \in \Psi_n$ and $set_n(w') \Subset_n set_n(w)$. In the first case, we can pick the first prefix of the infinite trace containing three \$ letters in total; clearly, this finite trace prefix cannot be continued into any acceptable infinite trace. In the second case, since there are no restrictions on the letters in u, we can easily see that the prefix $w\$w'\$$ is already a violation threshold: there is no $u' \in \Sigma_n^\omega$ such that $w\$w'\$u' \in P_n^\omega$. □

The bijection in the proof of Theorem 2 associates to each infinite-trace safety property a persistent finite-trace safety property by taking its

prefixes. Let P_n be the persistent finite-trace safety property $\mathsf{prefixes}(P_n^\omega)$ corresponding to P_n^ω. It is easy to see that P_n is the property

$$\Sigma_n^\star \bigcup \Sigma_n^\star \cdot \{\$\} \cdot \Sigma_n^\star \bigcup$$
$$\{X_n \$ X_n' \mid X_n, X_n' \in \Psi_n \text{ and } set_n(X_n') \stackrel{+}{\subseteq}_n set_n(X_n)\} \cdot \{\$\} \cdot \Sigma_n^\star.$$

Note that monitoring P_n^ω is the same as monitoring P_n: in both cases, besides the capability to checking whether there are more than two \$ letters, which is trivial, the monitor needs to store sufficient information about the nested set corresponding to X_n, so that, when the first \$ is seen, to be able to check whether it nested-includes the set corresponding to the upcoming, yet unknown X_n'.

Theorem 6 *Any synchronous or asynchronous monitor for P_n or P_n^ω needs space non-elementary in n, namely $\Omega(2^{2^{\cdot^{\cdot^{2^n}}}})$, with n nested power operations.*

Proof: Suppose that M is a monitor for P_n or P_n^ω and suppose that, during a monitoring session, it reads the prefix $X_n \in \Psi_n$. Regardless of how M is defined or implemented, in particular regardless of whether it reports violations synchronously or asynchronously, when the first \$ event is encountered, the state of M must contain enough information to sooner or later be able to decide whether the set $set_n(X_n')$ corresponding to *any* upcoming (unknown at the time the \$ is observed) sequence X_n' is nested-included in $set_n(X_n)$. Since $set(X_n')$ can in particular be equal to $set_n(X_n)$, and since once the second \$ event is observed there is no further event that may bring new knowledge to the monitor, we deduce that M must be able to distinguish any two different sets in $set_n(\Psi_n)$ when the first \$ event is encountered, that is, M's states must be different after reading words in Ψ_n whose corresponding nested sets are different. Therefore, M must be able to distinguish $|set_n(\Psi_n)|$ different possibilities. Since one needs $\Omega(\log N)$ space to distinguish among N different situations (one label for each), we conclude that M needs space $\Omega(\log(|set_n(\Psi_n)|))$, that is, $\Omega(2^{2^{\cdot^{\cdot^{2^n}}}})$ with n nested power operations. Hence, any monitor for P_n needs non-elementarily large space in n. □

We next show how to construct an ERE polynomial in size with n whose language is precisely P_n.

Theorem 7 *There is an ERE of size $O(n^3)$ whose language is P_n.*

Proof: Note that P_n is a union of three languages, the first two being trivial to express as languages of corresponding REs. As expected, the difficult part is to associate an ERE to the language

$$\{X_n\$\, X'_n \mid X_n, X'_n \in \Psi_n \text{ and } set_n(X'_n) \trianglelefteq_n set_n(X_n)\}.$$

Note that so far we did not need complementation. The property above can, however, be expressed as an ERE of size $O(n^3)$ using $O(n)$ nested complement operators. The idea is to define iteratively a sequence of EREs $\mathbb{K}_i$ for $0 \leq 1 \leq n$ whose languages contain words $X_i w\$w' X'_i$ with $set_i(X'_i) \trianglelefteq_i set(X_i)$, which are contiguous fragments of desired words $X_n\$X'_n$. Then $\mathbb{K}_n$ would be the language that we are looking for. To define $\mathbb{K}_i$, we observe that the nested-inclusion $S'_i \trianglelefteq_i S_i$ is equivalent to: there is no $S'_{i-1} \in S'_i$ such that it is not the case that we can find some $S_{i-1} \in S_i$ such that $S'_{i-1} \trianglelefteq_{i-1} S_{i-1}$. This crucial observation will allow us to define $\mathbb{K}_i$ in terms of $\mathbb{K}_{i-1}$. We next develop the technical details.

Let us first define regular patterns corresponding to each of the languages Ψ_i for $0 \leq i \leq n$; to avoid introducing new names, we ambiguously let the corresponding regular expressions have the same names as their languages:

- Let $\Psi_0 = (0+1)^n$, where for an RE, r^n is $r \cdot r \cdot \ldots \cdot r$ (n times); and
- Let $\Psi_i = \#_i \cdot \#_i + (\#_i \cdot \Psi_{i-1})^+ \cdot \#_i$ for all $1 \leq i \leq n$.

Iteratively eliminating the Ψ_{i-1} regular expressions from the right-hand-sides, we eventually obtain $n+1$ regular patters, each of size $O(n)$ (the size of Ψ_0 as a regular expression is $O(n)$ and each Ψ_i adds a constant size to that of Ψ_{i-1}).

Next we define REs for the languages $\mathsf{prefixes}(\Psi_i)$ and $\mathsf{suffixes}(\Psi_i)$ of prefixes and respectively suffixes of words in Ψ_i, for all $0 \leq i \leq n$. We only discuss the prefix closure languages; the suffix closures are dual. The prefix closures can be defined relatively easily inductively as follows:

- $\mathsf{prefixes}(\Psi_0) = \bigcup_{k=0}^{n} \{0,1\}^k$, and
- $\mathsf{prefixes}(\Psi_i) = \{\epsilon, \#_i\#_i\} \bigcup \{\#_i\} \cdot \mathsf{prefixes}(\Psi_{i-1}) \bigcup (\{\#_i\} \cdot \Psi_{i-1})^+ \bigcup (\{\#_i\} \cdot \Psi_{i-1})^+ \cdot \{\#_i\} \cdot \mathsf{prefixes}(\Psi_{i-1})$
  $= \{\#_i\#_i\} \bigcup (\{\#_i\} \cdot \Psi_{i-1})^\star \cdot (\{\epsilon\} \bigcup \{\#_i\} \cdot \mathsf{prefixes}(\Psi_{i-1}))$.

These languages can be expressed with the following REs; as before, we ambiguously use the same names for the corresponding REs:

- Let $\mathsf{prefixes}(\Psi_0) = \epsilon + (0+1) + (0+1)^2 + \cdots + (0+1)^n = (\epsilon + 0 + 1)^n$; and
- Let $\mathsf{prefixes}(\Psi_i) = \#_i \cdot \#_i + (\#_i \cdot \Psi_{i-1})^\star \cdot (\epsilon + \#_i \cdot \mathsf{prefixes}(\Psi_{i-1}))$.

Iteratively eliminating the REs $\mathsf{prefixes}(\Psi_{i-1})$ from the right-hand-sides, we eventually obtain $n+1$ REs, each of size $O(i^2+n)$ (the size of $\mathsf{prefixes}(\Psi_0)$ as an RE is $O(n)$ and each $\mathsf{prefixes}(\Psi_i)$ adds size $O(i)$ to that of $\mathsf{prefixes}(\Psi_{i-1})$). Dually,

- Let $\mathsf{suffixes}(\Psi_0) = \epsilon + (0+1) + (0+1)^2 + \cdots + (0+1)^n = (\epsilon + 0 + 1)^n$; and
- Let $\mathsf{suffixes}(\Psi_i) = \#_i \cdot \#_i + (\epsilon + \mathsf{suffixes}(\Psi_{i-1}) \cdot \#_i) \cdot (\Psi_{i-1} \cdot \#_i)^\star$.

We next define REs L_i and R_i for $0 \leq i \leq n$ whose languages contain the contiguous fragments of words in Ψ_n that are allowed to appear to the left and to the right of \$, respectively, so that words in $\mathcal{L}(L_i)$ start with $\#_i$ and words in $\mathcal{L}(R_i)$ end with $\#_i$. Let us also assume by convention that $L_{n+1} = R_{n+1} = \epsilon$ (the RE whose language contains only the empty word). It is easy to see that L_i and R_i can be defined as follows:

- Let $L_i = \#_i \cdot \Sigma_n^\star \cap \mathsf{suffixes}(\Psi_n)$, and
- Let $R_i = \Sigma_n^\star \cdot \#_i \cap \mathsf{prefixes}(\Psi_n)$.

Note that words in L_i and R_i may not necessarily start or end with a word in Ψ_i: indeed, the $\#_i$ that may start or end L_i or R_i could very well be followed or preceded, respectively, by a $\#_{i+1}$ or, if $i = n$, by \$.

Let us also define the EREs $\overline{L}_i$ and $\overline{R}_i$ whose languages are included in those of L_i and R_i, respectively, and whose words start or end with words in Ψ_i:

- Let $\overline{L}_i = \Psi_i \cdot \Sigma_n^* \cap \mathsf{suffixes}(\Psi_n)$, and
- Let $\overline{R}_i = \Sigma_n^* \cdot \Psi_i \cap \mathsf{prefixes}(\Psi_n)$.

It is not difficult to see that $\overline{L}_i = \Psi_i \cdot L_{i+1}$ and $\overline{R}_i = R_{i+1} \cdot \Psi_i$. Note that the sizes of L_i, R_i, $\overline{L}_i$ and $\overline{R}_i$ are $O(n^2)$.

Let us now define the EREs $\mathbb{K}_i$ for $0 \leq i \leq n$ as follows:

- $\mathbb{K}_0 = \overline{L}_0 \cdot \$ \cdot \overline{R}_0 \cap \bigcap_{k=0}^{n-1} (\Sigma_0^k \cdot 0 \cdot \Sigma^\star \cdot 0 \cdot \Sigma_0^{n-k-1} + \Sigma_0^k \cdot 1 \cdot \Sigma^\star \cdot 1 \cdot \Sigma_0^{n-k-1})$, and

- $\mathbb{K}_i = \overline{L}_i \cdot \$ \cdot \overline{R}_i \cap \neg((\neg((\#_i \cdot \Psi_{i-1})^\star \cdot \#_i \cdot \mathbb{K}_{i-1}) \cap L_i \cdot \$ \cdot R_{i-1}) \cdot (\#_i \cdot \Psi_{i-1})^\star \cdot \#_i)$.

We next show by induction on i that $\mathcal{L}(\mathbb{K}_i)$ is the language

$$\{X_i w X_i' \mid X_i, X_i' \in \Psi_i, w \in \mathcal{L}(L_{i+1} \cdot \$ \cdot R_{i+1}), set_i(X_i') \trianglelefteq_i set_i(X_i)\}.$$

It is easy to see that $\mathcal{L}(\mathbb{K}_0) = \{X_0 w X_0 \mid X_0 \in \Psi_0, w \in \mathcal{L}(L_1 \cdot \$ \cdot R_1)$, because the large conjunct in $\mathbb{K}_0$ states that the words formed with the first n letters and with the last ones, respectively, are equal and in Ψ_0, and because $\overline{L}_0 \cdot \$ \cdot \overline{R}_0 = \Psi_0 \cdot L_1 \cdot \$ \cdot R_1 \cdot \Psi_0$ and $\trianglelefteq_0$ is the identity on $\{0,1\}^n$. For the inductive step, let us now assume that for some arbitrary $1 \leq i < n$, $\mathcal{L}(\mathbb{K}_{i-1})$ is the language

$$\{X_{i-1} w X_{i-1}' \mid X_{i-1}, X_{i-1}' \in \Psi_{i-1}, w \in \mathcal{L}(L_i \cdot \$ \cdot R_i), set_{i-1}(X_{i-1}') \trianglelefteq_{i-1} set_{i-1}(X_{i-1})\}.$$

Then we can easily show that $\mathcal{L}((\#_i \cdot \Psi_{i-1})^\star \cdot \#_i \cdot \mathbb{K}_{i-1})$ is the language

$$\{X_i w X_{i-1}' \mid X_i \in \Psi_i, X_{i-1}' \in \Psi_{i-1}, w \in \mathcal{L}(L_{i+1} \cdot \$ \cdot R_i), set_{i-1}(X_{i-1}') \trianglelefteq_{i-1} set_i(X_i)\}.$$

Then we can show that $\mathcal{L}(\neg((\#_i \cdot \Psi_{i-1})^\star \cdot \#_i \cdot \mathbb{K}_{i-1}) \cap \overline{L}_i \cdot \$ \cdot \overline{R}_{i-1})$ is

$$\{X_i w X_{i-1}' \mid X_i \in \Psi_i, X_{i-1}' \in \Psi_{i-1}, w \in \mathcal{L}(L_{i+1} \cdot \$ \cdot R_i), set_{i-1}(X_{i-1}') \not\trianglelefteq_{i-1} set_i(X_i)\}.$$

Now we can show that $\mathcal{L}((\neg((\#_i \cdot \Psi_{i-1})^\star \cdot \#_i \cdot \mathbb{K}_{i-1}) \cap \overline{L}_i \cdot \$ \cdot \overline{R}_{i-1}) \cdot (\#_i \cdot \Psi_{i-1})^\star \cdot \#_i)$ is the language

$$\{X_i w X_i' \mid X_i, X_i' \in \Psi_i, w \in \mathcal{L}(L_{i+1} \cdot \$ \cdot R_{i+1}), set_i(X_i') \not\trianglelefteq_{i-1} set_i(X_i)\}.$$

Finally, we are now able to show that $\mathcal{L}(\mathbb{K}_i)$, that is,

$$\mathcal{L}(\overline{L}_i \cdot \$ \cdot \overline{R}_i \cap \neg((\neg((\#_i \cdot \Psi_{i-1})^\star \cdot \#_i \cdot \mathbb{K}_{i-1}) \cap L_i \cdot \$ \cdot R_{i-1}) \cdot (\#_i \cdot \Psi_{i-1})^\star \cdot \#_i))$$

is the language

$$\{X_i w X_i' \mid X_i, X_i' \in \Psi_i, w \in \mathcal{L}(L_{i+1} \cdot \$ \cdot R_{i+1}), set_i(X_i') \trianglelefteq_i set_i(X_i)\}.$$

Since $L_{n+1} = R_{n+1} = \epsilon$, it follows that

$$\mathcal{L}(\mathbb{K}_n) = \{X_n \$ X_n' \mid X_n, X_n' \in \Psi_n, set_i(X_i') \trianglelefteq_i set_i(X_i)\}.$$

The size of $\mathbb{K}_n$ is $O(n^3)$.

We can now show that the language of the ERE of size $O(n^3)$

$$(\epsilon + (\Sigma_n^\star + \mathbb{K}_n) \cdot \$) \cdot \Sigma_n^\star$$

is indeed P_n. □

We can now formulate our space lower-bound result for monitoring ERE-safety as a corollary of the two results above.

Corollary 1 *For any $n \in \mathbb{N}$, there is some safety property whose good prefixes are precisely the words in the language of an ERE of size $O(n)$ and whose monitoring (synchronous or asynchronous) requires space $\Omega(2^{2^{\cdot^{\cdot^{2^{\sqrt[3]{n}}}}}})$ with $\sqrt[3]{n}$ nested power operations.*

6.1.2 Generating Optimal Monitors for ERE

Abstract: Software engineers and programmers can easily understand regular patterns, as shown by the immense interest in and the success of scripting languages like Perl, based essentially on regular expression pattern matching. We believe that regular expressions provide an elegant and powerful specification language also for monitoring requirements*, because an execution trace of a program is in fact a string of states. Extended regular expressions (EREs) add complementation to regular expressions, which brings additional benefits by allowing one to specify patterns that must* not *occur during an execution. Complementation gives one the power to express patterns on strings more compactly. In this paper we present a technique to generate* optimal monitors *from EREs. Our monitors are deterministic finite automata (DFA) and our novel contribution is to generate them using a modern coalgebraic technique called* coinduction*. Based on experiments with our implementation, which can be publicly tested and used over the web, we believe that our technique is more efficient than the simplistic method based on complementation of automata which can quickly lead to a highly-exponential state explosion.*

Introduction

Regular expressions can express patterns in strings in a compact way. They proved very useful in practice; many programming/scripting languages like Perl, Python, Tcl/Tk support regular expressions as core features. Because of their power to express a rich class of patterns, regular expressions, are used not only in computer science but also in various other fields, such as molecular biology [31]. All these applications boast of very efficient implementation of regular expression pattern matching and/or membership algorithms. Moreover, it has been found that compactness of regular expressions can be increased non-elementarily by adding complementation

($\neg R$) to the usual union ($R_1 + R_2$), concatenation ($R_1 \cdot R_2$), and repetition (R^*) operators of regular expressions. These are known as *extended regular expressions* (EREs) and they proved very intuitive and succinct in expressing regular patterns.

Recent trends have shown that the software analysis community is inclining towards scalable techniques for software verification. Works in [22] merged temporal logics with testing, hereby getting the benefits of both worlds. The Temporal Rover tool (TR) and its follower DB Rover [14] are already commercial. In these tools the Java code is instrumented automatically so that it can check the satisfaction of temporal logic properties at runtime. The MaC tool [30, 38] has been developed to monitor safety properties in interval past time temporal logics. In [41, 42], various algorithms to generate testing automata from temporal logic formulae, are described. Java PathExplorer [20] is a runtime verification environment currently under development at NASA Ames. The Java MultiPathExplorer tool [51] proposes a technique to monitor all equivalent traces that can be extracted from a given execution, thus increasing the coverage of monitoring. [16, 21] present efficient algorithms for monitoring future time temporal logic formulae, while [24] gives a technique to synthesize efficient monitors from past time temporal formulae. [44] uses rewriting to perform runtime monitoring of EREs.

An interesting aspect of EREs is that they can express safety properties compactly, like those encountered in testing and monitoring. By generating automata from logical formulae, several of the works above show that the safety properties expressed by different variants of temporal logics are subclasses of regular languages. The converse is *not* true, because there are regular patterns which cannot be expressed using temporal logics, most notoriously those related to counting; e.g., the regular expression $(0 \cdot (0+1))^*$ saying that every other letter is 0 does not admit an equivalent temporal logic formula. Additionally, EREs tend to be often very natural and intuitive in expressing requirements. For example, let us try to capture the safety property "it should not be the case that in any trace of a traffic light we see green and then immediately red at any point". The natural and intuitive way to express it in ERE is $\neg((\neg\emptyset) \cdot \texttt{green} \cdot \texttt{red} \cdot (\neg\emptyset))$, where $\emptyset$ is the empty ERE (no words), so $\neg\emptyset$ means "anything".

Previous approaches to ERE membership testing [25, 40, 55, 34, 29] have focussed on developing techniques that are polynomial in both the size of the word and the size of the formulae. The best known result in these approaches is described in [34] where they can check if a word satisfies an ERE in time

$O(m \cdot n^2)$ and space $O(m \cdot m + k \cdot n^2)$, where m is the size of the ERE, n is the length of the word, and k is the number of negation/intersection operators. These algorithms, unfortunately, cannot be used for the purpose of monitoring. This is because they are not incremental. They assume the entire word is available before their execution. Additionally, their running time and space requirements are quadratic in the size of the trace. This is unacceptable when one has a long trace of events and wants to monitor a small ERE, as it is typically the case. This problem is removed in [44] where traces are checked against EREs through incremental rewriting. At present, we do not know if the technique in [44] is optimal or not.

A simple, straightforward, and practical approach is to generate optimal *deterministic finite automata* (DFA) from EREs [26]. This process involves the conversion of each negative sub-component of the ERE to a non-deterministic finite automaton (NFA), determinization of the NFA into a DFA, complementation of the DFA, and then its minimization. The algorithm runs in a bottom-up fashion starting from the innermost negative ERE sub components. This method, although generates the minimal automata, is too complex and cumbersome in practice. Its space requirements can be non-elementarily larger than the initial regular ERE, because negation involves an NFA-to-DFA translation, which implies an exponential blow-up; since negations can be nested, the size of such NFAs or DFAs could be highly exponential.

Our approach is to generate the minimal DFA from an ERE using coinductive techniques. In this paper, the DFA thus generated is called the *optimal monitor* for the given ERE. Currently, we are not aware of any other algorithm that does this conversion in a straightforward way. The complexity of our algorithm seems to be hard to evaluate, because it depends on the size of the minimal DFA associated to an ERE and we are not aware of any lower bound results in this direction. However, experiments are very encouraging. Our implementation, which is available for evaluation on the internet via a CGI server reachable from `http://fsl.cs.uiuc.edu/rv/`, rarely took longer than one second to generate a DFA, and it took only 18 minutes to generate the minimal 107 state DFA for the ERE in Example 12 which was used to show the exponential space lower bound of ERE monitoring in [44].

In a nutshell, in our approach we use the concept of derivatives of an ERE, as described in Subsection 6.1.2. For a given ERE one generates all possible derivatives of the ERE for all possible sequences of events. The size of this set of derivatives depends upon the size of the initial ERE. However,

several of these derivative EREs can be equivalent to each other. One can check the equivalence of EREs using coinductive technique as described in Section 6.1.2, that generates a set of equivalent EREs, called *circularities*. In Section 6.1.2, we show how circularities can be used to construct an efficient algorithm that generates optimal DFAs from EREs. In Section 6.1.2, we describe an implementation of this algorithm and give performance analysis results. We also made available on the internet a CGI interface to this algorithm.

Extended Regular Expressions and Derivatives

In this section we recall extended regular expressions and their derivatives.

Extended Regular Expressions

Extended regular expressions (ERE) define languages by inductively applying union ($+$), concatenation ($\cdot$), Kleene Closure ($_*$), intersection ($\cap$), and complementation ($\neg$). More precisely, for an alphabet E, whose elements are called *events* in this paper, an ERE over E is defined as follows, where $a \in E$:

$$R ::= \emptyset \mid \epsilon \mid a \mid R + R \mid R \cdot R \mid R^* \mid R \cap R \mid \neg R.$$

The language defined by an expression R, denoted by $\mathcal{L}(R)$, is defined inductively as

$$\begin{array}{rcl}
\mathcal{L}(\emptyset) & = & \emptyset, \\
\mathcal{L}(\epsilon) & = & \{\epsilon\}, \\
\mathcal{L}(A) & = & \{A\}, \\
\mathcal{L}(R_1 + R_2) & = & \mathcal{L}(R_1) \cup \mathcal{L}(R_2), \\
\mathcal{L}(R_1 \cdot R_2) & = & \{w_1 \cdot w_2 \mid w_1 \in \mathcal{L}(R_1) \text{ and } w_2 \in \mathcal{L}(R_2)\}, \\
\mathcal{L}(R^*) & = & (\mathcal{L}(R))^*, \\
\mathcal{L}(R_1 \cap R_2) & = & \mathcal{L}(R_1) \cap \mathcal{L}(R_2), \\
\mathcal{L}(\neg R) & = & \Sigma^* \setminus \mathcal{L}(R).
\end{array}$$

Given an ERE, as defined above using union, concatenation, Kleene Closure, intersection and complementation, one can translate it into an equivalent expression that does not have any intersection operation, by applying De Morgan's Law: $R_1 \cap R_2 = \neg(\neg R_1 + \neg R_2)$. The translation only results in a linear blowup in size. Therefore, in the rest of the paper we do not consider expressions containing intersection. More precisely, we only consider EREs of the form

$$R ::= R + R \mid R \cdot R \mid R^* \mid \neg R \mid a \mid \epsilon \mid \emptyset.$$

Derivatives

In this subsection we recall the notion of *derivative*, or "residual" (see [6, 5], where several interesting properties of derivatives are also presented). It is based on the idea of "event consumption", in the sense that an extended regular expression R and an event a produce another extended regular expression, denoted $R\{a\}$, with the property that for any trace w, $aw \in R$ if and only if $w \in R\{a\}$.

In the rest of the paper assume defined the typical operators on EREs and consider that the operator $\_ + \_$ is associative and commutative and that the operator $\_ \cdot \_$ is associative. In other words, reasoning is performed modulo the equations:

$$\begin{aligned}(R_1 + R_2) + R_3 &= R_1 + (R_2 + R_3),\\ R_1 + R_2 &= R_2 + R_1,\\ (R_1 \cdot R_2) \cdot R_3 &= R_1 \cdot (R_2 \cdot R_3).\end{aligned}$$

We next consider an operation $\_\{\_\}$ which takes an ERE and an event, and give several equations which define its operational semantics recursively, on the structure of regular expressions:

$$\begin{aligned}(R_1 + R_2)\{a\} &= R_1\{a\} + R_2\{a\} && (1)\\ (R_1 \cdot R_2)\{a\} &= (R_1\{a\}) \cdot R_2 + \texttt{if } (\epsilon \in R_1) \texttt{ then } R_2\{a\} \texttt{ else } \emptyset \texttt{ fi} && (2)\\ (R^*)\{a\} &= (R\{a\}) \cdot R^* && (3)\\ (\neg R)\{a\} &= \neg(R\{a\}) && (4)\\ b\{a\} &= \texttt{if } (b == a) \texttt{ then } \epsilon \texttt{ else } \emptyset \texttt{ fi} && (5)\\ \epsilon\{a\} &= \emptyset && (6)\\ \emptyset\{a\} &= \emptyset && (7)\end{aligned}$$

The right-hand sides of these equations use operations which we describe next. "`if` $(\_)$ `then` $\_$ `else` $\_$ `fi`" takes a boolean term and two EREs as arguments and has the expected meaning defined by two equations:

$$\begin{aligned}\texttt{if } (\mathit{true}) \texttt{ then } R_1 \texttt{ else } R_2 \texttt{ fi} &= R_1 && (8)\\ \texttt{if } (\mathit{false}) \texttt{ then } R_1 \texttt{ else } R_2 \texttt{ fi} &= R_2 && (9)\end{aligned}$$

We assume a set of equations that properly define boolean expressions and reasoning. Boolean expressions include the constants *true* and *false*, as well as the usual connectors $\_ \wedge \_$, $\_ \vee \_$, and *not*. Testing for empty trace membership (which is used by (2)) can be defined via the following equations:

$$\begin{aligned}
\epsilon \in (R_1 + R_2) &= (\epsilon \in R_1) \vee (\epsilon \in R_2) && (10)\\
\epsilon \in (R_1 \cdot R_2) &= (\epsilon \in R_1) \wedge (\epsilon \in R_2) && (11)\\
\epsilon \in (R^*) &= \mathit{true} && (12)\\
\epsilon \in (\neg R) &= \mathit{not}(\epsilon \in R) && (13)\\
\epsilon \in b &= \mathit{false} && (14)\\
\epsilon \in \epsilon &= \mathit{true} && (15)\\
\epsilon \in \emptyset &= \mathit{false} && (16)
\end{aligned}$$

The 16 equations above are natural and intuitive. [44] shows that these equations, when regarded as rewriting rules are terminating and ground Church-Rosser (modulo associativity and commutativity of $\_ + \_$ and modulo associativity of $\_ \cdot \_$), so they can be used as a functional procedure to calculate derivatives. Due to the fact that the 16 equations defining the derivatives can generate useless terms, in order to keep EREs compact we also propose defining several *simplifying equations*, including at least the following:

$$\begin{aligned}
\emptyset + R &= R,\\
\emptyset \cdot R &= \emptyset,\\
\epsilon \cdot R &= R,\\
R + R &= R.
\end{aligned}$$

The following result (see, e.g., [44] for a proof) gives a simple procedure, based on derivatives, to test whether a word belongs to the language of an ERE:

Theorem 8 *For any ERE $\mathcal{R}$ and any events a, a_1, a_2, ..., a_n in A, the following hold:*

1. *$a_1 a_2 ... a_n \in \mathcal{L}(R\{a\})$ if and only if $a a_1 a_2 ... a_n \in \mathcal{L}(R)$; and*
2. *$a_1 a_2 ... a_n \in \mathcal{L}(R)$ if and only if $\epsilon \in R\{a_1\}\{a_2\}...\{a_n\}$.*

Hidden Logic and Coinduction

We use circular coinduction, defined rigorously in the context of hidden logics and implemented in the BOBJ system [43, 17, 18], to test whether

two EREs are equivalent, that is, if they have the same language. Since the goal of this paper is to translate an ERE into a minimal DFA, standard techniques for checking equivalence, such as translating the two expressions into DFAs and then comparing those, do not make sense in this framework. A particularly appealing aspect of circular coinduction in the framework of EREs is that it does not only show that two EREs are equivalent, but also generates a larger set of equivalent EREs which will all be used in order to generate the target DFA.

Hidden logic is a natural extension of algebraic specification which benefits of a series of generalizations in order to capture various natural notions of behavioral equivalence found in the literature. It distinguishes *visible* sorts for data from *hidden* sorts for states, with states *behaviorally equivalent* if and only if they are indistinguishable under a formally given set of experiments. To keep the presentation simple and self contained, in this section we define an oversimplified version of hidden logic together with its associated circular coinduction proof rule, still general enough to support defining and proving EREs behaviorally equivalent.

Algebraic Preliminaries

The reader is assumed familiar with basic equational logic and algebra in this section. We recall a few notions in order to just make our notational conventions precise. An S-sorted signature Σ is a set of sorts/types S together with operational symbols on those, and a Σ-algebra A is a collection of sets $\{A_s \mid s \in S\}$ and a collection of functions appropriately defined on those sets, one for each operational symbol. Given an S-sorted signature Σ and an S-indexed set of variables Z, let $T_\Sigma(Z)$ denote the Σ-term algebra over variables in Z. If $V \subseteq S$ then $\Sigma{\restriction_V}$ is a V-sorted signature consisting of all those operations in Σ with sorts entirely in V. We may let $\sigma(X)$ denote the term $\sigma(x_1, ..., x_n)$ when the number of arguments of σ and their order and sorts are not important. If only one argument is important, then to simplify writing we place it at the beginning; for example, $\sigma(t, X)$ is a term having σ as root with only variables as arguments except one, and we do not care which one, which is t. If t is a Σ-term of sort s' over a special variable $*$ of sort s and A is a Σ-algebra, then $A_t : A_s \to A_{s'}$ is the usual interpretation of t in A.

Behavioral Equivalence, Satisfaction and Specification

Given disjoint sets V, H called *visible* and *hidden sorts*, a *hidden* (V, H)-*signature*, say Σ, is a many sorted $(V \cup H)$-signature. A *hidden subsignature*

of Σ is a hidden (V,H)-signature Γ with $\Gamma \subseteq \Sigma$ and $\Gamma{\restriction_V} = \Sigma{\restriction_V}$. The *data signature* is $\Sigma{\restriction_V}$. An operation of visible result not in $\Sigma{\restriction_V}$ is called an *attribute*, and a hidden sorted operation is called a *method*.

Unless otherwise stated, the rest of this section assumes fixed a hidden signature Σ with a fixed subsignature Γ. Informally, Σ-algebras are universes of possible states of a system, i.e., "black boxes," where one is only concerned with behavior under experiments with operations in Γ, where an experiment is an observation of a system attribute after perturbation; this is formalized below.

A Γ-*context for sort* $s \in V \cup H$ is a term in $T_\Gamma(\{* : s\})$ with one occurrence of $*$. A Γ-context of visible result sort is called a Γ-*experiment*. If c is a context for sort h and $t \in T_{\Sigma,h}$ then $c[t]$ denotes the term obtained from c by substituting t for $*$; we may also write $c[*]$ for the context itself.

Given a hidden Σ-algebra A with a hidden subsignature Γ, for sorts $s \in (V \cup H)$, we define Γ-*behavioral equivalence of* $a, a' \in A_s$ by $a \equiv^\Gamma_\Sigma a'$ iff $A_c(a) = A_c(a')$ for all Γ-experiments c; we may write $\equiv$ instead of $\equiv^\Gamma_\Sigma$ when Σ and Γ can be inferred from context. We require that all operations in Σ are compatible with $\equiv^\Gamma_\Sigma$. Note that behavioral equivalence is the identity on visible sorts, since the trivial contexts $* : v$ are experiments for all $v \in V$. A major result in hidden logics, underlying the foundations of coinduction, is that Γ-behavioral equivalence is the largest equivalence which is identity on visible sorts and which is compatible with the operations in Γ.

Behavioral satisfaction of equations can now be naturally defined in terms of behavioral equivalence. A hidden Σ-algebra A Γ-*behaviorally satisfies* a Σ-equation $(\forall X)\ t = t'$, say e, iff for each $\theta : X \to A$, $\theta(t) \equiv^\Gamma_\Sigma \theta(t')$; in this case we write $A \models^\Gamma_\Sigma e$. If E is a set of Σ-equations we then write $A \models^\Gamma_\Sigma E$ when A Γ-behaviorally satisfies each Σ-equation in E. We may omit Σ and/or Γ from $\models^\Gamma_\Sigma$ when they are clear.

A *behavioral* Σ-*specification* is a triple (Σ, Γ, E) where Σ is a hidden signature, Γ is a hidden subsignature of Σ, and E is a set of Σ-sentences equations. Non-data Γ-operations (i.e., in $\Gamma - \Sigma{\restriction_V}$) are called *behavioral*. A Σ-algebra A *behaviorally satisfies* a behavioral specification $\mathcal{B} = (\Sigma, \Gamma, E)$ iff $A \models^\Gamma_\Sigma E$, in which case we write $A \models \mathcal{B}$; also $\mathcal{B} \models e$ iff $A \models \mathcal{B}$ implies $A \models^\Gamma_\Sigma e$.

EREs can be very naturally defined as a behavioral specification. The enormous benefit of doing so is that the behavioral inference, including most importantly coinduction, provide a *decision procedure* for equivalence of EREs. [17] shows how standard regular expressions (without negation)

can be defined as a behavioral specification, a BOBJ implementation, and also how BOBJ with its circular coinductive rewriting algorithm can prove automatically several equivalences of regular expressions. Related interesting work can also be found in [48]. In this paper we extend that to general EREs, generate minimal observer monitors, and also give several other examples.

Example 6 *A behavioral specification of EREs defines a set of two visible sorts* $V = \{Bool,\ Event\}$*, one hidden sort* $H = \{Ere\}$*, one behavioral attribute* $\epsilon \in \_ : Ere \rightarrow Bool$ *and one behavioral method, the derivative,* $\_\{\_\} : Ere \times Event \rightarrow Ere$*, together with all the other operations in Subsection 6.1.2 defining EREs, including the events in* E *which are defined as visible constants of sort* Event*, and all the equations in Subsection 6.1.2. We call it the* ERE behavioral specification *and let* $\mathcal{B}_{ERE}$ *denote it.*

Since the only behavioral operators are the test for ϵ *membership and the derivative, it follows that the experiments have exactly the form* $\epsilon \in *\{a_1\}\{a_2\}...\{a_n\}$*, for any events* a_1*,* a_2*, ...,* a_n*. In other words, an experiment consists of a series of derivations followed by an* ϵ *membership test, and therefore two regular expressions are* behavioral equivalent *if and only if they cannot be distinguished by such experiments. Notice that the above reasoning applies within* any algebra *satisfying the presented behavioral specification. The one we are interested in is, of course, the* free *one, whose set carriers contain exactly the extended regular expressions as presented in Subsection 6.1.2, and the operations have the obvious interpretations. We informally call it the* ERE algebra*.*

Letting $\equiv$ denote the behavioral equivalence relation generated on the ERE algebra, then Theorem 8 immediately yields the following important result.

Theorem 9 *If* R_1 *and* R_2 *are two EREs then* $R_1 \equiv R_2$ *if and only if* $\mathcal{L}(R_1) = \mathcal{L}(R_2)$*.*

This theorem allows us to prove equivalence of EREs by making use of behavioral inference in the ERE behavioral specification, from now on simply referred to by $\mathcal{B}$, including (especially) circular coinduction. The next section shows how circular coinduction works and how it can be used to show EREs equivalent.

Circular Coinduction as an Inference Rule

In the simplified version of hidden logics defined above, the usual equational inference rules, i.e., reflexivity, symmetry, transitivity, substitution and

congruence [43] are all sound for behavioral satisfaction. However, equational reasoning can derive only a very limited amount of interesting behavioral equalities. For that reason, *circular coinduction* has been developed as a very powerful automated technique to show behavioral equivalence. We let $\Vdash$ denote the relation being defined by the equational rules plus circular coinduction, for deduction from a specification to an equation.

Before we present circular coinduction formally, we give the reader some intuitions by duality to structural induction. The reader who is only interested in using the presented procedure or who is not familiar with structural induction, can skip this paragraph. Inductive proofs show equality of terms $t(x), t'(x)$ over a given variable x (seen as a constant) by showing $t(\sigma(x))$ equals $t'(\sigma(x))$ for all σ in a basis, while circular coinduction shows terms t, t' behaviorally equivalent by showing equivalence of $\delta(t)$ and $\delta(t')$ for all behavioral operations δ. Coinduction applies behavioral operations at the top, while structural induction applies generator/constructor operations at the bottom. Both induction and circular coinduction assume some "frozen" instances of t, t' equal when checking the inductive/coinductive step: for induction, the terms are frozen at the bottom by replacing the induction variable by a constant, so that no other terms can be placed beneath the induction variable, while for coinduction, the terms are frozen at the top, so that they cannot be used as subterms of other terms (with some important but subtle exceptions which are not needed here; see [18]).

Freezing terms at the top is elegantly handled by a simple trick. Suppose every specification has a special visible sort b, and for each (hidden or visible) sort s in the specification, a special operation $[\_] : s \to b$. No equations are assumed for these operations and no user defined sentence can refer to them; they are there for technical reasons. Thus, with just the equational inference rules, for any behavioral specification $\mathcal{B}$ and any equation $(\forall X)\ t = t'$, it is necessarily the case that $\mathcal{B} \Vdash (\forall X)\ t = t'$ iff $\mathcal{B} \Vdash (\forall X)\ [t] = [t']$. The rule below preserves this property. Let the sort of t, t' be hidden; then

Circular Coinduction:

$$\frac{\mathcal{B} \cup \{(\forall X)\ [t] = [t']\} \Vdash (\forall X, W)\ [\delta(t, W)] = [\delta(t', W)], \text{for all appropriate } \delta \in \Gamma}{\mathcal{B} \Vdash (\forall X)\ t = t'}$$

We call the equation $(\forall X)\ [t] = [t']$ added to $\mathcal{B}$ a **circularity**; it could just as well have been called a coinduction hypothesis or a co-hypothesis, but we find the first name more intuitive because from a coalgebraic point

of view, coinduction is all about finding circularities.

Theorem 10 *The usual equational inference rules together with* Circular Coinduction *are sound. That means that if* $\mathcal{B} \Vdash (\forall X)\ t = t'$ *and* $sort(t, t') \neq b$*, or if* $\mathcal{B} \Vdash (\forall X)\ [t] = [t']$*, then* $\mathcal{B} \models (\forall X)\ t = t'$*.*

Example 7 *Suppose that we want to show that the EREs* $(a + b)^*$ *and* $(a^*b^*)^*$ *admit the same language. By Theorem 9, we can instead show that* $\mathcal{B}_{ERE} \models (\forall \emptyset)\ (a + b)^* = (a^*b^*)^*$*. Notice that* a *and* b *are treated as constant events here; one can also prove the result when* a *and* b *are variables, but one would need to first make use of the theorem of hidden constants [43]. To simplify writing, we omit the empty quantifier of equations. By the* Circular Coinduction *rule, one generates the following three proof obligations*

$$\begin{array}{lll}
\mathcal{B}_{ERE} \cup \{[(a+b)^*] = [(a^*b^*)^*]\} & \Vdash & [\epsilon \in (a+b)^*] = [\epsilon \in (a^*b^*)^*], \\
\mathcal{B}_{ERE} \cup \{[(a+b)^*] = [(a^*b^*)^*]\} & \Vdash & [(a+b)^*\{a\}] = [(a^*b^*)^*\{a\}], \\
\mathcal{B}_{ERE} \cup \{[(a+b)^*] = [(a^*b^*)^*]\} & \Vdash & [(a+b)^*\{b\}] = [(a^*b^*)^*\{b\}].
\end{array}$$

The first proof task follows immediately by using the equations in $\mathcal{B}$ *as rewriting rules, while the other two tasks reduce to*

$$\begin{array}{lll}
\mathcal{B}_{ERE} \cup \{[(a+b)^*] = [(a^*b^*)^*]\} & \Vdash & [(a+b)^*] = [a^*(a^*b^*)^*], \\
\mathcal{B}_{ERE} \cup \{[(a+b)^*] = [(a^*b^*)^*]\} & \Vdash & [(a+b)^*] = [b^*(a^*b^*)^*].
\end{array}$$

By applying Circular Coinduction *twice, after simplifying the two obvious proof tasks stating the* ϵ *membership, one gets the following four proof obligations*

$$\begin{array}{lll}
\mathcal{B}_{ERE} \cup \{[(a+b)^*] = [(a^*b^*)^*], [(a+b)^*] = [a^*(a^*b^*)^*]\} & \Vdash & [(a+b)^*\{a\}] = [a^*(a^*b^*)^*\{a\}], \\
\mathcal{B}_{ERE} \cup \{[(a+b)^*] = [(a^*b^*)^*], [(a+b)^*] = [a^*(a^*b^*)^*]\} & \Vdash & [(a+b)^*\{b\}] = [a^*(a^*b^*)^*\{b\}], \\
\mathcal{B}_{ERE} \cup \{[(a+b)^*] = [(a^*b^*)^*], [(a+b)^*] = [b^*(a^*b^*)^*]\} & \Vdash & [(a+b)^*\{a\}] = [b^*(a^*b^*)^*\{a\}], \\
\mathcal{B}_{ERE} \cup \{[(a+b)^*] = [(a^*b^*)^*], [(a+b)^*] = [b^*(a^*b^*)^*]\} & \Vdash & [(a+b)^*\{b\}] = [b^*(a^*b^*)^*\{b\}],
\end{array}$$

which, after simplification translate into

$$\begin{array}{lll}
\mathcal{B}_{ERE} \cup \{[(a+b)^*] = [(a^*b^*)^*], [(a+b)^*] = [a^*(a^*b^*)^*]\} & \Vdash & [(a+b)^*] = [a^*(a^*b^*)^*], \\
\mathcal{B}_{ERE} \cup \{[(a+b)^*] = [(a^*b^*)^*], [(a+b)^*] = [a^*(a^*b^*)^*]\} & \Vdash & [(a+b)^*] = [b^*(a^*b^*)^*], \\
\mathcal{B}_{ERE} \cup \{[(a+b)^*] = [(a^*b^*)^*], [(a+b)^*] = [b^*(a^*b^*)^*]\} & \Vdash & [(a+b)^*] = [a^*(a^*b^*)^*], \\
\mathcal{B}_{ERE} \cup \{[(a+b)^*] = [(a^*b^*)^*], [(a+b)^*] = [b^*(a^*b^*)^*]\} & \Vdash & [(a+b)^*] = [b^*(a^*b^*)^*],
\end{array}$$

Again by applying circular coinduction we get

$$\begin{array}{ll}
\mathcal{B}_{ERE} \cup \{[(a+b)^*] = [(a^*b^*)^*], [(a+b)^*] = [a^*(a^*b^*)^*], & [(a+b)^*] = [b^*(a^*b^*)^*]\} \Vdash \\
& [(a+b)^*] = [a^*(a^*b^*)^*], \\
\mathcal{B}_{ERE} \cup \{[(a+b)^*] = [(a^*b^*)^*], [(a+b)^*] = [a^*(a^*b^*)^*], & [(a+b)^*] = [b^*(a^*b^*)^*]\} \Vdash \\
& [(a+b)^*] = [b^*(a^*b^*)^*], \\
\mathcal{B}_{ERE} \cup \{[(a+b)^*] = [(a^*b^*)^*], [(a+b)^*] = [b^*(a^*b^*)^*], & [(a+b)^*] = [a^*(a^*b^*)^*]\} \Vdash \\
& [(a+b)^*] = [a^*(a^*b^*)^*], \\
\mathcal{B}_{ERE} \cup \{[(a+b)^*] = [(a^*b^*)^*], [(a+b)^*] = [b^*(a^*b^*)^*], & [(a+b)^*] = [a^*(a^*b^*)^*]\} \Vdash \\
& [(a+b)^*] = [b^*(a^*b^*)^*],
\end{array}$$

which now follow all immediately. Notice that BOBJ uses the newly added (to $\mathcal{B}_{ERE}$) equations as rewriting rules when it applies its circular coinductive rewriting algorithm, so the proof above is done slightly differently, but entirely automatically.

Example 8 *Suppose now that one wants to show that $\neg(a^*b) \equiv \epsilon + a^* + (a+b)^*b(a+b)(a+b)^*$. One can also do it entirely automatically by circular coinduction as above, generating the following list of circularities:*

$$\begin{array}{rcl}
[\neg(a^*b)] & = & [\epsilon + a^* + (a+b)^*b(a+b)(a+b)^*], \\
[\neg(\epsilon)] & = & [(a+b)^*b(a+b)(a+b)^* + (a+b)(a+b)^*], \\
[\neg(\emptyset)] & = & [(a+b)^*b(a+b)(a+b)^* + (a+b)^*], \\
[\neg(\emptyset)] & = & [(a+b)^*b(a+b)(a+b)^* + (a+b)(a+b)^* + (a+b)^*].
\end{array}$$

Example 9 *One can also show by circular coinduction that concrete EREs satisfy systems of guarded equations. This is an interesting but unrelated subject, so we do not discuss it in depth here. However, we show how easily one can prove by coinduction that a^*b is the solution of the equation $R = a \cdot R + b$. This equation can be given by adding a new ERE constant r to $\mathcal{B}_{ERE}$, together with the equations $\epsilon \in r = false$, $r\{a\} = r$, and $r\{b\} = \epsilon$.* Circular Coinduction *applied on the goal $r = a^*b$ generates the proof tasks:*

$$\begin{array}{lcl}
\mathcal{B}_{ERE} \cup \{[r] = [a^*b]\} & \Vdash & [\epsilon \in r] = [\epsilon \in a^*b], \\
\mathcal{B}_{ERE} \cup \{[r] = [a^*b]\} & \Vdash & [r\{a\}] = [a^*b\{a\}], \\
\mathcal{B}_{ERE} \cup \{[r] = [a^*b]\} & \Vdash & [r\{b\}] = [a^*b\{b\}],
\end{array}$$

which all follow immediately.

The following says that circular coinduction provides a decision procedure for equivalence of EREs.

Theorem 11 *If R_1 and R_2 are two EREs, then $\mathcal{L}(R_1) = \mathcal{L}(R_2)$ if and only if $\mathcal{B}_{ERE} \Vdash R_1 = R_2$. Moreover, since the rules in $\mathcal{B}_{ERE}$ are ground Church-Rosser and terminating,* circular coinductive rewriting *[17, 18], which iteratively rewrites proof tasks to their normal forms followed by a one step coinduction if needed, gives a decision procedure for ERE equivalence.*

Generating Minimal DFA Monitors by Coinduction

In this section we show how one can use the set of circularities generated by applying the circular coinduction rules in order to generate a minimal DFA from any ERE. This DFA can then be used as an optimal monitor for that ERE. The main idea here is to associate states in DFA to EREs obtained by deriving the initial ERE; when a new ERE is generated, it is tested for equivalence with all the other already generated EREs by using the coinductive procedure presented in the previous section. A crucial observation which significantly reduces the complexity of our procedure is that, once an equivalence is proved by circular coinductive rewriting, the entire set of circularities accumulated represent equivalent EREs. These can be used to later quickly infer the other equivalences, without having to generate the same circularities over and over again.

Since BOBJ does not (yet) provide any mechanism to return the set of circularities accumulated after proving a given behavioral equivalence, we were unable to use BOBJ to implement our optimal monitor generator. Instead, we have implemented our own version of coinductive rewriting engine for EREs, which is described below.

We are given an initial ERE R_0 over alphabet A and from that we want to generate the equivalent minimal DFA $D = (S, A, \delta, s_0, F)$, where S is the set of states, $\delta : S \times A \to S$ is the transition function, s_0 is the initial state, and $F \subseteq S$ is the set of final states. The coinductive rewriting engine explicitly accumulates the proven circularities in a set. The set is initialized to an empty set at the beginning of the algorithm. It is updated with the accumulated circularities whenever we prove equivalence of two regular expressions in the algorithm. The algorithm maintains the set of states S in the form of non-equivalent EREs. At the beginning of the algorithm S is initialized with a single element, which is the given ERE R_0. Next, we generate all the derivatives of the initial ERE one by one in a depth first manner. A derivative $R_x = R\{x\}$ is added to the set S, if the set does not contain any ERE equivalent to the derivative R_x. We then extend the transition function by setting $\delta(R, x) = R_x$. If an ERE R' equivalent to the

derivative already exists in the set S, we extend the transition function by setting $\delta(R, x) = R'$. To check if an ERE equivalent to the derivative R_x already exists in the set S, we sequentially go through all the elements of the set S and try to prove its equivalence with R_x. In testing the equivalence we first add the set of circularities to the initial $\mathcal{B}$. Then we invoke the coinductive procedure. If for some ERE $R' \in S$, we are able to prove that $R' \equiv R_x$ i.e $\mathcal{B} \cup Eq_{\text{all}} \cup Eq_{\text{new}} \Vdash \mathrm{R}' = \mathrm{R}_x$, then we add the new equivalences Eq_{new}, created by the coinductive procedure, to the set of circularities. Thus we reuse the already proven equivalences in future proofs.

The derivatives of the initial ERE R_0 with respect to all events in the alphabet A are generated in a depth first fashion. The pseudo code for the whole algorithm is given in Figure 1.

```
dfs(R)
begin
   foreach x ∈ A do
      R_x ← R{x};
      if ∃R' ∈ S such that B ∪ Eq_all ∪ Eq_new ⊪ R' = R_x then
         δ(R, x) = R';   Eq_all ← Eq_all ∪ Eq_new
      else S ← S ∪ {R_x};   δ(R, x) = R_x;  dfs(R_x); fi
   endfor
end
```

Figure 6.1: ERE to minimal DFA generation algorithm

In the procedure **dfs** the set of final states F consists of the EREs from S which contain ϵ. This can be tested efficiently using the equations (10-16) in Subsection 6.1.2. The DFA generated by the procedure **dfs** may now contain some states which are non-final and from which the DFA can never reach a final state. We remove these redundant states by doing a breadth first search in backward direction from the final states. This can be done in time linear in the size of the DFA.

Theorem 12 *If D is the DFA generated for a given ERE R by the above algorithm then*

1. $\mathcal{L}(D) = \mathcal{L}(R)$,
2. *D is the minimal DFA accepting $\mathcal{L}(R)$.*

Proof: Suppose $a_1a_2 \ldots a_n \in \mathcal{L}(R)$. Then $\epsilon \in R\{a_1\}\{a_2\} \ldots \{a_n\}$. If $R_i = R\{a_1\}\{a_2\} \ldots \{a_i\}$ then $R_{i+1} = R_i\{a_{i+1}\}$. To prove that $a_1a_2 \ldots a_n \in \mathcal{L}(D)$, we use induction to show that for each $1 \leq i \leq n$, $R_i \equiv \delta(R, a_1a_2 \ldots a_i)$. For the base case if $R_1 \equiv R\{a_1\}$ then **dfs** extends the transition function by setting $\delta(R, a_1) = R$. Therefore, $R_1 \equiv R = \delta(R, a_1)$. If $R_1 \not\equiv R$ then **dfs** extends δ by setting $\delta(R, a_1) = R_1$. So $R_1 \equiv \delta(R, a_1)$ holds in this case also. For the induction step let us assume that $R_i \equiv R' = \delta(R, a_1a_2 \ldots a_i)$. If $\delta(R', a_{i+1}) = R''$ then from the **dfs** procedure we can see that $R'' \equiv R'\{a_{i+1}\}$. However, $R_i\{a_{i+1}\} \equiv R'\{a_{i+1}\}$. So $R_{i+1} \equiv R'' = \delta(R', a_{i+1}) = \delta(R, a_1a_2 \ldots a_{i+1})$. Also notice $\epsilon in R_n \equiv \delta(R, a_1a_2 \ldots a_n)$; this implies that $\delta(R, a_1a_2 \ldots a_n)$ is a final state and hence $a_1a_2 \ldots a_n \in \mathcal{L}(D)$.

Now suppose $a_1a_2 \ldots a_n \in \mathcal{L}(D)$. The proof that $a_1a_2 \ldots a_n \in \mathcal{L}(R)$ goes in a similar way by showing that $R_i \equiv \delta(R, a_1, a_2 \ldots a_i)$. □

Implementation and Evaluation

We have implemented the coinductive rewriting engine in the rewriting specification language Maude 2.0 [11]. The interested readers can download the implementation from the website `http://fsl.cs.uiuc.edu/rv/`. The operations on extended regular languages that are supported by our implementation are `~ _` for negation, `_ *` for Kleene Closure, `_ _` for concatenation, `_ & _` for intersection, and `_ + _` for union in increasing order of precedence. Here, the intersection operator `_ & _` is a syntactic sugar and is translated to an ERE containing union and negation using De Morgan's Law:

```
eq R1 & R2 = ~ (~ R1 + ~ R2) .
```

To evaluate the performance of the algorithm we have generated the minimal DFA for all possible EREs of size up to 9. Surprisingly, the size of any DFA for EREs of size up to 9 did not exceed 9. Here the number of states gives the size of a DFA. The following table shows the performance of our procedure for the worst EREs of a given size. The code is executed on a

Pentium 4 2.4GHz, 4 GB RAM linux machine.

Size	ERE	no. of states in DFA	Time (ms)	Rewrites
4	¬ (a b)	4	< 1	863
5	(a ¬ b) *	4	< 1	1370
6	¬ ((a ¬ b) *)	4	1	1453
7	¬ (a ¬ a a)	6	1	2261
8	¬ ((a ¬ b) * b)	7	1	3778
9	¬ (a ¬ a b) b	9	5	9717

Example 10 *In particular, for the ERE ¬ (a ¬ a b) b the generated minimal DFA is given in Figure 6.2.*

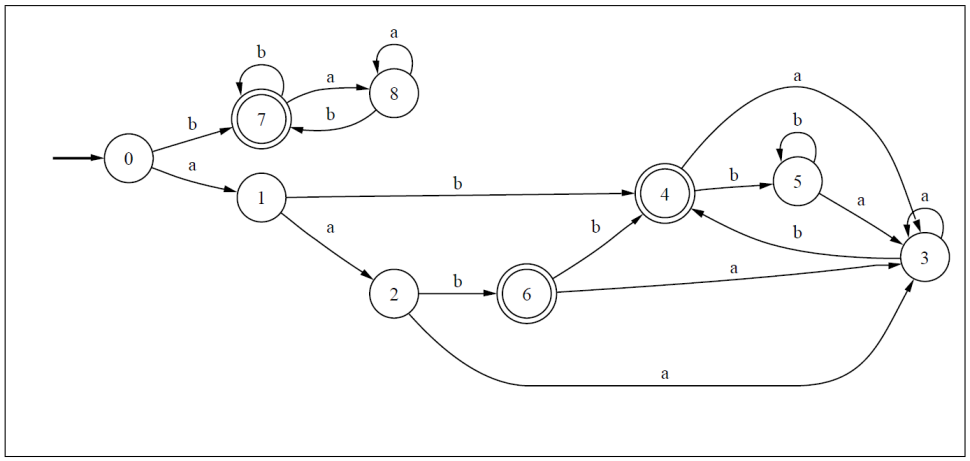

Figure 6.2: ¬ (a ¬ a b) b

Example 11 *The ERE ¬ ((¬* `empty`*) (*`green red`*) (¬* `empty`*)) states the safety property that it should not be the case that in any trace of a traffic light we see green and red consecutively at any point. The set of events are assumed to be {* `green, red, yellow`*}. We think that this is the most intuitive and natural expression for this safety property. The implementation took 1ms and 1663 rewrites to generate the minimal DFA with 2 states. The DFA is given in Figure 6.3.*

However for large EREs the algorithm may take a long time to generate a minimal DFA. The size of the generated DFA may grow non-elementarily

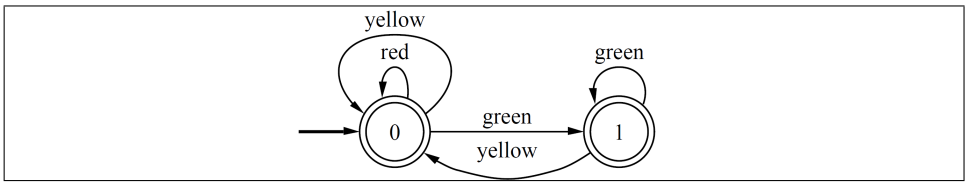

Figure 6.3: ¬ ((¬ `empty`) (`green red`) (¬ `empty`))

in the worst case. We generated DFAs for some complex EREs of larger sizes and got relatively promising results. One such sample result is as follows.

Example 12 *Let us consider the following ERE of size 110*

$$
\begin{array}{l}
(\neg\$)^*\$(\neg\$)^*\cap \\
\quad (0+1+\#)^*\#(\\
\qquad ((0+1)0\#(0+1+\#)^*\$(0+1)0+(0+1)1\#(0+1+\#)^*\$(0+1)1) \\
\qquad \cap(0(0+1)\#(0+1+\#)^*\$0(0+1)+1(0+1)\#(0+1+\#)^*\$1(0+1))).
\end{array}
$$

This ERE accepts the language L_2, where

$$L_k = \{\sigma\#w\#\sigma'\$w \mid w \in \{0,1\}^k \text{ and } \sigma,\sigma' \in \{0,1,\#\}^*\}$$

The language L_k was first introduced in [8] to show the power of alternation, used in [44] to show an exponential lower bound on ERE monitoring, and in [32, 33] to show the lower bounds for model checking. Our implementation took almost 18 minutes to generate the minimal DFA of size 107 and in the process it performed 1,374,089,220 rewrites.

The above example shows that the procedure can take a large amount of time and space to generate DFAs for large EREs. To avoid the computation associated with the generation of minimal DFA we plan to maintain a database of EREs and their corresponding minimal DFAs on the internet. Whenever someone wants to generate the minimal DFA for a given ERE he/she can look up the internet database for the minimal DFA. If the ERE and the corresponding DFA exists in the database he/she can retrieve the corresponding DFA and use it as a monitor. Otherwise, he/she can generate the minimal DFA for the ERE and submit it to the internet database to create a new entry. The database will check the equivalence of the submitted ERE and the corresponding minimal DFA and insert it in the database. In this way one can avoid the computation of generating minimal DFA if

it is already done by someone else. To further reduce the computation, circularities could also be stored in the database.

Online Monitor Generation and Visualization

We have extended our implementation to create an internet server for optimal monitor generation that can be accessed from the the url `http://fsl.cs.uiuc.edu/rv/`. Given an ERE the server generates the optimal DFA monitor for a user. The user submits the ERE through a web based form. A CGI script handling the web form takes the submitted ERE as an input, invokes the Maude implementation to generate the minimal DFA, and presents it to the user either as a graphical or a textual representation. To generate the graphical representation of the DFA we are currently using the GraphViz tool [15].

We presented a new technique to generate optimal monitors for extended regular expressions, which avoids the traditional technique based on complementation of automata, that we think is quite complex and not necessary. Instead, we have considered the (co)algebraic definition of EREs and applied coinductive inferencing techniques in an innovative way to generate the minimal DFA. Our approach to store already proven equivalences has resulted into a very efficient and straightforward algorithm to generate minimal DFA. We have evaluated our implementation on several hundreds EREs and have got promising results in terms of running time. Finally we have installed a server on the internet which can generate the optimal DFA for a given ERE.

At least two major contributions have been made. Firstly, we have shown that coinduction is a viable and quite practical method to prove equivalence of extended regular expressions. Previously this was done only for regular expressions without complementation. Secondly, building on the coinductive technique, we have devised an algorithm to generate minimal DFAs from EREs. At present we have no bound for the size of the optimal DFA, but we know for sure that the DFAs we generate are indeed optimal. However we know that the size of an optimal DFA is bounded by some exponential in the size of the ERE. As future work, it seems interesting to investigate the size of minimal DFAs generated from EREs, and also to apply our coinductive techniques to generate monitors for other logics, such as temporal logics.

6.2 Monitoring ω-Languages and LTL Safety Formulae

...

6.3 Optimal Monitoring of "Always Past" Temporal Safety

A monitor synthesis algorithm from linear temporal logic (LTL) safety formulae of the form $\Box\varphi$ where φ is a past time LTL formula was presented in [23]. The generated monitors implemented the recursive semantics of past-time LTL using a dynamic programming technique, and needed $O(|\varphi|)$ time to process each new event and $O(|\varphi|)$ total space. Some compiler-like optimizations of the generated monitors were also proposed in [23], which would further reduce the required space. It is not clear how much the required space could be reduced by applying those optimizations.

We here show how to generate using a divide-and-conquer technique directly monitors that need $O(k)$ space and still $O(|\varphi|)$ time, where k is the number of temporal operators in φ.

6.3.1 The Monitor Synthesis Algorithm

For simplicity, we assume only two past operators, namely $\circ$ (previously) and $\mathcal{S}$ (since). Let us first note that one cannot asymptotically reduce the space requirements below $\Omega(k)$, where k is the number of temporal operators appearing in the formula to monitor φ. Indeed, one can take $\varphi = (\#_1 \rightarrow t_1) \wedge \cdots \wedge (\#_k \rightarrow t_k)$, where for each $1 \leq i \leq k$, $\#_i$ is some event and t_i is some temporal formula containing precisely one past temporal operator, i.e., a $\circ$ or a $\mathcal{S}$. Any monitor for φ must directly or indirectly store the status of each t_i at every event, to be able to react accordingly in case the next event is some $\#_i$. Assuming that the events $\#_i$ are distinct and that the formulae t_i are unrelated, then the monitor needs to distinguish among 2^k possible states, so it needs $\Omega(k)$ space.

In what follows, we assume the usual recursive semantics of LTL, also presented below, restricted to safety formulae of the form $\Box\varphi$, where φ is a past-time LTL. We adopt the simplifying assumption that the empty trace invalidates any atomic proposition and any past temporal operator; as

argued in [23], this may not always be the best choice, but other semantic variations regarding the empty trace present no difficulties for monitoring.

Definition 23 *(adapted from [39]) LTL formulae of the form $\Box\varphi$ (read "always φ"), where φ is a past-time LTL formula, are called* LTL safety formulae*; we may call them just* safety formulae *when LTL is understood from the context. An infinite trace $u \in \Sigma^\omega$* satisfies *$\Box\varphi$, written $u \models \Box\varphi$, iff each $w \in \mathsf{prefixes}(u)$ satisfies the past-time LTL formula φ, written also $w \models \varphi$ and defined inductively as follows:*

$$
\begin{array}{lll}
w \models \mathit{true} & & \mathit{is\ always\ true,} \\
ws \models a & \mathit{iff} & a(s)\ \mathit{holds,} \\
w \models \neg\varphi & \mathit{iff} & w \not\models \varphi, \\
w \models \varphi_1 \wedge \varphi_2 & \mathit{iff} & w \models \varphi_1 \ \mathit{and}\ w \models \varphi_2, \\
ws \models \circ\varphi & \mathit{iff} & w \models \varphi, \\
ws \models \varphi_1 \mathcal{S} \varphi_2 & \mathit{iff} & ws \models \varphi_2 \ \mathit{or}\ ws \models F_\varphi \ \mathit{and}\ w \models \varphi_1 \mathcal{S} \varphi_2 \\
\epsilon \models \varphi & & \mathit{is\ false\ otherwise}
\end{array}
$$

Given safety formula $\Box\varphi$, we let $\mathcal{L}(\Box\varphi) \subseteq \Sigma^\omega$ be the set $\{u \in \Sigma^\omega \mid u \models \Box\varphi\}$.

Proposition 14 *$\mathcal{L}(\Box\varphi) \in \mathsf{Safety}^\omega$ for any past-time LTL formula φ.*

Proof: By the definition of $\mathcal{L}(\Box\varphi)$ in Definition 23 and the definition of $\Box P$ in Definition 12, one can easily note that $\mathcal{L}(\Box\varphi) = \Box\mathcal{L}(\varphi)$, where $\mathcal{L}(\varphi) = \{w \in \Sigma^\star \mid w \models \Box\varphi\}$. Therefore, $\mathcal{L}(\Box\varphi) \in \mathsf{Safety}^\omega_\Box$. The rest follows by Theorem 4. □

Let us next investigate the problem of monitoring safety properties $P \in \mathsf{Safety}^\omega$ expressed as languages of safety formulae, that is, $P = \mathcal{L}(\Box\varphi)$ for some past-time LTL formula φ. Because of the recursive nature of the satisfaction relation, a first important observation is that the generated monitor only needs to store information regarding the status of temporal operators from the previous state. More precisely, the monitor needs one bit per temporal operator, keeping the satisfaction status of the subformula corresponding to that temporal operator; when a new state is received, the satisfaction status of the subformula is recalculated according to the recursive semantics above and then the bit is updated. The order in which the temporal operators are processed when a new state is received is important: the nested operators must be processed first.

We next present the actual monitor synthesis algorithm at a high-level. We refrain from giving detailed pseudocode as we did in [23], because different

applications may choose different implementation paradigms. For example, we are currently using rewriting techniques to implement the monitor synthesis algorithms in MOP [10]; Section 6.3.2 shows our complete Maude rewriting implementation of the subsequent monitor synthesis algorithm.

Step 1 Let φ_1, ..., φ_k be the k subformulae of φ corresponding to temporal operators, such that, if φ_i is a subformula of φ_j, then $i < j$; this can be easily achieved by a DFS traversal of φ.

Step 2 Let $bit[1..k]$ be a vector of k bits initialized to 0 (or false); $bit[i]$ will store information related to φ_i from the previous state:

- if $\varphi_i = \circ\psi$ then $bit[i]$ says if ψ was satisfied at the previous state;
- if $\varphi_i = \psi \,\mathcal{S}\, \psi'$ then $bit[i]$ says if φ_i was satisfied at the previous step.

Step 3 Let $bit'[1..k]$ be another vector of k bits; this will be used to store temporary results, which will be moved eventually into the vector $bit[1..k]$.

Step 4 Generate a loop that executes whenever a new state s is available; the body of the loop executes the following code:

Step 4.1 For each i from 1 to k execute a bit assignment as follows, where for a subformula ψ of φ, $\overline{\psi}$ is the boolean expression replacing in ψ each non-nested temporal subformula φ_j by $bit[j]$ if φ_j is a "previously" formula or by $bit'[j]$ if φ_j is a "since" formula, and each remaining atomic proposition a by its satisfaction in the current state, $a(s)$:

- if $\varphi_i = \circ\psi$ then generate the assignment $bit'[i] := \overline{\psi}$
- if $\varphi_i = \psi \,\mathcal{S}\, \psi'$ then generate the assignment $bit'[i] := \overline{\psi'} \vee \overline{\psi} \wedge bit[i]$

Step 4.2 Generate the conditional: if $\overline{\varphi}$ is false then error (formula violated)

Step 4.3 Generate code to move the contents of $bit'[1..k]$ into $bit[1..k]$.

Note that the generated monitors are well-defined, because each time a $\overline{\psi}$ boolean expression is generated, all the bits in $bit'[1..k]$ that are needed are already calculated. One can also perform boolean simplifications when calculating $\overline{\psi}$ to reduce runtime overhead even further. For example, in

our implementation that also generated the code below (see Section 6.3.2), we used the simplification $\neg\neg\psi = \psi$. To illustrate the monitor generation algorithm above, let us consider the past time LTL formula: $\varphi = \neg(a \wedge \neg(\circ b \wedge (c\ \mathcal{S}\ (d \wedge (\neg e\ \mathcal{S}\ f)))))$. Step 1 produces the following enumeration of φ's subformulae: $\varphi_1 = \circ b$, $\varphi_2 = \neg e\ \mathcal{S}\ f$, and $\varphi_3 = c\ \mathcal{S}\ (d \wedge (\neg e\ \mathcal{S}\ f))$. The other steps eventually generate the code:

```
bit[1..3] := false;        // three global bits
foreach new state s do {
    // first update the bits in a consistent order
    bit'[1] := b(s);
    bit'[2] := f(s) ∨ (¬e(s) ∧ bit[2]);
    bit'[3] := d(s) ∧ bit'[2]) ∨ (c(s) ∧ bit[3]);
    // then check whether the formula is violated
    if a(s) ∧ ¬(bit[1] ∧ bit'[3]) then Error;
    // finally, update the state of the monitor
    bit[1..3] := bit'[1..3]
}
```

It is easy to see that for any past LTL formula φ of k temporal operators, the state of the generated monitor is encoded on k bits, namely the vector $bit[1..k]$. The runtime of the generated monitor is still $O(|\varphi|)$, because each temporal operator in φ results in an assignment and a read operation in the monitor, while each boolean operator in φ is "executed" by the monitor.

6.3.2 A Maude Implementation of the Monitor Synthesizer

We here show a term rewriting implementation of the algorithm above, using the Maude system [12]. Implementations in other languages are obviously also possible; however, rewriting proved to be an elegant means to generate monitors from logical formulae in several other contexts, and so seems to be here. In what follows we show the complete Maude code that takes as input a formula, parses it, generates the monitor, and then pretty prints it. We use the K technique here [46], which is a rewriting-based language and/or logic definitional technique; to use K, one needs to first upload the generic, i.e., application-independent, module discussed at the end of this section.

Atomic Predicates

We start by defining the atomic state predicates that one can use in formulae. These can be either identifiers (of the form `'a`, `'abc`, `'a123`, etc.; these are

provided by the Maude builtin module `QID`):

```
fmod PREDICATE is
  --- atomic predicates can be quoted identifiers
  protecting QID .
  sort Predicate .
  subsort Qid < Predicate .
endfm
```

Syntax of Formulae

Let us next define the syntax of formulae. We here use Maude's mixfix notation for defining syntax as algebraic operators, where underscores stay for arguments. Also, note that operators are assigned precedences (declared as operator attributes), to relive the user from writing parentheses (the lower the precedence the tighter the binding):

```
fmod SYNTAX is
  protecting PREDICATE .
  sort Formula .
  subsort Predicate < Formula .
  op !_ : Formula -> Formula [prec 20] .
  op _/\_ : Formula Formula -> Formula [prec 23] .
  op O_ : Formula -> Formula [prec 21] .
  op _S_ : Formula Formula -> Formula [prec 22] .
endfm
```

Target Language

We are done with the input language. Let us now define the output language. We need a very simple language for implementing the generated monitors, namely one with limited assignment, conditional and looping. The generated code, as well as the target language, play no role in this paper; one is expected to change the language below to one's desired target language (Java, C, C#, assembler, etc.). Our chosen language below has bits, expressions, statements and code. Bits are also expressions; code is a list of statements composed sequentially using "`;`" or just concatenation. The syntax below is also making use of precedence attributes. The `format` attributes are useful solely for pretty-printing reasons (see Maude's manual [12] for details on formatting):

```
fmod CODE is
  --- syntax for the generated code
```

```
  protecting PREDICATE + INT + STRING .
  sorts Bit Exp Statement Code .
  subsorts Bit < Exp .
  subsort Statement < Code .
  ops (bit[_]) (bit'[_]) : Nat -> Bit .
  ops (bit[1 .. _]) (bit'[1 .. _]) : Int -> Bit .
  op _(s) : Predicate -> Exp [prec 0] .
  ops true false : -> Exp .
  op !_ : Exp -> Exp [prec 20] .
  op _/\_ : Exp Exp -> Exp [prec 23] .
  op _\/_ : Exp Exp -> Exp [prec 24] .
  op _:=_ : Exp Exp -> Statement [prec 27 format(ni d d d)] .
  op if_then_ : Exp Statement -> Statement
                [format(ni d d ++ --) prec 30] .
  op foreach new state s do _ : Code -> Statement
                [format(n d d d d s++ --n)] .
  op Error : -> Statement [format(ni d)] .
  op //_ : String -> Statement [format(ni d d)] .
  op nil : -> Code .
  op _;_ : Code Code -> Code [assoc id: nil prec 40] .
  op __ : Code Code -> Code [assoc id: nil prec 40] .
  op {_} : Code -> Statement [format(d d --ni ++)] .
  --- code simplification rules
  var B : Exp .
  eq ! ! B = B .
endfm
```

The following module defines the actual monitor synthesis algorithm. We use the K definitional technique here, because it yields a very compact implementation. K is centered on the basic intuition of *computation*; computations are encoded as first-order data-structures that "evolve", via rewriting, to *results*. Computations are sequentialized using the list constructor "`_->_`"; thus, if `K` and `K'` are computations, then `K -> K'` is the computation consisting of `K` followed by `K'`. Computations may eventually yield results; for example, `K -> K'` may rewrite (in context) to `R -> K'`, meaning that `R` is the result that `K` reduces to. An important feature of K is that one can schedule lists of tasks for reduction; for example, `[K1,K2,K3] -> K` may eventually reduce to `[R1,R2,R3] -> K`, where `R1`, `R2`, and `R3` are the results that `K1`, `K2`, and `K3` reduce to, in this order. To use K, one needs to import the module `K` discussed at the and of this section. The equations of the module `K` (three in total) are all about reducing a list of computations to a list of results, supposing that one knows how to reduce one computation to one result.

K is a definitional framework that is generic in computations and results.

More precisely, it provides sorts `KComputation` and `KResult`, and expects its user to define the desired computations and results, as well as rules to reduce a computation to a result. Computations typically can be reduced to results only in context; to facilitate this, `K` provides a sort `KConfiguration`, which is also supposed to be populated accordingly. The sort `KConfiguration` is a multi-set sort over a sort `KConfigurationItem`, where the multi-set constructor is just concatenation; also, the sort `KComputation` is a list sort over `KComputationItem`, where the list constructor is `_->_`. To make use of `K`, one needs to first define constructors for the sorts `KConfigurationItem`, `KComputationItem` and `KResult`, and then to define how each computation item reduces to a result.

In our case, the computations are the formulae or subformulae that still need to be processed, and the results are the corresponding boolean expressions that need to be checked in the current (generated code) context to see whether the formula has been violated or not. We define the following additional constructors: we add four constructors for configurations, namely "`k`" that wraps the current computation, "`code`" that wraps the current generated code, and "`nextBit`" that wraps the next available bit; we add one main constructor for computations, "`form`", that wraps a formula, and one constant computation item per operator in the input language (the later is needed to know how to combine back the results of the corresponding subexpressions; finally, we add one constructor for results, "`exp`", that wraps a boolean expression.

The formula is processed in a depth-first-order, following a divide-and-conquer philosophy. Each subformula is decomposed into a list of computation subtasks consisting of its subformulae, then the corresponding results are composed back into a result corresponding to the original subformula. Recall that equations/rules apply wherever they match, not only at the top. Let us only discuss the two equations defining the "since" (`_S_`), the last two in the module below. The first one is straightforward: it decomposes the task of processing `F1 S F2` to the subtasks of processing `F1` and `F2`; the computation item `S` is placed in the computation structure to prepare the terrain for the next equation. The next equation applies after `F1` and `F2` have been processed, say to expressions `B1` and `B2`, respectively; if `C` is the code generated so far and if `I+1` is the next bit available, then the boolean expression corresponding to the current since formula is indeed `bit'(I+1)`, provided that one adds the corresponding code capturing the recursive semantics of since to the generated code.

```
fmod MONITOR-GENERATION is
  protecting K + SYNTAX + CODE .
  op k : KComputation -> KConfigurationItem .
  op code : Code -> KConfigurationItem .
  op nextBit : Nat -> KConfigurationItem .
  op process : Formula -> KConfiguration .
  op form : Formula -> KComputationItem .
  op exp : Exp -> KResult .
  ops ! /\ O S : -> KComputationItem .
  var P : Predicate .  vars F F1 F2 : Formula .  var C  : Code .
  var I : Nat .  vars B B1 B2 : Exp .  var K : KComputation .
  eq process(F) = k(form(F)) code(nil) nextBit(0) .
  eq k(form(P) -> K) = k(exp(P(s)) -> K) .
  eq form(! F) = form(F) -> ! .
  eq exp(B) -> ! = exp(! B) .
  eq form(F1 /\ F2) = [form(F1),form(F2)] -> /\ .
  eq [exp(B1),exp(B2)] -> /\ = exp(B1 /\ B2) .
  eq form(O F) = form(F) -> O .
  eq k(exp(B) -> O -> K) code(C) nextBit(I)
   = k(exp(bit[I + 1]) -> K) code(C ; bit'[I + 1] := B)
     nextBit(I + 1) .
  eq form(F1 S F2) = [form(F1), form(F2)] -> S .
  eq k([exp(B1),exp(B2)] -> S -> K) code(C) nextBit(I)
   = k(exp(bit'[I + 1]) -> K)
     code(C ; bit'[I + 1] := B2 \/ B1 /\ bit[I + 1]) nextBit(I + 1) .
endfm
```

Putting It All together

The following module plugs the code generated above into the general pattern:

```
fmod PRETTY-PRINT is
  protecting MONITOR-GENERATION .
  sort Monitor .
  op genMonitor : Formula -> Code .
  op makeMonitor : KConfiguration -> Code .

  var F : Formula .  var B : Exp .  var C : Code .  vars N M : Nat .
  eq genMonitor(F) = makeMonitor(process(F)) .
  eq makeMonitor(k(exp(B)) code(C) nextBit(N))
   = bit[1 .. N] := false ;
     foreach new state s do {
       // "first update the bits in a consistent order"
       C ;
       // "then check whether the formula is violated"
       if !(B) then Error ;
```

```
      // "finally, update the state of the monitor"
      bit[1 .. N] := bit'[1 .. N]
    } .
endfm
```

Our implementation of the monitor synthesizer is now complete. To use it, one can ask Maude reduce terms of the form `genMonitor(F)`, where `F` is the formula that one wants to generate into a monitor. For example:

```
reduce genMonitor(
     !('a /\ !(O 'b /\ 'c S ('d /\ (! 'e S 'f))))
) .
```

For the formula above, Maude will give the expected answer, pretty printed as follows:

```
     \||||||||||||||||||/
   --- Welcome to Maude ---
     /||||||||||||||||||\
    Maude 2.2 built: Mar 15 2006 16:37:22
     Copyright 1997-2005 SRI International
   Sat Jan 27 12:01:20 2007
Maude> in p
==========================================
fmod K
==========================================
fmod PREDICATE
==========================================
fmod SYNTAX
==========================================
fmod CODE
==========================================
fmod MONITOR-GENERATION
==========================================
fmod PRETTY-PRINT
==========================================
reduce in PRETTY-PRINT :
  genMonitor(! ('a /\ ! (O 'b /\ 'c S ('d /\ ! 'e S 'f)))) .
rewrites: 46 in -93406740ms cpu (1ms real) (~ rewrites/second)
result Code:
bit[1 .. 3] := false ;
foreach new state s do {
  // "first update the bits in a consistent order"
  bit'[1] := 'b(s) ;
  bit'[2] := 'f(s) \/ ! 'e(s) /\ bit[2] ;
  bit'[3] := 'd(s) /\ bit'[2] \/ 'c(s) /\ bit[3] ;
```

```
  // "then check whether the formula is violated"
  if 'a(s) /\ ! (bit[1] /\ bit'[3]) then
    Error ;
  // "finally, update the state of the monitor"
  bit[1 .. 3] := bit'[1 .. 3]
}

Maude>
```

The K Module

One should upload the next module whenever one wants to use the K technique to define a language, logic or tool. Note that the module below has nothing to do with our particular logic under consideration in this paper; that is the reason for which we exiled it here.

```
fmod K is
  sorts KConfigurationItem KConfiguration .
  subsort KConfigurationItem < KConfiguration .
  op empty : -> KConfiguration .
  op __ : KConfiguration KConfiguration -> KConfiguration [assoc comm id: empty] .

  sorts KComputationItem KNeComputation KComputation .
  subsort KComputationItem < KNeComputation < KComputation .
  op nil : -> KComputation .
  op _->_ : KComputation KComputation -> KComputation [assoc id: nil] .
  op _->_ : KNeComputation KNeComputation -> KNeComputation [ditto] .

  sort KComputationList .
  subsort KComputation < KComputationList .
  op nil : -> KComputationList .
  op _,_ : KComputationList KComputationList -> KComputationList [assoc id: nil] .

  sort KResult KResultList .
  subsorts KResult < KResultList < KComputation .
  op nil : -> KResultList .
  op _,_ : KResultList KResultList -> KResultList [assoc id: nil] .

  op [_] : KComputationList -> KComputationItem .
  op [_] : KResultList -> KComputationItem .
  op [_|_] : KComputationList KResultList -> KComputationItem .

  var K : KNeComputation .  var Kl : KComputationList .
  var R : KResult .       var Rl : KResultList .
  eq [K,Kl] = K -> [Kl | nil] .
  eq R -> [K,Kl | Rl] = K -> [Kl | Rl,R] .
```

```
  eq R -> [nil | Rl] = [Rl,R] .
endfm
```

To use K, after importing the module above, one should define one's own constructors for configuration items (sort `KConfigurationItem`), for computation items (sort `KComputationItem`), and for results (sort `KResult`). For our example, we defined all these at the beginning of the module `MONITOR-GENERATION`.

Chapter 7

Parametric Property Monitoring

Chapter 8

Predictive Runtime Analysis

Chapter 9

Static Analysis to Improve Runtime Verification

Chapter 10

Semantics-Based Runtime Verification

Chapter 11

Conclusion and Future Work

11.1 Safety Properties and Monitoring

Chapters 3 and 4 presented a comprehensive study of safety properties and of their monitoring, using a uniform formalism and notation. Technically, there were two novel contributions. First, it introduced the notion of a *persistent* safety property, which is the finite-trace correspondent of an infinite-trace safety property, and used it to show the cardinal equivalence of the various notions of safety property encountered in the literature. Second, it rigorously defined the problem of monitoring a safety property, and it showed that it can be arbitrarily hard. These results established a firm foundation for studying safety properties and corresponding monitors and algorithms for various domains of interest, where requirements can be expressed using domain-specific formalisms, such as future-time and past-time temporal logics, context-free grammars, push-down automata, and so on.

Bibliography

[1] Martín Abadi and Leslie Lamport. The existence of refinement mappings. In *LICS*, pages 165–175. IEEE Computer Society, 1988.

[2] Martín Abadi and Leslie Lamport. The existence of refinement mappings. *Theoretical Computer Science*, 82(2):253–284, 1991.

[3] Chris Allan, Pavel Avgustinov, Aske Simon Christensen, Laurie Hendren, Sascha Kuzins, Ondrej Lhoták, Oege de Moor, Damien Sereni, Ganesh Sittampalam, and Julian Tibble. Adding trace matching with free variables to AspectJ. In Richard P. Gabriel, editor, *ACM Conference on Object-Oriented Programming, Systems and Languages (OOPSLA)*, pages 345–364. ACM Press, 2005.

[4] Bowen Alpern and Fred B. Schneider. Defining liveness. *IPL*, 21(4):181–185, 1985.

[5] V. M. Antimirov. Partial derivatives of regular expressions and finite automaton constructions. *Theoretical Computer Science*, 155(2):291–319, 1996.

[6] V. M. Antimirov and P. D. Mosses. Rewriting extended regular expressions. *Theoretical Computer Science*, 143(1):51–72, 1995.

[7] Krzysztof R. Apt, Nissim Francez, and Shmuel Katz. Appraising fairness in languages for distributed programming. In *POPL*, pages 189–198, 1987.

[8] A. K. Chandra, D. C. Kozen, and L. J. Stockmeyer. Alternation. *Journal of the ACM*, 28(1):114–133, 1981.

[9] Ashok K. Chandra, Dexter Kozen, and Larry J. Stockmeyer. Alternation. *J. ACM*, 28(1):114–133, 1981.

[10] Feng Chen, Marcelo D'Amorim, and Grigore Roşu. Checking and correcting behaviors of Java programs at runtime with Java-MOP. In *RV'05*, volume 144(4) of *ENTCS*, 2005.

[11] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and J. F. Quesada. Towards Maude 2.0. In *3rd International Workshop on Rewriting Logic and its Applications (WRLA'00)*, volume 36 of *Electronic Notes in Theoretical Computer Science*. Elsevier, 2000.

[12] Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, and Carolyn Talcott. Maude Manual. `http://maude.cs.uiuc.edu`.

[13] Marcelo d'Amorim and Grigore Roşu. Efficient monitoring of ω-languages. In Kousha Etessami and Sriram K. Rajamani, editors, *CAV*, volume 3576 of *LNCS*, pages 364–378, 2005.

[14] D. Drusinsky. The Temporal Rover and the ATG Rover. In *SPIN Model Checking and Software Verification*, volume 1885 of *Lecture Notes in Computer Science*, pages 323–330. Springer, 2000.

[15] E. R. Gansner and S. C. North. An open graph visualization system and its applications to software engineering. *Software Practice and Experience*, 30(1):1203–1233, September 2000.

[16] D. Giannakopoulou and K. Havelund. Automata-Based Verification of Temporal Properties on Running Programs. In *Proceedings, International Conference on Automated Software Engineering (ASE'01)*, pages 412–416. Institute of Electrical and Electronics Engineers, 2001. Coronado Island, California.

[17] J. Goguen, K. Lin, and G. Roşu. Circular coinductive rewriting. In *Proceedings, Automated Software Engineering '00*, pages 123–131. IEEE, 2000. (Grenoble, France).

[18] J. Goguen, K. Lin, and G. Rosu. Conditional circular coinductive rewriting with case analysis. In *Recent Trends in Algebraic Development Techniques (WADT'02)*, Lecture Notes in Computer Science, to appear, Frauenchiemsee, Germany, September 2002. Springer-Verlag.

[19] Kevin W. Hamlen, J. Gregory Morrisett, and Fred B. Schneider. Computability classes for enforcement mechanisms. *ACM Trans. Program. Lang. Syst.*, 28(1):175–205, 2006.

[20] K. Havelund and G. Roşu. Java PathExplorer – A Runtime Verification Tool. In *The 6th International Symposium on Artificial Intelligence, Robotics and Automation in Space: A New Space Odyssey*, Montreal, Canada, June 18 - 21, 2001.

[21] K. Havelund and G. Roşu. Monitoring Programs using Rewriting. In *Proceedings, International Conference on Automated Software Engineering (ASE'01)*, pages 135–143. Institute of Electrical and Electronics Engineers, 2001. Coronado Island, California.

[22] K. Havelund and G. Roşu. *Runtime Verification 2002*, volume 70(4) of *Electronic Notes in Theoretical Computer Science*. Elsevier Science, 2002. Proceedings of a *Computer Aided Verification (CAV'02)* satellite workshop.

[23] K. Havelund and G. Roşu. Efficient monitoring of safety properties. *Software Tools and Technology Transfer*, 6(2):158–173, 2004. (also TACAS'02, LNCS 2280).

[24] K. Havelund and G. Roşu. Synthesizing monitors for safety properties. In *Tools and Algorithms for Construction and Analysis of Systems (TACAS'02)*, volume 2280 of *Lecture Notes in Computer Science*, pages 342–356. Springer, 2002.

[25] S. Hirst. A new algorithm solving membership of extended regular expressions. Technical report, The University of Sydney, 1989.

[26] J. E. Hopcroft and J. D. Ullman. *Introduction to Automata Theory, Languages and Computation*. Addison Wesley, 1979.

[27] J.E. Hopcroft and J.D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, 1979.

[28] L. Ilie, B. Shan, and S. Yu. Fast algorithms for extended regular expression matching and searching. In *Proceedings of STACS'03*, volume 2607 of *LNCS*, pages 179–190, 2003.

[29] L. Ilie, B. Shan, and S. Yu. Fast algorithms for extended regular expression matching and searching. In H. Alt and M. Habib, editors, *Proceedings of the 20th International Symposium on Theoretical Aspects of Computer (STACS 03)*, volume 2607 of *Lecture Notes in Computer Science*, page 179. Springer-Verlag, Berlin, 2003.

[30] M. Kim, S. Kannan, I. Lee, and O. Sokolsky. Java-MaC: a Run-time Assurance Tool for Java. In *Proceedings of Runtime Verification (RV'01)*, volume 55 of *Electronic Notes in Theoretical Computer Science*. Elsevier Science, 2001.

[31] J.R. Knight and E.W. Myers. Super-pattern matching. *Algorithmica*, 13(1/2):211–243, 1995.

[32] O. Kupferman and M. Y. Vardi. Freedom, Weakness, and Determinism: From Linear-Time to Branching-Time. In *Proc. of the IEEE Symposium on Logic in Computer Science*, pages 81–92, 1998.

[33] O. Kupferman and M. Y. Vardi. Model Checking of Safety Properties. In *Proc. of CAV'99: Conference on Computer-Aided Verification*, Trento, Italy, 1999.

[34] O. Kupferman and S. Zuhovitzky. An improved algorithm for the membership problem for extended regular expressions. In *Proc. of MFCS'02*, volume 2420 of *LNCS*, pages 446–458, 2002.

[35] Orna Kupferman and Moshe Y. Vardi. Model checking of safety properties. *Formal Methods in System Design*, 19(3):291–314, 2001.

[36] Leslie Lamport. Proving the correctness of multiprocess programs. *IEEE Trans. Software Eng.*, 3(2):125–143, 1977.

[37] Leslie Lamport. Logical foundation. In M. W. Alford, J. P. Ansart, G. Hommel, L. Lamport, B. Liskov, G. P. Mullery, F. B. Schneider, M. Paul, and H. J. Siegert, editors, *Distributed systems: Methods and tools for specification. An advanced course*, volume 190 of *LNCS*, pages 119–130. Springer-Verlag, 1985.

[38] I. Lee, S. Kannan, M. Kim, O. Sokolsky, and M. Viswanathan. Runtime Assurance Based on Formal Specifications. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications*, 1999.

[39] Zohar Manna and Amir Pnueli. *Temporal verification of reactive systems: safety*. Springer-Verlag New York, Inc., New York, NY, USA, 1995.

[40] G. Myers. A four russians algorithm for regular expression pattern matching. *Journal of the ACM*, 39(4):430–448, 1992.

[41] T. O'Malley, D. Richardson, and L. Dillon. Efficient Specification-Based Oracles for Critical Systems. In *In Proceedings of the California Software Symposium*, 1996.

[42] D. J. Richardson, S. L. Aha, and T. O. O'Malley. Specification-Based Test Oracles for Reactive Systems. In *Proceedings of the Fourteenth International Conference on Software Engineering, Melbourne, Australia*, pages 105–118, 1992.

[43] G. Roşu. *Hidden Logic*. PhD thesis, University of California at San Diego, 2000.

[44] G. Roşu and M. Viswanathan. Testing extended regular language membership incrementally by rewriting. In *RTA'03*, volume 2706 of *LNCS*. Springer, 2003.

[45] Grigore Roşu and Klaus Havelund. Rewriting-based techniques for runtime verification. *Automated Software Engineering*, 12(2):151–197, 2005.

[46] Grigore Roşu. K: a rewrite-based framework for modular lang. design, semantics, analysis and implementation (V2). Technical Report UIUCDCS-R-2006-2802, 2006.

[47] Grigore Rosu. An effective algorithm for the membership problem for extended regular expressions. In *Proceedings of the 10th International Conference on Foundations of Software Science and Computation Structures (FOSSACS'07)*, volume 4423 of *LNCS*, pages 332–345. Springer-Verlag, 2007.

[48] J. J. M. M. Rutten. Automata and coinduction (an exercise in coalgebra). In *Proceedings of the 9th International Conference on Concurrency Theory (CONCUR 98)*, volume 1466 of *Lecture Notes in Computer Science*, pages 194–218. Springer-Verlag, 1998.

[49] Fred B. Schneider. *On Concurrent Programming*. Springer, 1997.

[50] Fred B. Schneider. Enforceable security policies. *ACM Trans. Inf. Syst. Secur.*, 3(1):30–50, 2000.

[51] K. Sen, G. Roşu, and G. Agha. Runtime safety analysis of multithreaded programs. Technical Report UIUCDCS-R-2003-2334, University of Illinois at Urnaba Champaign, April 2003.

[52] L. J. Stockmeyer and A. R. Meyer. Word problems requiring exponential time (preliminary report). In *STOC*, pages 1–9. ACM Press, 1973.

[53] Larry Joseph Stockmeyer. *The Complexity of Decision Problems in Automata Theory and Logic*. PhD thesis, Massachusetts Institute of Technology, 1974.

[54] K. Thompson. Regular expression search algorithm. *CACM*, 11(6):419–422, 1968.

[55] H. Yamamoto. An automata-based recognition algorithm for semi-extended regular expressions. In *Proc. of MFCS'00*, volume 1893 of *LNCS*, pages 699–708, 2000.

[56] H. Yamamoto. A new recognition algorithm for extended regular expressions. In *Proceedings of ISAAC'01*, volume 2223 of *LNCS*, pages 257–267, 2001.

[57] H. Yamamoto and T. Miyazaki. A fast bit-parallel algorithm for matching extended regular expressions. In *Proc. of COCOON'03*, volume 2697 of *LNCS*, pages 222–231, 2003.