Comparing Trace Expressions and Linear Temporal Logic for Runtime Verification

Davide Ancona(✉), Angelo Ferrando, and Viviana Mascardi

DIBRIS, Università di Genova, Genoa, Italy
{davide.ancona,viviana.mascardi}@unige.it,
angelo.ferrando@dibris.unige.it

AQ1
AQ2

Abstract. Trace expressions are a compact and expressive formalism, initially devised for runtime verification of agent interactions in multiagent systems, which has been successfully employed to model real protocols, and to generate monitors for mainstream multiagent system platforms, and generalized to support runtime verification of different kinds of properties and systems.

In this paper we formally compare the expressive power of trace expressions with the Linear Temporal Logic (LTL), a formalism widely adopted in runtime verification. We show that any LTL formula can be translated into a trace expression which is equivalent from the point of view of runtime verification. Since trace expressions are able to express and verify sets of traces that are not context-free, we can derive that in the context of runtime verification trace expressions are more expressive than LTL.

1 Introduction

Runtime verification (RV) is a software verification technique that complements formal static verification (as model checking), and testing. In RV dynamic checking of the correct behavior of a system is performed by a monitor which is generated from a formal specification of the properties to be verified.

As happens for formal static verification, RV relies on a high level specification formalism to specify the expected properties of a system; similarly to testing, RV is a lightweight, effective but non exhaustive technique to verify complex properties of a system at runtime.

In contrast to formal static verification and testing, RV offers opportunities for error recovery which make this approach more attractive for the development of reliable software: not only a system can be constantly monitored for its whole lifetime to detect possible misbehavior, but also appropriate handlers can be executed for error recovery.

There are several specification formalisms employed by RV; some of them are well-known formalisms that have been originally introduced for other aims, as regular expressions, context free grammars, and LTL, while others have been expressly devised for RV.

V. Mascardi—Partly funded by "Progetto MIUR PRIN CINA Prot. 2010LHT4KM".

E. Ábrahám et al. (Eds.): de Boer Festschrift, LNCS 9660, pp. 1–18, 2016.
DOI: 10.1007/978-3-319-30734-3_6

Trace expressions belong to this latter group; they are an evolution of global types [2], which have been initially proposed for RV of agent interactions in multiagent systems. Trace expressions are an expressive formalism based on a set of operators (including prefixing, concatenation, shuffle, union, and intersection) to denote finite and infinite traces of events. Their semantics is based on a labeled transition system defined by a simple set of rewriting rules which directly drive the behavior of monitors generated from trace expressions.

In this paper we formally compare trace expressions with LTL, a formalism to specify infinite traces of events that is widely used for RV, even though it was initially introduced for model checking.

When used for RV, the expressive power of LTL is reduced, because at runtime only finite traces can be checked. For instance, the formula Fp (finally p) which states that an event satisfying the predicate p will eventually occur after a finite trace of other occurred events, can only be partially verified at runtime, because no monitor is able to reject an infinite trace of events that do not satisfy p, which, of course, is not a model for Fp.

To provide a formal account for this limitation, a three-valued semantics for LTL, called LTL_3 has been proposed [3]. A third truth value "?" is introduced to specify that after a finite trace of events has occurred, the outcome of a monitor can be inconclusive. For instance, if we consider the formula Fp, and the event e which does not satisfy p, then no monitor generated from Fp is able to decide whether Fp is satisfied or not after the trace eee.

In trace expressions this limitation of RV is naturally modeled by the standard semantics: if the semantics of a trace expression τ contains all finite traces e, ee, eee, ..., then it must also contain the infinite trace $e \dots e \dots$ because no monitor generated from τ will be able to reject it. This corresponds to the more formal claim stating that the semantics of any trace expression is a complete metric space of traces, when the standard distance between traces is considered.

As a consequence, when the standard semantics is considered, one can conclude that LTL and trace expressions are not comparable: neither is more expressive than the other. However, since the two formalisms are considered in the context of RV, if the more appropriate three-valued semantics is considered, then trace expressions are strictly more expressive than LTL: every LTL formula can be encoded into a trace expression with an equivalent three-valued semantics, whereas the opposite property does not hold, since trace expressions are also able to specify context-free and non context-free languages.

The paper is organized in the following way: Sect. 2 introduces trace expressions, whereas Sect. 3 is concerned with their expressive power; examples show that trace expressions can specify context-free and non context-free languages. Section 4 introduces LTL and the corresponding three-valued semantics, and formally compares this logic with trace expressions, while Sect. 5 provides a brief survey of related work. Conclusions are drawn in Sect. 6.

2 Trace Expressions

Trace expressions are a specification formalism expressly designed for RV; they are an evolution of global types, which have been initially proposed by Ancona, Drossopoulou and Mascardi [2] for RV of agent interactions in multiagent systems.

Trace expressions introduce three novelties:

- while global types are strongly based on the notion of agent interaction, because they have been expressly conceived for RV of protocol compliance in multiagent systems, trace expressions support a general notion of event, and can be exploited for RV in more general scenarios; for instance, besides agent interactions, trace expressions can be used for monitoring events as method invocations, or resource acquisition and release by threads;
- as a further generalization, trace expressions support the notion of *event type*: sets of events can be simply represented by predicates;
- besides the union (a.k.a. choice), concatenation, and shuffle (a.k.a. fork) operators, trace expressions support intersection as well. Intersection replaces the constrained shuffle operator [1,9], an extension of the shuffle operator introduced for making global types more expressive. Constrained shuffle imposes synchronization constraints on the events inside a shuffle, thus making global types and their semantics more complex; furthermore, constrained shuffle is not compositional: it cannot be expressed as an operation between sets of event traces (that is, the mathematical entities denoted by trace expressions). In contrast, the intersection operator has a simple, intuitive, and compositional semantics (as suggested by the name itself) and yet is very expressive; for instance, as shown in Sect. 3, it can be used for specifying non context-free sets of event traces.

Events. In the following we denote by $\mathcal{E}$ a fixed universe of events. An event trace over $\mathcal{E}$ is a possibly infinite sequence of events in $\mathcal{E}$. In the rest of the paper the meta-variables e, w, σ and u will range over the sets $\mathcal{E}$, $\mathcal{E}^\omega$, $\mathcal{E}^*$, and $\mathcal{E}^\omega \cup \mathcal{E}^*$, respectively; juxtaposition $e\,u$ denotes the trace where e is the first event, and u is the rest of the trace. A trace expression over $\mathcal{E}$ denotes a set of event traces over $\mathcal{E}$.

As a possible example, we might have

$$\mathcal{E} = \{o.m \mid o \text{ object identity}, m \text{ method name}\}$$

where the event $o.m$ corresponds to an invocation of method named[1] m on the target object o. This is a typical example of set of events arising when monitoring object-oriented systems (we will show an example later on).

[1] Here, for simplicity, an event does not include the signature of the method as it should be the case for those languages supporting static overloading.

Event Types. To be more general, trace expressions are built on top of event types (chosen from a set $\mathcal{ET}$), rather than of single events; an event type denotes a subset of $\mathcal{E}$, and corresponds to a predicate of arity $k \geq 1$, where the first implicit argument corresponds to the event e under consideration; referring to the example where events are method invocations, we may introduce the type $safe(o)$ of all safe method invocations for a given object o, defined by the predicate *safe* of arity 2 s.t. $safe(e, o)$ holds iff $e = o.isEmpty$.

The first argument of the predicate is left implicit in the event type, and we write $e \in safe(o)$ to mean that $safe(e, o)$ holds. Similarly, the set of events specified by an event type ϑ is denoted by $[\![\vartheta]\!]$; for instance, $[\![safe(o)]\!] = \{e \mid e \in safe(o)\}$.

For generality, we leave unspecified the formalism used for defining event types; however, in practice we do not expect that much expressive power is required. For instance, for all examples presented in this paper a formalism less powerful than regular expressions is sufficient.

Trace Expressions. A trace expression τ represents a set of possibly infinite event traces, and is defined on top of the following operators:[2]

- ϵ (empty trace), denoting the singleton set $\{\epsilon\}$ containing the empty event trace ϵ.
- $\vartheta{:}\tau$ (*prefix*), denoting the set of all traces whose first event e matches the event type ϑ ($e \in \vartheta$), and the remaining part is a trace of τ.
- $\tau_1 \cdot \tau_2$ (*concatenation*), denoting the set of all traces obtained by concatenating the traces of τ_1 with those of τ_2.
- $\tau_1 \wedge \tau_2$ (*intersection*), denoting the intersection of the traces of τ_1 and τ_2.
- $\tau_1 \vee \tau_2$ (*union*), denoting the union of the traces of τ_1 and τ_2.
- $\tau_1 | \tau_2$ (*shuffle*), denoting the set obtained by shuffling the traces of τ_1 with the traces of τ_2.

To support recursion without introducing an explicit construct, trace expressions are regular (a.k.a. rational or cyclic) terms: they correspond to trees where nodes are either the leaf ϵ, or the node (corresponding to the prefix operator) ϑ with one child, or the nodes $\cdot$, $\wedge$, $\vee$, and $|$ all having two children. According to the standard definition of rational trees, their depth is allowed to be infinite, but the number of their subtrees must be finite. As originally proposed by Courcelle [8], such regular trees can be modeled as partial functions from $\{0, 1\}^*$ to the set of nodes (in our case $\{\epsilon, \cdot, \wedge, \vee, |\} \cup \mathcal{ET}$) satisfying certain conditions.

A regular term can be represented by a finite set of syntactic equations, as happens, for instance, in most modern Prolog implementations where unification supports cyclic terms.

As an example of non recursive trace expression, let $\mathcal{E}$ be the set $\{e_1, \ldots, e_7\}$, and ϑ_i, $i = 1, \ldots, 7$, be the event types such that $e \in \vartheta_i$ iff $e = e_i$ (that is,

[2] Binary operators associate from left, and are listed in decreasing order of precedence, that is, the first operator has the highest precedence.

$[\![\vartheta_i]\!] = \{e_i\}$); then the trace expression

$$TE_1 = ((\vartheta_1 : \epsilon | \vartheta_2 : \epsilon) \vee (\vartheta_3 : \epsilon | \vartheta_4 : \epsilon)) \cdot (\vartheta_5 : \vartheta_6 : \epsilon | \vartheta_7 : \epsilon)$$

denotes the following set of event traces:

$$\left\{ \begin{array}{l} e_1e_2e_5e_6e_7, e_1e_2e_5e_7e_6, e_1e_2e_7e_5e_6, e_2e_1e_5e_6e_7, e_2e_1e_5e_7e_6, e_2e_1e_7e_5e_6, \\ e_3e_4e_5e_6e_7, e_3e_4e_5e_7e_6, e_3e_4e_7e_5e_6, e_4e_3e_5e_6e_7, e_4e_3e_5e_7e_6, e_4e_3e_7e_5e_6 \end{array} \right\}$$

As an example of recursive trace expression, if ϑ_i denotes the same event type defined above for $i = 1, \ldots, 7$, and $[\![\vartheta]\!] = \{e_4, e_5, e_6, e_7\}$, $[\![\vartheta']\!] = \{e_1, e_2, e_6, e_7\}$, and $[\![\vartheta'']\!] = \{e_1, e_2, e_3, e_4\}$, then the trace expression

$$\begin{array}{l} TE_2 = (E | \vartheta_1 : \vartheta_2 : \vartheta_3 : \epsilon) \wedge (E' | \vartheta_3 : \vartheta_4 : \vartheta_5 : \epsilon) \wedge (E'' | \vartheta_5 : \vartheta_6 : \vartheta_7 : \epsilon) \\ E = \epsilon \vee \vartheta : E \qquad E' = \epsilon \vee \vartheta' : E' \qquad E'' = \epsilon \vee \vartheta'' : E'' \end{array}$$

denotes the set $\{e_1e_2e_3e_4e_5e_6e_7\}$.

Finally, the recursive trace expressions $T_1 = (\epsilon \vee \vartheta_1 : T_1) \cdot T_2$, $T_2 = (\epsilon \vee \vartheta_2 : T_2)$ represent the infinite but regular terms $(\epsilon \vee \vartheta_1 : (\epsilon \vee \vartheta_1 : \ldots)) \cdot (\epsilon \vee \vartheta_2 : (\epsilon \vee \vartheta_2 : \ldots))$ and $(\epsilon \vee (\vartheta_2 : (\epsilon \vee (\vartheta_2 : \ldots))))$, respectively.

In the rest of the paper we will limit our investigation to *contractive* (a.k.a. *guarded*) trace expressions.

Definition 1. *A trace expression τ is* contractive *if all its infinite paths contain the prefix operator.*

In contractive trace expressions all recursive subexpressions must be guarded by the prefix operator; for instance, the trace expression defined by $T_1 = (\epsilon \vee (\vartheta : T_1))$ is contractive: its infinite path contains infinite occurrences of $\vee$, but also of the $:$ operator; conversely, the trace expression $T_2 = (\vartheta : T_2) | T_2$ is not contractive.

Trivially, every trace expression corresponding to a finite tree (that is, a non cyclic term) is contractive.

For all contractive trace expressions, any path from their root must always reach either a ϵ or a $:$ node in a finite number of steps. Since in this paper all definitions over trace expressions treat $\vartheta : \tau$ as a base case (that is, the definition is not propagated to the subexpression τ), restricting trace expressions to contractive ones has the advantage that most of the definitions and proofs requires induction, rather than coinduction, despite trace expressions can be cyclic. As a consequence, the implementation of trace expressions becomes considerably simpler. For this reason, in the rest of the paper we will only consider contractive trace expressions.

The semantics of trace expressions is specified by the transition relation $\delta \subseteq \mathcal{T} \times \mathcal{E} \times \mathcal{T}$, where $\mathcal{T}$ and $\mathcal{E}$ denote the set of trace expressions and of events, respectively. As it is customary, we write $\tau_1 \xrightarrow{e} \tau_2$ to mean $(\tau_1, e, \tau_2) \in \delta$. If the trace expression τ_1 specifies the current valid state of the system, then an event e is considered valid iff there exists a transition $\tau_1 \xrightarrow{e} \tau_2$; in such a case, τ_2 will specify the next valid state of the system after event e. Otherwise, the event e is

$$\text{(prefix)}\frac{}{\vartheta{:}\tau \xrightarrow{e} \tau}\ e\in\vartheta \qquad \text{(or-l)}\frac{\tau_1 \xrightarrow{e} \tau_1'}{\tau_1\vee\tau_2 \xrightarrow{e} \tau_1'} \qquad \text{(or-r)}\frac{\tau_2 \xrightarrow{e} \tau_2'}{\tau_1\vee\tau_2 \xrightarrow{e} \tau_2'}$$

$$\text{(and)}\frac{\tau_1 \xrightarrow{e} \tau_1' \quad \tau_2 \xrightarrow{e} \tau_2'}{\tau_1\wedge\tau_2 \xrightarrow{e} \tau_1'\wedge\tau_2'} \qquad \text{(shuffle-l)}\frac{\tau_1 \xrightarrow{e} \tau_1'}{\tau_1|\tau_2 \xrightarrow{e} \tau_1'|\tau_2} \qquad \text{(shuffle-r)}\frac{\tau_2 \xrightarrow{e} \tau_2'}{\tau_1|\tau_2 \xrightarrow{e} \tau_1|\tau_2'}$$

$$\text{(cat-l)}\frac{\tau_1 \xrightarrow{e} \tau_1'}{\tau_1\cdot\tau_2 \xrightarrow{e} \tau_1'\cdot\tau_2} \qquad \text{(cat-r)}\frac{\tau_2 \xrightarrow{e} \tau_2'}{\tau_1\cdot\tau_2 \xrightarrow{e} \tau_2'}\ \epsilon(\tau_1)$$

Fig. 1. Operational semantics of trace expressions

$$\text{(ϵ-empty)}\frac{}{\epsilon(\epsilon)} \qquad \text{(ϵ-or-l)}\frac{\epsilon(\tau_1)}{\epsilon(\tau_1\vee\tau_2)} \qquad \text{(ϵ-or-r)}\frac{\epsilon(\tau_2)}{\epsilon(\tau_1\vee\tau_2)} \qquad \text{(ϵ-shuffle)}\frac{\epsilon(\tau_1) \quad \epsilon(\tau_2)}{\epsilon(\tau_1|\tau_2)}$$

$$\text{(ϵ-cat)}\frac{\epsilon(\tau_1) \quad \epsilon(\tau_2)}{\epsilon(\tau_1\cdot\tau_2)} \qquad \text{(ϵ-and)}\frac{\epsilon(\tau_1) \quad \epsilon(\tau_2)}{\epsilon(\tau_1\wedge\tau_2)}$$

Fig. 2. Empty trace containment

not considered to be valid in the current state represented by τ_1. Figure 1 defines the inductive rules for the transition function.

While the transition relation δ with its corresponding rules in Fig. 1 defines the non empty traces of a trace expression, the predicate $\epsilon(\_)$, inductively defined by the rules in Fig. 2, defines the trace expressions that contain the empty trace ϵ. If $\epsilon(\tau)$ holds, then the empty trace is a valid trace for τ.

Rule (prefix) states that valid traces of $\vartheta{:}\tau$ can only start with an event e of type ϑ (side condition $e \in \vartheta$), and continue with traces in τ.

Rules (or-l) and (or-r) state that the only valid traces of $\tau_1\vee\tau_2$ have shape $e\ u$, where either $e\ u$ is valid for τ_1 (rule (or-l)), or $e\ u$ is valid for τ_2 (rule (or-r)).

Rule (and) states that the only valid traces of $\tau_1\wedge\tau_2$ have shape $e\ u$, where $e\ u$ is valid for both τ_1 and τ_2.

Rules (shuffle-l) and (shuffle-r) state that the only valid traces of $\tau_1|\tau_2$ have shape $e\ u$, where either $e\ u_1'$ and u_2 are valid traces for τ_1 and τ_2, respectively, and u can be obtained as the shuffle of u_1' with u_2 (rule (shuffle-l)), or u_1 and $e\ u_2'$ are valid traces for τ_1 and τ_2, respectively, and u can be obtained as the shuffle of u_1 with u_2' (rule (shuffle-r)).

Rules (cat-l) and (cat-r) state that the only valid traces of $\tau_1\cdot\tau_2$ have shape $e\ u$, where either $e\ u_1'$ and u_2 are valid traces for τ_1 and τ_2, respectively, and u can be obtained as the concatenation of u_1' to u_2 (rule (cat-l)), or ϵ is a valid trace for τ_1 (side condition $\epsilon(\tau_1)$) and $e\ u$ is a valid trace for τ_2 (rule (cat-r)).

For what concerns Fig. 2, rules (ϵ-shuffle), (ϵ-cat) and (ϵ-and) require the empty trace to be contained in both subexpressions τ_1 and τ_2, whereas for the union operator it suffices that the empty trace is contained in either τ_1 (rule (ϵ-or-l)) or τ_2 (rule (ϵ-or-r)). Trace expressions built with the prefix operator can never contain the empty trace, whereas ϵ contains just the empty trace (rule (ϵ-empty)).

The set of traces $[\![\tau]\!]$ denoted by a trace expression τ is defined in terms of the transition relation δ, and the predicate $\epsilon(\_)$. Since $[\![\tau]\!]$ may contain infinite traces, the definition of $[\![\tau]\!]$ is coinductive.

Definition 2. *For all possibly infinite event traces u and trace expressions τ, $u \in [\![\tau]\!]$ is coinductively defined as follows:*

- *either $u = \epsilon$ and $\epsilon(\tau)$ holds,*
- *or $u = e\,u'$, and there exists τ' s.t. $\tau \xrightarrow{e} \tau'$ and $u' \in [\![\tau']\!]$ hold.*

In the following we will need to consider the reflexive and transitive closure of the transition relation: if σ is a finite (possibly empty) event trace, then the relation $\tau \xrightarrow{\sigma} \tau'$ is inductively defined as follows: $\tau \xrightarrow{\sigma} \tau'$ holds iff

- $\sigma = \epsilon$, and $\tau' = \tau$;
- or $\sigma = e\,\sigma'$, and there exists τ'' s.t. $\tau \xrightarrow{e} \tau''$, and $\tau'' \xrightarrow{\sigma'} \tau'$.

Let us consider again the previous examples of trace expressions:

$$
\begin{array}{l}
TE_1 = ((\vartheta_1{:}\epsilon|\vartheta_2{:}\epsilon)\vee(\vartheta_3{:}\epsilon|\vartheta_4{:}\epsilon))\cdot(\vartheta_5{:}\vartheta_6{:}\epsilon|\vartheta_7{:}\epsilon) \\
TE_2 = (E|\vartheta_1{:}\vartheta_2{:}\vartheta_3{:}\epsilon)\wedge(E'|\vartheta_3{:}\vartheta_4{:}\vartheta_5{:}\epsilon)\wedge(E''|\vartheta_5{:}\vartheta_6{:}\vartheta_7{:}\epsilon) \\
E = \epsilon\vee\vartheta{:}E \qquad E' = \epsilon\vee\vartheta'{:}E' \qquad E'' = \epsilon\vee\vartheta''{:}E'' \\
\forall\, i \in \{1..7\}\ [\![\vartheta_i]\!] = \{e_i\} \qquad [\![\vartheta]\!] = \{e_4, e_5, e_6, e_7\} \\
[\![\vartheta']\!] = \{e_1, e_2, e_6, e_7\} \quad [\![\vartheta'']\!] = \{e_1, e_2, e_3, e_4\}
\end{array}
$$

We show that there exist τ_1, τ_2 s.t. $TE_1 \xrightarrow{\sigma_1} \tau_1$, with $\sigma_1 = e_1e_2e_5e_6e_7$, $\epsilon(\tau_1)$, $TE_2 \xrightarrow{\sigma_2} \tau_2$, with $\sigma_2 = e_1e_2e_3e_4e_5e_6e_7$, and $\epsilon(\tau_2)$.

For $TE_1 \xrightarrow{\sigma_1} \tau_1$ we have $\vartheta_1{:}\epsilon|\vartheta_2{:}\epsilon \xrightarrow{e_1} \epsilon|\vartheta_2{:}\epsilon \xrightarrow{e_2} \epsilon|\epsilon$, $(\vartheta_1{:}\epsilon|\vartheta_2{:}\epsilon)\vee(\vartheta_3{:}\epsilon|\vartheta_4{:}\epsilon) \xrightarrow{e_1e_2} \epsilon|\epsilon$, and $TE_1 \xrightarrow{e_1e_2} (\epsilon|\epsilon)\cdot(\vartheta_5{:}\vartheta_6{:}\epsilon|\vartheta_7{:}\epsilon)$. Furthermore, $\vartheta_5{:}\vartheta_6{:}\epsilon|\vartheta_7{:}\epsilon \xrightarrow{e_5} \vartheta_6{:}\epsilon|\vartheta_7{:}\epsilon \xrightarrow{e_6} \epsilon|\vartheta_7{:}\epsilon \xrightarrow{e_7} \epsilon|\epsilon$, hence $\vartheta_5{:}\vartheta_6{:}\epsilon|\vartheta_7{:}\epsilon \xrightarrow{e_5e_6e_7} \epsilon|\epsilon$, and, because $\epsilon(\epsilon|\epsilon)$, we can conclude $(\epsilon|\epsilon)\cdot(\vartheta_5{:}\vartheta_6{:}\epsilon|\vartheta_7{:}\epsilon) \xrightarrow{e_5e_6e_7} \epsilon|\epsilon$, hence, $TE_1 \xrightarrow{e_1e_2e_5e_6e_7} \epsilon|\epsilon$.

For $TE_2 \xrightarrow{\sigma_2} \tau_2$ we have $E|\vartheta_1{:}\vartheta_2{:}\vartheta_3{:}\epsilon \xrightarrow{e_1e_2e_3} E|\epsilon \xrightarrow{e_4e_5e_6e_7} E|\epsilon$, $E'|\vartheta_3{:}\vartheta_4{:}\vartheta_5{:}\epsilon \xrightarrow{e_1e_2} E'|\vartheta_3{:}\vartheta_4{:}\vartheta_5{:}\epsilon \xrightarrow{e_3e_4e_5} E'|\epsilon \xrightarrow{e_6e_7} E'|\epsilon$, $E''|\vartheta_5{:}\vartheta_6{:}\vartheta_7{:}\epsilon \xrightarrow{e_1e_2e_3e_4} E''|\vartheta_5{:}\vartheta_6{:}\vartheta_7{:}\epsilon \xrightarrow{e_5e_6e_7} E''|\epsilon$. Therefore $TE_2 \xrightarrow{e_1e_2e_3e_4e_5e_6e_7} (E|\epsilon)\wedge(E'|\epsilon)\wedge(E''|\epsilon)$ and $\epsilon(E|\epsilon)$, $\epsilon(E'|\epsilon)$, and $\epsilon(E''|\epsilon)$, hence $\epsilon((E|\epsilon)\wedge(E'|\epsilon)\wedge(E''|\epsilon))$.

Since the semantics of trace expressions is coinductive, they can specify non terminating behavior; for instance, the trace expression defined by $T = \vartheta_1{:}T$ denotes the set with just the infinite trace $e_1\,e_1 \ldots e_1 \ldots$ containing infinite occurrences of e_1; had we considered an inductive semantics, T would have denoted the empty set. For the very same reason, the trace expression defined by $T' = \epsilon\vee\vartheta_1{:}T'$ denotes the set containing all finite traces of the event e_1, but also the infinite trace $e_1\,e_1 \ldots e_1 \ldots$. From the point of view of RV, the only difference between the two types is that for T' the monitored system is allowed to halt at any time, whereas for T the system can never stop.

Since at runtime it is not possible to check that a given monitored system will always eventually stop, trace expressions cannot denote sets of traces

which are not complete metric spaces, with the standard distance between traces: $d(u_1, u_2) = 2^{-n}$, where n denotes the smallest index (starting from 0) at which the two traces are different; by convention, if the two traces are equal, than $n = \infty$, and $2^{-n} = 0$. For instance, if the semantics of a trace expression τ contains traces of arbitrarily large length of the event e_1, then it also contains the infinite trace $e_1\, e_1 \ldots e_1 \ldots$; indeed, the monitor associated with τ will not be able to reject it.

Such a limitation is independent of the used formalism, but it is intimately related to RV; as pointed out in Sect. 4, similar issues arise when LTL is used for RV: its semantics has to be revisited to take into account the fact that at runtime only finite traces can be monitored and checked.

Deterministic Trace Expressions. There are trace expressions τ for which the problem of word recognition is less efficient because of non determinism. Non determinism originates from the union, shuffle, and concatenation operators, because for each of them two possibly overlapping transition rules are defined.

Let us consider the trace expression $\tau = (\vartheta_1{:}\vartheta_2{:}\epsilon){\vee}(\vartheta_1{:}\vartheta_3{:}\epsilon)$, where $[\![\vartheta_i]\!] = \{e_i\}$ for $i = 1, \ldots, 3$. Both transitions $\tau \xrightarrow{e_1} \vartheta_2{:}\epsilon$ and $\tau \xrightarrow{e_1} \vartheta_3{:}\epsilon$ are valid, but $[\![\vartheta_2{:}\epsilon]\!] \neq [\![\vartheta_3{:}\epsilon]\!]$; therefore, to correctly accept the trace e_1e_3, both rules have to be applied simultaneously, and the set of trace expressions $\{\vartheta_2{:}\epsilon, \vartheta_3{:}\epsilon\}$ has to be considered, as it is done for non deterministic automata.

Similarly, for the trace expression $\tau' = (\vartheta_1{:}\vartheta_2{:}\epsilon)|(\vartheta_1{:}\vartheta_3{:}\epsilon)$, both transitions $\tau' \xrightarrow{e_1} (\vartheta_2{:}\epsilon)|(\vartheta_1{:}\vartheta_3{:}\epsilon)$ and $\tau' \xrightarrow{e_1} (\vartheta_1{:}\vartheta_2{:}\epsilon)|(\vartheta_3{:}\epsilon)$ are valid, but $[\![(\vartheta_2{:}\epsilon)|(\vartheta_1{:}\vartheta_3{:}\epsilon)]\!] \neq [\![(\vartheta_1{:}\vartheta_2{:}\epsilon)|(\vartheta_3{:}\epsilon)]\!]$.

Finally, for the trace expression $\tau'' = (\epsilon{\vee}\vartheta_1{:}\vartheta_2{:}\epsilon){\cdot}(\vartheta_1{:}\epsilon)$ both transitions $\tau'' \xrightarrow{e_1} (\vartheta_2{:}\epsilon){\cdot}(\vartheta_1{:}\epsilon)$ and $\tau'' \xrightarrow{e_1} \epsilon$ are valid, but $[\![(\vartheta_2{:}\epsilon){\cdot}(\vartheta_1{:}\epsilon)]\!] \neq [\![\epsilon]\!]$.

In the rest of this paper we will focus on deterministic trace expressions: indeed, the problem of word recognition is simpler and more efficient in the deterministic case.

Deterministic trace expressions are defined as follows.

Definition 3. *Let τ be a trace expression; τ is* deterministic *if for all finite event traces σ, if $\tau \xrightarrow{\sigma} \tau'$ and $\tau \xrightarrow{\sigma} \tau''$ are valid, then $[\![\tau']\!] = [\![\tau'']\!]$.*

The trace expressions τ, τ', and τ'', as defined above, are not deterministic, while the respectively equivalent trace expressions $\vartheta_1{:}(\vartheta_2{:}\epsilon{\vee}\vartheta_3{:}\epsilon)$, $\vartheta_1{:}(((\vartheta_2{:}\epsilon)|(\vartheta_1{:}\vartheta_3{:}\epsilon)){\vee}((\vartheta_1{:}\vartheta_2{:}\epsilon)|(\vartheta_3{:}\epsilon)))$, and $\vartheta_1{:}(\epsilon{\vee}\vartheta_2{:}\vartheta_1{:}\epsilon)$ are deterministic.

3 Examples of Specifications with Trace Expressions

In this section we provide some examples to show the expressive power of trace expressions. Unless specified otherwise, for simplicity in the rest of the paper we will consider singleton event types, that is, event types representing a single event; with abuse of notation, we will abbreviate events with their corresponding singleton event types.

3.1 Derived Operators

We first introduce some useful operators that will be used in the rest of the paper.

Constants. The constants 1 and 0 denote the set of all possible traces over $\mathcal{E}$ and the empty set, respectively. Constant 1 is equivalent to the expression $T = \epsilon \vee any{:}T$, where *any* is the event type s.t. $[\![any]\!] = \mathcal{E}$; constant 0 is equivalent to the expression $none{:}\epsilon$, where *none* is the event type s.t. $[\![none]\!] = \emptyset$.

Filter Operator. The filter operator is useful for making trace expressions more compact and readable. The expression $\vartheta \gg \tau$ denotes the set of all traces contained in τ, when deprived of all events that do not match ϑ. Assuming that event types are closed by complementation, the expression above is a convenient syntactic shortcut for $T|\tau$, where $T = \epsilon \vee \overline{\vartheta}{:}T$, and $\overline{\vartheta}$ is the complement event type of ϑ, that is, $[\![\overline{\vartheta}]\!] = \mathcal{E} \setminus [\![\vartheta]\!]$.

The corresponding rules for the transition relation and the auxiliary function $\epsilon(\_)$ can be easily derived:

$$\text{(cond-t)}\frac{\tau \xrightarrow{e} \tau'}{\vartheta \gg \tau \xrightarrow{e} \vartheta \gg \tau'}\ e \in \vartheta \qquad \text{(cond-f)}\frac{}{\vartheta \gg \tau \xrightarrow{e} \vartheta \gg \tau}\ e \notin \vartheta \qquad (\epsilon\text{-cond})\frac{\epsilon(\tau)}{\epsilon(\vartheta \gg \tau)}$$

Stack Objects. We expand the example where events correspond to method invocations on objects; besides the already introduced event type $safe(o)$ s.t. $e \in safe(o)$ iff $e = o.isEmpty$, we define the following other event types:

$$[\![pop(o)]\!] = \{o.pop\},\ [\![top(o)]\!] = \{o.top\},\ [\![push(o)]\!] = \{o.push\},$$
$$[\![stack(o)]\!] = \{o.pop, o.top, o.push, o.isEmpty\},$$
$$[\![unsafe(o)]\!] = \{o.pop, o.top, o.push\}.$$

Our purpose is to specify through a trace expression *Stack* all safe traces of method invocations on a stack object o which we assume to be initially empty. Safety requires that methods *top* and *pop* can never be invoked on o when o represents the empty stack.

More in details, a trace of method invocations on a given object having identity o is correct iff any finite prefix does not contain more $pop(o)$ event types than $push(o)$, and the event type $top(o)$ can appear only if the number of $pop(o)$ event types is strictly less than the number of $push(o)$ event types occurring before $top(o)$.

The trace expression *Stack* is defined as follows:

$$Stack = Any \wedge unsafe(o) \gg Unsafe \qquad Any = \epsilon \vee stack(o){:}Any$$
$$Unsafe = \epsilon \vee (push(o){:}(Unsafe | (Tops \cdot (pop(o){:}\epsilon \vee \epsilon)))) \qquad Tops = \epsilon \vee top(o){:}Tops$$

A correct stack trace is specified by *Stack* which is the intersection of *Any* and $unsafe(o) \gg Unsafe$; *Any* specifies any possible trace of method invocations on stack objects, whereas if an event has type $unsafe(o)$, then it has to verify

the trace expression *Unsafe*, which requires that a *push* event must precede a possible empty trace of *top* events, which, in turn, must precede an optional event *pop*; the expression is recursively shuffled with itself, since any *push* event can be safely shuffled with a *top* or a *pop* event.

The specification is deterministic. To make an example, we can consider $Stack \xrightarrow{\sigma} \tau$ with $\sigma = push(o)\, push(o)$, and

$$\tau = Any \wedge unsafe(o) \gg (Unsafe \,|\, Tops \cdot ((pop(o) : \epsilon) \vee \epsilon) \,|\, Tops \cdot ((pop(o) : \epsilon) \vee \epsilon)).$$

We may observe that $\tau \xrightarrow{e} \tau_1$ and $\tau \xrightarrow{e} \tau_2$, with[3] $e = pop(o)$, and

$$\begin{aligned} \tau_1 &= Any \wedge unsafe(o) \gg (Unsafe \,|\, \epsilon \,|\, Tops \cdot ((pop(o) : \epsilon) \vee \epsilon)) \\ \tau_2 &= Any \wedge unsafe(o) \gg (Unsafe \,|\, Tops \cdot ((pop(o) : \epsilon) \vee \epsilon) \,|\, \epsilon), \end{aligned}$$

but $[\![\tau_1]\!] = [\![\tau_2]\!]$.

3.2 Alternating Bit Protocol

A more complex example concerning interactions is the alternating bit protocol (ABP), as defined by Deniélou and Yoshida [11], where two parties, Alice and Bob, are involved, and four different types of events can occur: Alice sends a first kind of message to Bob (event type msg_1), Alice sends a second kind of message to Bob (event type msg_2), Bob replies to Alice with an acknowledge to the first kind of message (event type ack_1), Bob replies to Alice with an acknowledge to the second kind of message (event type ack_2). The protocol has to satisfy the following constraints for all event occurrences:

- The n-th occurrence of the event of type msg_1 must precede the n-th occurrence of the event of type msg_2, which, in turn, must precede the $(n+1)$-th occurrence of the event of type msg_1.
- The n-th occurrence of the event of type msg_1 must precede the n-th occurrence of the event of type ack_1, which, in turn, must precede the $(n+1)$-th occurrence of the event of type msg_1.
- The n-th occurrence of the event of type msg_2 must precede the n-th occurrence of the event of type ack_2, which, in turn, must precede the $(n+1)$-th occurrence of the event of type msg_2.

The protocol can be specified by the following trace expression (starting from variable $AltBit_1$):

$$\begin{aligned} AltBit_1 &= msg_1 : M_2 & AltBit_2 &= msg_2 : M_1 \\ M_1 &= msg_1 : A_2 \vee ack_2 : AltBit_1 & M_2 &= msg_2 : A_1 \vee ack_1 : AltBit_2 \\ A_1 &= ack_1 : M_1 \vee ack_2 : ack_1 : AltBit_1 & A_2 &= ack_2 : M_2 \vee ack_1 : ack_2 : AltBit_2 \end{aligned}$$

[3] For efficiency reasons, our implementation exploits simplification opportunities after each transition step, therefore in practice for this example the two transitions would lead to the same expression.

In this case the prefix and union operators are sufficient for specifying the correct behavior of the system, however, the corresponding trace expression is not very readable. More importantly, if only the prefix and union operators are employed, the size of the expressions grows exponentially with the number of different involved event types.

This problem can be avoided by the use of the intersection and filter operators.

Let $msg\_ack(i)$, $i = 1, 2$, and msg denote the event types s.t. $[\![msg\_ack(i)]\!] = [\![msg_i]\!] \cup [\![ack_i]\!]$, $i = 1, 2$, and $[\![msg]\!] = [\![msg_1]\!] \cup [\![msg_2]\!]$. Then the ABP can be specified by the following deterministic trace expression:

$$AltBit = (msg \gg MM) \wedge (msg\_ack(1) \gg MA_1) \wedge (msg\_ack(2) \gg MA_2)$$
$$MM = msg_1{:}msg_2{:}MM \quad MA_1 = msg_1{:}ack_1{:}MA_1 \quad MA_2 = msg_2{:}ack_2{:}MA_2$$

The three trace expressions defined by MM, MA_1, and MA_2 correspond to the three constraints informally stated above. The main trace expression $AltBit$ can be easily read as follows: if an event has type msg_1 or msg_2, then it must verify MM, and if an event has type msg_1 or ack_1, then it must verify MA_1, and if an event has type msg_2 or ack_2, then it must verify MA_2.

The trace expression can be easily generalized to k different kinds of messages (with $k \geq 2$), with the size of the expression growing linearly with the number of different involved event types. For instance, for $k = 3$ we have the following trace expression:

$$\begin{aligned}
&AltBit = \\
&(msg \gg MM) \wedge (msg_ack(1) \gg MA_1) \wedge (msg_ack(2) \gg MA_2) \wedge (msg_ack(3) \gg MA_3) \\
&MM = msg_1{:}msg_2{:}msg_3{:}MM \quad MA_1 = msg_1{:}ack_1{:}MA_1 \\
&MA_2 = msg_2{:}ack_2{:}MA_2 \qquad\quad MA_3 = msg_3{:}ack_3{:}MA_2.
\end{aligned}$$

3.3 Non Context Free Languages

Trace expressions allow the specification of non context free languages; let us consider for instance the typical example of non context free language $\{a^n b^n c^n \mid n \geq 0\}$. This language can be specified by the following trace expression (defined by T)

$$T = (a\_or\_b \gg AB) \wedge (b\_or\_c \gg BC) \qquad AB = \epsilon \vee (a{:}(AB \cdot (b{:}\epsilon)))$$
$$BC = \epsilon \vee (b{:}(BC \cdot (c{:}\epsilon)))$$

where $[\![a]\!] = \{a\}$, $[\![b]\!] = \{b\}$, $[\![c]\!] = \{c\}$, $[\![a\_or\_b]\!] = \{a, b\}$, and $[\![b\_or\_c]\!] = \{b, c\}$.

Assuming the universe of events $\mathcal{E} = \{a, b, c\}$, the expression $a\_or\_b \gg AB$ denotes all traces of events over $\mathcal{E}$ that, when restricted to finite length[4] and to events a or b, correspond to the sequence $a^n b^n$ for some $n \in \mathbb{N}$; similarly, the

[4] Recall that for a comparison with context-free languages we need to disregard infinite traces; for instance, $a\_or\_b \gg AB$ and $b\_or\_c \gg BC$ contain also the infinite traces a^ω and b^ω, respectively.

expression $b\_or\_c \gg BC$ denotes all traces of events over $\mathcal{E}$ that, when restricted to finite length and to events b or c, correspond to the sequence $b^n c^n$ for some $n \in \mathbb{N}$. Hence the finite traces of T, which is the intersection of $a\_or\_b \gg AB$ and $b\_or\_c \gg BC$, are the non-context free language $\{a^n b^n c^n \mid n \geq 0\}$.

Although T is deterministic, it has the drawback that non correct traces can be detected with a certain latency. For instance the transition $T \xrightarrow{aabc} T'$ holds, with $T' = (a\_or\_b \gg (b{:}\epsilon)) \wedge (b\_or\_c >> \epsilon)$, and clearly $aabc$ is not a valid prefix for the language; however, $[\![T']\!] = \emptyset$, and T' is not able to accept any further event, that is, recognition fails, independently from the next event.

To avoid this problem, the following equivalent (assuming that $\mathcal{E} = \{a, b, c\}$) deterministic trace expression can be employed:

$$T_2 = (AB{\cdot}C) \wedge (b\_or\_c \gg BC) \quad AB = \epsilon \vee (a{:}(AB{\cdot}(b{:}\epsilon)))$$
$$BC = \epsilon \vee (b{:}(BC{\cdot}(c{:}\epsilon))) \qquad C = \epsilon \vee c{:}C$$

In this case, $AB{\cdot}C$ forces events of type c to occur only after all required events of type b have been already occurred. In this case there is no T_2'' s.t. $T_2 \xrightarrow{aabc} T_2''$ holds; indeed, $T_2 \xrightarrow{aab} T_2'$ with $T_2' = ((b{:}\epsilon){\cdot}(\epsilon \vee (c{:}C))) \wedge (b\_or\_c \gg (BC{\cdot}(c{:}\epsilon)))$, and there exists no T_2'' s.t. $T_2' \xrightarrow{c} T_2''$, since the only possible transition from T_2' is $T_2' \xrightarrow{b} T_2''$, with $T_2'' = (\epsilon \vee (c{:}C)) \wedge (b\_or\_c \gg ((\epsilon \vee (b{:}BC{\cdot}(c{:}\epsilon))){\cdot}((c{:}\epsilon){\cdot}(c{:}\epsilon))))$, and $[\![T_2'']\!] = \{cc\}$.

4 Comparison with LTL

In this section we formally prove that trace expressions are more expressive than LTL, when both formalisms are used for RV. To this purpose we consider the LTL_3 semantics [3], an adaptation of the standard semantics of LTL formulas expressly introduced to take into account the limitations of RV due to its inability to check infinite traces. Despite there are LTL formulas which do not have an equivalent trace expression according to the standard LTL semantics, when LTL_3 is considered such a difference is no longer exhibited: for any LTL formula φ it is possible to build a contractive and deterministic trace expression τ such that the monitors generated by φ and τ, respectively, are behaviorally equivalent.

4.1 Background

LTL is a modal logic which has been introduced for specifying temporal properties of systems; despite its original main application is static verification through model checking, more recently it has been adopted as a specification formalism for RV, and some RV tools support it [6,12].

LTL Syntax and Semantics. Given a finite set of atomic propositions AP, the set of LTL formulas over AP is inductively defined as follows:

– *true* is an LTL formula;

– if $p \in AP$ then p is an LTL formula;
– if φ and ψ are LTL formulas then $\neg\psi$, $\varphi \vee \psi$, $X\psi$, and $\varphi U \psi$ are LTL formulas.

Additional operators can be derived in the standard way: $\varphi \wedge \psi = \neg(\neg\varphi \vee \neg\psi)$, $\varphi \Rightarrow \psi = \neg\varphi \vee \psi$, $F\varphi$ (or $\Diamond\varphi$) $= true\, U \varphi$, and $G\varphi$ (or $\Box\varphi$) $= \neg(true\, U \neg\varphi)$.

Let $\Sigma = 2^{AP}$ be the set of all possible subsets of AP; if $p \in AP$ and $a \in \Sigma$, then p holds in a iff $p \in a$. An LTL model is an infinite trace $w \in \Sigma^\omega$; $w(i)$ denotes the element $a \in \Sigma$ at position i in trace w; more formally, if $w = aw'$, then $w(0) = a$, and $w(i) = w'(i-1)$ if $i > 0$.

The semantics of a formula φ depends on the satisfaction relation $w, i \vDash \varphi$ (w satisfies φ in i) defined as follows:

– $w, i \vDash p$ iff $p \in w(i)$;
– $w, i \vDash \neg\phi$ iff $w, i \nvDash \phi$;
– $w, i \vDash \varphi \vee \psi$ iff $w, i \vDash \varphi$ or $w, i \vDash \psi$;
– $w, i \vDash X\varphi$ iff $w, i+1 \vDash \varphi$ (next operator);
– $w, i \vDash \varphi U \psi$ iff $\exists j \geq 0 \;\; w, j \vDash \psi$ and $\forall 0 \leq k < j \;\; w, k \vDash \varphi$ (until operator).

Finally, $w \vDash \varphi$ (w satisfies φ) holds iff $w, 0 \vDash \varphi$ holds.

We recall that the set of all models of LTL formulas is the language of star-free ω-regular languages over Σ [7].

In order to encode an LTL formula into an equivalent trace expression we exploit the result stating that an LTL formula can be translated into an equivalent non deterministic Büchi automaton [3,14].

Non Deterministic Büchi Automata. A Büchi automaton is a type of ω-automaton which extends a finite automaton to infinite inputs. It accepts an infinite input sequence if there exists a run of the automaton that visits (at least) one of the final states infinitely often.

A (non deterministic) Büchi automaton (NBA) is a tuple $(\Sigma, Q, Q_0, \delta, F)$, where

– Σ is a finite alphabet;
– Q is a finite non-empty set of states;
– $Q_0 \subseteq Q$ is a set of initial states;
– $\delta{:}Q \times \Sigma \to 2^Q$ is a transition function;
– $F \subseteq Q$ is a set of accepting states.

A run of an automaton $(\Sigma, Q, Q_0, \delta, F)$ on a word $w \in \Sigma^\omega$ is an infinite trace $\rho = q_0 w(0) q_1 w(1) q_2 \ldots$, s.t. $q_0 \in Q_0$, and for all $i \geq 0$ $q_{i+1} \in \delta(q_i, w(i))$. A run ρ is called accepting iff $Inf(\rho) \cap F \neq \emptyset$, where $Inf(\rho)$ denotes the states visited infinitely often.

LTL_3. LTL_3 is a three-valued semantics [3] for LTL formulas, devised to adapt the standard semantics to RV, to correctly consider the limitation that at runtime only finite traces can be checked.

Given a finite trace $\sigma \in \Sigma^*$ of length $|\sigma| = n$, a continuation of σ is an infinite trace $w \in \Sigma^\omega$ s.t. for all $0 \leq i < n$ $w(i) = \sigma(i)$.

Given a finite trace $\sigma \in \Sigma^*$, and an LTL formula φ, the LTL$_3$ semantics of φ, denoted by $\sigma \vDash_3 \varphi$, is defined as follows:

$$\sigma \vDash_3 \varphi = \begin{cases} \top \text{ iff } w \vDash \varphi \text{ for all continuations } w \text{ of } \sigma \\ \bot \text{ iff } w \nvDash \varphi \text{ for all continuations } w \text{ of } \sigma \\ ? \text{ iff neither of the two conditions above holds} \end{cases}$$

As an example, let us consider the formula $\varphi = pUq$, where $p, q \in AP$; according to the definition above, $\{p\}\{p\}\{q\} \vDash_3 \varphi = \top$, that is, φ is satisfied by the finite trace $\{p\}\{p\}\{q\}$, and monitoring succeeds; $\{p\}\{p\}\emptyset \vDash_3 \varphi = \bot$, that is, φ is not satisfied by the finite trace $\{p\}\{p\}\emptyset$, and monitoring fails; finally, $\{p\}\{p\}\{p\} \vDash_3 \varphi =?$, that is, at this stage monitoring is inconclusive, and the monitor has to keep monitoring the property expressed by φ. Assuming that $AP = \{p, q\}$, the LTL$_3$ semantics of pUq corresponds to the finite state machine (FSM) defined in Fig. 3, which fully determines the expected behavior of a monitor for the RV of pUq.

More in general, for all LTL formulas φ, it is possible to build an FSM which is a deterministic finite automaton (DFA) where the alphabet is Σ (that is, 2^{AP}), all states are final, each state returns either $\top$ (successful), or $\bot$ (failure), or ? (inconclusive), and the behavior of the FSM respects the LTL$_3$ semantics of φ: for all finite traces $\sigma \in \Sigma^*$, the FSM accepts σ with final state that returns $v \in \{\top, \bot, ?\}$ iff $\sigma \vDash_3 \varphi = v$.

The sequence of steps required to generate from an LTL formula φ an FSM that respects the LTL$_3$ semantics of φ [3] is summarized in Fig. 4.

For each LTL formula φ and $\neg\varphi$ (1), the equivalent NBAs $\mathcal{A}^{\varphi}$, and $\mathcal{A}^{\neg\varphi}$ are built (2), all states that generate a non empty language are identified (3) and made final and the NBAs are transformed into the corresponding NFAs $\hat{\mathcal{A}}^{\varphi}$, and $\hat{\mathcal{A}}^{\neg\varphi}$ (4), and, then, into the equivalent DFAs $\tilde{\mathcal{A}}^{\varphi}$ and $\tilde{\mathcal{A}}^{\neg\varphi}$ (5). Finally, the product of $\tilde{\mathcal{A}}^{\varphi}$ and $\tilde{\mathcal{A}}^{\neg\varphi}$ is computed, and from it the final FSM $\mathcal{M}^{\varphi}$ is derived by minimization, and by classifying the states in the following way: (q, q') returns

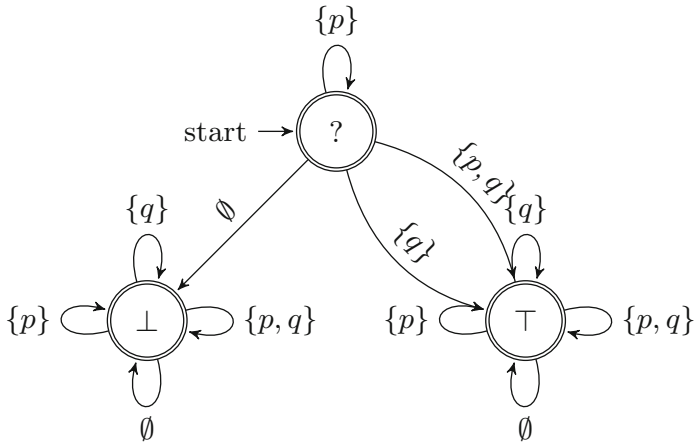

Fig. 3. FSM of the monitor for pUq, with $AP = \{p, q\}$

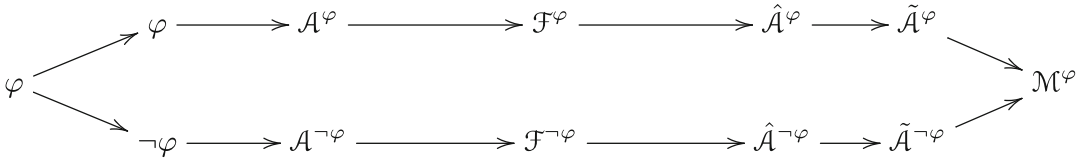

Fig. 4. Steps required to generate an FSM from an LTL formula φ

$\top$ iff q' is not final in $\tilde{\mathcal{A}}^{\neg\varphi}$, $\bot$ iff q is not final in $\tilde{\mathcal{A}}^{\varphi}$, and ? if both q and q' are final in $\tilde{\mathcal{A}}^{\varphi}$, and $\tilde{\mathcal{A}}^{\neg\varphi}$, respectively.

4.2 Comparing Trace Expressions with LTL

We have shown that LTL formulas as $p\,U\,q$ cannot be fully verified at runtime, therefore a three-valued semantics LTL_3 has been introduced. To be able to compare LTL formulas with trace expressions, the same three-valued semantics is considered for trace expressions as well.

Given a finite trace $\sigma \in \Sigma^*$ of length $|\sigma| = n$, a continuation of σ is an finite or infinite trace $u \in \Sigma^* \cup \Sigma^\omega$ s.t. for all $0 \leq i < n$ $u(i) = \sigma(i)$.

The three-valued semantics of a trace expression τ is defined as follows:

$$\sigma \in [\![\tau]\!]_3 = \begin{cases} \top \text{ iff } u \in [\![\tau]\!] \text{ for all continuations } u \text{ of } \sigma \\ \bot \text{ iff } u \notin [\![\tau]\!] \text{ for all continuations } u \text{ of } \sigma \\ ? \text{ iff neither of the two conditions above holds} \end{cases}$$

Let us consider again the formula $\varphi = p\,U\,q$; if we assume that each atomic predicate in AP has a corresponding event type denoted in the same way, then the closest trace expression τ into which φ can be translated is defined by $T = p{:}T \vee q{:}1$, where 1 is the derivable constant introduced in Sect. 3 denoting all possible traces. If we consider the standard semantics we have that, since $\{p\}$ is an event that satisfies p, $\{p\}^\omega \in [\![\tau]\!]$, but $\{p\}^\omega \not\models \varphi$. However, when considering the three-valued semantics we have that for all $v \in \{\top, \bot, ?\}$ and $\sigma \in \Sigma^*$, $\sigma \vDash \varphi = v$ iff $\sigma \in [\![\tau]\!]_3 = v$. In particular, for all $n \geq 0$, $\{p\}^n \vDash \varphi = ?$ and $\{p\}^n \in [\![\tau]\!]_3 = ?$.

To translate an LTL formula φ into a trace expression τ s.t. the three-valued semantics is preserved, we exploit the result presented in Sect. 4.1. First, φ is translated into an equivalent FSM $\mathcal{M}^\varphi$, then $\mathcal{M}^\varphi$ is translated into an equivalent contractive and deterministic trace expression τ^φ. The latter translation is defined as follows:

- if the initial state returns $\top$, then φ is a tautology, and the corresponding trace expression is the constant 1;
- if the initial state returns $\bot$, then φ is a unsatisfiable, and the corresponding trace expression is the constant 0;
- if the initial state returns ?, then the corresponding trace expression is defined by a finite set of equations $X_1 = \tau_1, \ldots, X_n = \tau_n$, where n is the number of states in $\mathcal{M}^\varphi$ that return ?, each of such states is associated with a distinct

variable X_i, X_1 is the variable associated with the initial state which corresponds to the whole trace expression τ^{φ}.

The expressions τ_i are defined as follows: let k be the number of states $q_1, \ldots, q_k$ that do not return $\bot$ for which there exists an incoming edge, labeled with the element $a_i \in 2^{AP}$, from the node associated with X_i; we know that $k > 0$, because the node associated with X_i returns ?. Then $\tau_i = a_1{:}f(q_1) \vee \ldots \vee a_k{:}f(q_k)$, where $f(q)$ is defined as follows: if q returns $\top$, then $f(q) = 1$, otherwise (that is, q returns ?), $f(q) = X_q$ (that is, the variable uniquely associated with q is returned).

Since all variables in the expressions $\tau_1, \ldots, \tau_n$ are guarded by the prefix operator, τ^{φ} is contractive; furthermore, it is deterministic because $\mathcal{M}^{\varphi}$ is deterministic.

Theorem 1. *Let $\mathcal{M}^{\varphi}$ be the FSM equivalent to φ generated by the procedure described in Sect. 4.1. Then, the trace expression τ^{φ} generated from $\mathcal{M}^{\varphi}$ as specified in Sect. 4.2 preserves the semantics of $\mathcal{M}^{\varphi}$: for all $\sigma \in \Sigma^*$ $\mathcal{M}^{\varphi}$ accepts σ with output $v \in \{\top, \bot, ?\}$ iff $\sigma \in [\![\tau^{\varphi}]\!]_3 = v$.*

Proof Sketch: the proof proceeds by induction on the length of σ. The cases where the initial state of the FSM returns $\top$ or $\bot$ are immediate to be proved. The proof when the initial state returns ? is based on the fact that, in this case, by construction $[\![\tau^{\varphi}]\!] \neq \emptyset$ and there always exists a trace u s.t. $u \notin [\![\tau^{\varphi}]\!]$, therefore $\epsilon \in [\![\tau^{\varphi}]\!]_3 = ?$. □

In Sect. 3.3 we have shown a trace expression τ that specifies a non context free language of traces (when only finite traces are considered). More formally, $\sigma \in [\![\tau \cdot 1]\!]_3 = \top$ iff $\sigma \in \{a^n b^n c^n \mid n \geq 0\}$.

This means that for RV (that is, when the three-values semantics is considered) trace expressions are strictly more expressive than LTL logic, since the LTL logic is less expressive than ω-regular languages.

5 Related Work

In this section we briefly survey work related to runtime verification, and to formalisms, other than LTL, for specifying event traces.

Global Types and Multi-party Sessions. Though trace expressions and global types [5] are rather similar (indeed, global types correspond to trace expressions without the concatenation and the intersection operators), the aim of trace expressions diverges from that of Castagna et al.'s behavioral types for many reasons:

– trace expressions are not intended to be used for annotating and statically checking programs, but rather, for specifying properties that have to be verified at runtime;

– while Castagna et al.'s types are expressly designed for describing multiparty interactions between distributed components, trace expressions are meant as a more general formalism which can be used for runtime verification of different kinds of properties and systems;
– finally, trace expressions have a coinductive, rather than inductive, semantics, hence they can denote sets containing infinite traces; this is important for being able to verify systems that must not terminate.

Object-Oriented Languages. In the context of runtime verification of object-oriented languages, there exist several formalisms for specifying valid or invalid traces of method invocations, as done in the stack objects example in Sect. 3.1.

Program Query Language (PQL) [13] allows developers to express a large class of application specific code patterns. PQL is more expressive than context-free languages, since its class of languages is that of the closure of context-free languages combined with intersection, hence, the formalism seems to be as expressive as trace expressions. However, no formal semantics is defined for PQL, and it is not clear whether PQL queries can denote infinite traces.

The jassda [4] framework and tool enable runtime checking of Java programs against a CSP-like specification. Like in trace expressions, the trace semantics of a process is defined by collecting all event sequences that are possible with respect to the operational semantics. Processes are built with operators similar to those of trace expressions, except for concatenation and intersection, which are not supported by jassda.

SAGA [10] is a tool for runtime verification of properties of Java programs specified with attribute grammars. The implementation is based on four different components: a state-based assertion checker, a parser generator, a debugger and a general tool for meta-programming. The tool is extremely powerful and has been successfully applied to an industrial case from the e-commerce with multi-threaded Java. The main difference w.r.t. our approach is that SAGA has been developed for runtime checking of a combination of protocol- and data-oriented properties of object-oriented programs, whereas, at the moment, trace expressions have been successfully employed for runtime verification of multiagent systems.

6 Conclusion

Trace expressions are a compact and expressive formalism that has been used for RV of interaction protocols in multiagent systems.

In this paper we have formally compared trace expressions with LTL, a formalism widely adopted in RV. To this aim we have employed the three-valued semantics [3] proposed for LTL in the context of RV, and we have proved that for the purpose of RV, trace expressions are strictly more expressive than LTL: every LTL formula can be encoded into a trace expression which preserves its three-valued semantics, but the opposite property does not hold, since trace expressions are able to specify context-free and non context-free languages.

Anyway, the benefits of trace expressions over LTL in the context of runtime verification needs to be studied on the basis of an implementation and case studies.

Another interesting subject for further investigation would consists in the study of the class of language that is covered by trace expressions, and by contractive and/or deterministic trace expressions.

References

1. Ancona, D., Briola, D., Ferrando, A., Mascardi, V.: Global protocols as first class entities for self-adaptive agents. In: Proceedings of the International Conference on Autonomous Agents and Multiagent Systems, AAMAS 2015, pp. 1019–1029 (2015)
2. Ancona, D., Drossopoulou, S., Mascardi, V.: Automatic generation of self-monitoring MASs from multiparty global session types in Jason. In: Baldoni, M., Dennis, L., Mascardi, V., Vasconcelos, W. (eds.) DALT 2012. LNCS, vol. 7784, pp. 76–95. Springer, Heidelberg (2013)
3. Bauer, A., Leucker, M., Schallhart, C.: Runtime verification for LTL and TLTL. ACM Trans. Softw. Eng. Methodol. (TOSEM) **20**, 1–64 (2009)
4. Brörkens, M., Möller, M.: Dynamic event generation for runtime checking using the JDI. Electr. Notes Theor. Comput. Sci. **70**(4), 21–35 (2002)
5. Castagna, G., Dezani-Ciancaglini, M., Padovani, L.: On global types and multi-party session. Logical Methods Comput. Sci. **8**(1), 1–45 (2012)
6. Chen, F., Rosu, G.: Mop: an efficient and generic runtime verification framework. In: OOPSLA 2007, pp. 569–588 (2007)
7. Cohen, J., Perrin, D., Pin, J.-E.: On the expressive power of temporal logic. J. Comput. Syst. Sci. **46**, 271–294 (1993)
8. Courcelle, B.: Fundamental properties of infinite trees. Theoret. Comput. Sci. **25**, 95–169 (1983)
9. Ancona D., Barbieri, M., Mascardi, V.: Constrained global types for dynamic checking of protocol conformance in multi-agent systems. In: Proceedings of the 28th Annual ACM Symposium on Applied Computing, SAC 2013, pp. 1377–1379 (2013)
10. de Boer, F.S., de Gouw, S.: Combining monitoring with run-time assertion checking. In: Bernardo, M., Damiani, F., Hähnle, R., Johnsen, E.B., Schaefer, I. (eds.) SFM 2014. LNCS, vol. 8483, pp. 217–262. Springer, Heidelberg (2014)
11. Deniélou, P.-M., Yoshida, N.: Multiparty session types meet communicating automata. In: Seidl, H. (ed.) ESOP 2012. LNCS, vol. 7211, pp. 194–213. Springer, Heidelberg (2012)
12. Luo, Q., Zhang, Y., Lee, C., Jin, D., Meredith, P.O.N., Şerbănuţă, T.F., Roşu, G.: RV-Monitor: efficient parametric runtime verification with simultaneous properties. In: Bonakdarpour, B., Smolka, S.A. (eds.) RV 2014. LNCS, vol. 8734, pp. 285–300. Springer, Heidelberg (2014)
13. Martin, M.C., Livshits, V.B., Lam, M.S.: Finding application errors and security flaws using PQL: a program query language. OOPSLA **2005**, 365–383 (2005)
14. Sistla, A.P., Vardi, M.Y., Wolper, P.: The complementation problem for büchi automata with appplications to temporal logic. Theor. Comput. Sci. **49**, 217–237 (1987)