Parametric Trace Expressions for Runtime Verification of Java-Like Programs

Davide Ancona Angelo Ferrando Luca Franceschini Viviana Mascardi

DIBRIS, University of Genova, Italy

davide.ancona@unige.it, angelo.ferrando@dibris.unige.it, luca.franceschini@dibris.unige.it, viviana.mascardi@unige.it

Abstract

Parametric trace expressions are a formalism expressly designed for parametric runtime verification (RV) which has been introduced and successfully employed in the context of runtime monitoring of multiagent systems.

Trace expressions are built on the general notion of event type, which allows them to be adopted in different contexts. In this paper we show how trace expressions can be used for conveniently specifying the expected behavior of a Java-like program to be monitored at runtime.

Furthermore, we investigate the basic properties of the primitive operators on which trace expressions are coinductively defined in terms of a labeled transition system; this provides a basis for formal reasoning about equivalence of trace expressions and for adopting useful optimization techniques to speed up runtime verification.

CCS Concepts • **Theory of computation** → **Program verification**; *Operational semantics*

Keywords runtime monitoring; trace expressions; labeled transition systems; object-oriented languages

1. Introduction

Trace expressions have been introduced (Ancona et al. 2012, 2016a) as a formalism based on behavioral types initially called global session types, to formally specify interaction protocols in multiagent systems for monitoring the correct behavior of agents at runtime. They have been successfully adopted in practice for widespread multiagent system platforms (Ancona et al. 2014; Briola et al. 2014).

The semantics of a trace expression is the set of all possible event traces that can be correctly observed during the execution of a program; this is formalized in an operational way through a labeled transition system whose rewriting rules can be directly turned into an algorithm for event trace recognition. Thanks to this, runtime verification of the correct behavior of a system specified by a trace expression τ can be performed by a monitor directly driven by the reduction semantics of τ.

Trace expressions are defined on top of the notion of *event type*, to favor their use in different languages, systems, and applications, and for dynamically monitoring or enforcing different kinds of properties. In this paper we provide some examples showing how trace expressions can be suitably employed for specifying the expected behavior of a Java-like program, and for monitoring it at runtime. Trace expressions can be used to specify and verify conformance to a given protocol, which can be useful in the context of object-oriented systems (Vasconcelos and Ravara 2017).

Recently, trace expressions have been extended (Ancona et al. 2017) to allow specifications to be *parametric* (Luo et al. 2014) in data that can be captured and monitored only at runtime; thanks to this extension, specifications which, for instance, depend on the values exchanged by objects through methods, or on the dynamically evolving collection of objects or resources available at runtime, can be suitably modeled, and the number of correct programming patterns that can be specified is significantly enlarged.

Such an extension poses some challenges for what concerns performance: runtime monitoring is effective only if it does not undermine the efficiency of the monitored program; ideally, monitoring exhibits a time complexity linear in the length of the analyzed event trace. Preliminary experiments (Ancona et al. 2017) have shown that in some cases time complexity can be kept linear also for parametric trace expressions by suitably simplifying the trace expressions dynamically generated during the rewriting steps. This calls for a better understanding of the equational theory of trace expressions; in this paper we investigate and formally prove the basic properties of the primitive operators of trace expressions, with the main aim of formally reasoning about the equivalence between trace expressions and adopting useful optimization techniques to speed up RV.

FTFJP'17, June 18-23, 2017, Barcelona , Spain

ISBN 978-1-4503-5098-3/17/06. . . $15.00

DOI: http://dx.doi.org/10.1145/3103111.3104037

The rest of the paper consists of the following sections: Section 2 is a gentle introduction to trace expressions, and shows some examples of their use for RV of Java-like languages; the full formalization of trace expressions can be found in the subsequent Section 3. Section 4 contains the main claims about the properties of the operators of trace expressions (the detailed proofs can be found in the Appendix), while Section 5 concludes with some remarks.

2. A gentle introduction to trace expressions

Trace expressions are defined on top of the notions of *event* and *event type*; a trace expression denotes a set of possibly infinite traces of events belonging to a fixed universe of events $\mathcal{E}$ modeling the observations which a specific monitoring activity is focused on; then, an event trace $\bar{e}$ is an element of $\mathcal{E}^* \cup \mathcal{E}^\omega$. Although at runtime only finite prefixes can be monitored, to correctly specify the behavior of non-terminating programs trace expressions have to include also infinite traces; this significantly affects the model and the corresponding equational theory (see Section 4): for instance, to specify the correct behavior of a program P consisting of a first task T_1 followed by another task T_2, one has to consider the possibility that task T_1 does not terminate, and, hence, that in this case the event trace of P coincides with the infinite trace generated by T_1.

As an initial simple example, we may want to monitor the sequence of methods invoked on a specific print-writer object, that is, an instance of `java.io.PrintWriter`; in this case, events include the names of all invocable methods ($print$, $close$, etc.) and the distinguished event new corresponding to the fact that the object has been created by invoking any of the available constructors, and its underlying stream has been opened; since in this initial example the monitoring activity we want to perform is very simple, events are not structured and do not carry values.

For this object we are interested in monitoring that a correct sequence of methods is invoked on it, according to the following requirements:

- no method can be invoked before the creation of the object;
- no method can be invoked after method `close` has been invoked on it.

Since trace expressions are built on top of event types, an appropriate language $\mathcal{ET}$ of event types has to be defined; to make the formalism more flexible, there is no a priori fixed language of event types, and only a minimal set of assumptions discussed in the next sections must be satisfied.

In general, event types are allowed to contain variables ranging over the set of values which events can carry; however, for this initial example we consider only *ground* event types. The semantics of a ground event type ϑ is a set of events, that is, a subset of $\mathcal{E}$. For this example only three constant event types are needed:

- new_{ty}, whose semantics is $\{new\}$;
- $close_{ty}$, whose semantics is $\{close\}$;
- no_close_{ty}, whose semantics is $\mathcal{E} \setminus \{new, close\}$; this type denotes all events corresponding to invocation on the print-writer of any method other than `close`.

According to the definition above, new_{ty} and $close_{ty}$ are singleton event types, whereas no_close_{ty} denotes more events.

Given the event types above, we can now define a trace expression specifying the correct use of a print-writer object.

$$PW = \epsilon \vee (new_{ty} : U)$$
$$U = (close_{ty} : \epsilon) \vee (no\_close_{ty} : U)$$

Trace expressions can be recursively defined, hence are usually defined by a finite set of syntactic equations, as it is customary for regular (a.k.a. rational) terms, that is, terms corresponding with trees that are allowed to have infinite depth, but have always a finite set of subtrees (Courcelle 1983). The meta-variable PW just assigns a name to the main trace expression for convenience, whereas the use of meta-variable U is necessary to define recursively the associated trace expression. In this example, besides the above defined event types, we use the following trace expression operators:

- ϵ is a constant denoting the singleton set containing the empty trace; this is used whenever no other events are expected to occur;
- $\vee$ is a binary operator which takes two trace expressions and builds a new trace expression corresponding to set theoretic union (that is, logical disjunction) of sets of event traces;
- : is a binary operator which takes an event type ϑ and a trace expression τ and builds the new trace expressions $\vartheta : \tau$ whose semantics is the set of event traces obtained by prefixing the event traces denoted by τ with the events matching ϑ.

Considering the informal semantics given above, it is possible to derive the following semantics for the trace expression denoted by PW: either no event occurs at all, or a new print-writer pw is created, then

- either an infinite sequence of methods different from `close` is invoked on pw;
- or a finite (possibly empty) sequence of methods different from `close` is invoked on pw, then method `close` is invoked on pw, and, subsequently, no other methods are invoked on pw.

For this simple example a suitable regular expression would apparently work as well for specifying the behavior of a print-writer; however, regular expressions fail to model infinite traces, therefore an ω-automaton should be required instead. Furthermore, trace expressions support three additional binary operators, namely, concatenation, intersection,

and shuffle, and it can be proved that for the purpose of runtime monitoring trace expressions are more expressive than ω-automata and context-free grammars (Ancona et al. 2016b).

We now turn to considering a more complex example involving parametricity, and, thus, non-ground event types, and other trace expression operators. The trace expression defined above and denoted by PW specifies the behavior of a single print-writer object, therefore its usefulness is limited. We show how it is possible to specify and monitor the behavior of a whole community of print-writer objects with a parametric trace expression. To this aim we first need to extend the language of events and event types so that events can indicate the object for which the event has occurred by carrying its corresponding object identifier.

- $new_{ty}(oid)$, with semantics $\{new(oid)\}$;
- $close_{ty}(oid)$, with semantics $\{close(oid)\}$;
- $no\_close_{ty}(oid)$, with semantics $\{n(oid) \mid n \in PrintWriterMeths, n \neq close\}$, where $PrintWriterMeths$ denotes the set of all method names that are defined for the `PrintWriter` class.

For instance, the event $new(oid)$ corresponds to the fact that a new print-writer has been created and opened with object identifier oid, whereas $close(oid)$ describes the fact that method `close` has been invoked on print-writer identified by oid. For simplicity we assume that object identifiers are generated during monitoring to guarantee their uniqueness even in the presence of object deallocation by means of the garbage collector; hence, the notion of object identifier is more abstract than the implementation dependent notion of object reference.

We also need to introduce the new event type $any_{ty}(oid)$, with semantics $\{n(oid) \mid n \in PrintWriterMeths \cup \{new\}\}$; $any_{ty}(oid)$ includes all events involving the print-writer with identifier oid.

We can now define a parametric trace expression specifying the behavior of a dynamically evolving community of print-writer objects.

$$PPW = \epsilon \vee <\mathsf{oid} ; new_{ty}(\mathsf{oid}) : (cond(any_{ty}(\mathsf{oid}), PU, PPW))>$$
$$PU = (close_{ty}(\mathsf{oid}) : \epsilon) \vee (no\_close_{ty}(\mathsf{oid}) : PU)$$

The trace expression uses the derived operator[1] $cond$; the construct $cond(any_{ty}(\mathsf{oid}), PU, PPW)$ has the following intuitive semantics: if an event e is matched by $any_{ty}(\mathsf{oid})$ (that is, any operation has been performed on print-writer oid), then e must belong to a trace specified by PU, otherwise it must belong to a trace specified by PPW (thus e must correspond to the creation of a new print-writer).

Even most relevant for our purposes, the trace expression uses the new construct $<\mathsf{oid} ; \tau>$ which binds the free occurrences of variable oid in $\tau = new_{ty}(\mathsf{oid}) : (cond(any_{ty}(\mathsf{oid}), PU, PPW))$, where oid is expected to hold an object identifier. In this specific case, all free occurrences of oid in τ are substituted with the value oid iff the event type $new_{ty}(\mathsf{oid})$ matches the event $new(oid)$. The trace expression PPW does not contain free occurrences of oid, because all uses of oid are guarded by the binder construct $<\mathsf{oid} ; \tau>$, therefore the only free occurrences of oid in τ are those in PU, and the one contained in the leftmost occurrence of the event type $new_{ty}(\mathsf{oid})$; more precisely, the recursively defined trace expression PPW contains infinitely many occurrences of $new_{ty}(\mathsf{oid})$, and, hence, of oid, but all such variables are bound to different binders. This allows the different occurrences of $new_{ty}(\mathsf{oid})$ to match events with different object identifiers.

[1] More details can be found in the next section.

3. Syntax and semantics

The semantics of event types is specified by the function $match$; since the language of event types is not fixed a priori, we assume that function $match$ is given. If ϑ is an event type, possibly containing free variables, then $match(e, \vartheta) = \sigma$ iff event e matches event type ϑ with computed substitution σ; σ is a finite partial map where the domain $dom(\sigma)$ is assumed to coincide with the set of variables in ϑ, while the codomain is a universe of values $\mathcal{V}$. We say that $match(e, \vartheta)$ *succeeds* iff there exists σ s.t. $match(e, \vartheta) = \sigma$, otherwise we say that $match(e, \vartheta)$ *fails*.

The substitution with the empty domain is denoted by $\emptyset$. The equality $\sigma = \sigma_1 \cup \sigma_2$ holds iff $dom(\sigma) = dom(\sigma_1) \cup dom(\sigma_2)$, and for all $\mathsf{x} \in dom(\sigma)$, $\sigma(\mathsf{x}) = \sigma_1(\mathsf{x})$ if $\mathsf{x} \in dom(\sigma_1)$, and $\sigma(\mathsf{x}) = \sigma_2(\mathsf{x})$ if $\mathsf{x} \in dom(\sigma_2)$ (hence, σ_1 and σ_2 must coincide on $dom(\sigma_1) \cap dom(\sigma_2)$). The notation $\sigma_{\setminus \mathsf{x}}$ denotes the substitution where x is removed from its domain: $\sigma_{\setminus \mathsf{x}} = \sigma'$ iff $dom(\sigma') = dom(\sigma) \setminus \{\mathsf{x}\}$ and for all $\mathsf{x} \in dom(\sigma')$, $\sigma'(\mathsf{x}) = \sigma(\mathsf{x})$. The notation $\sigma\vartheta$ denotes the event type obtained from ϑ by substituting all occurrences of $\mathsf{x} \in dom(\sigma)$ in ϑ with $\sigma(\mathsf{x})$.

A parametric trace expression τ is a regular term built on top of the following operators:

- ϵ (empty trace)
- $\vartheta : \tau$ (*prefix*)
- $\tau_1 \cdot \tau_2$ (*concatenation*)
- $\tau_1 \wedge \tau_2$ (*intersection*)
- $\tau_1 \vee \tau_2$ (*union*)
- $\tau_1 \mid \tau_2$ (*shuffle*, a.k.a. *interleaving*)
- $<\mathsf{x} ; \tau>$ (*binder*)

The trace expression $\sigma\tau$ obtained from τ by substituting all free occurrences of $\mathsf{x} \in dom(\sigma)$ in τ with $\sigma(\mathsf{x})$, is coinductively defined as follows:

$$\sigma(\vartheta : \tau) = (\sigma\vartheta) : (\sigma\tau)$$
$$\sigma(\tau_1 \mathbin{op} \tau_2) = (\sigma\tau_1) \mathbin{op} (\sigma\tau_2) \text{ for } op \in \{\vee, \wedge, \mid, \cdot\}$$
$$\sigma(<\mathsf{x} ; \tau>) = <\mathsf{x} ; \sigma_{\setminus \mathsf{x}}\tau>$$

$$\text{(main)}\ \frac{\tau \overset{e}{\rightarrowtail} \tau';\emptyset}{\tau \overset{e}{\rightarrow} \tau'} \qquad \text{(prefix)}\ \frac{}{\vartheta : \tau \overset{e}{\rightarrowtail} \tau;\sigma}\ \sigma{=}match(e,\vartheta) \qquad \text{(or-l)}\ \frac{\tau_1 \overset{e}{\rightarrowtail} \tau_1';\sigma}{\tau_1 \vee \tau_2 \overset{e}{\rightarrowtail} \tau_1';\sigma} \qquad \text{(or-r)}\ \frac{\tau_2 \overset{e}{\rightarrowtail} \tau_2';\sigma}{\tau_1 \vee \tau_2 \overset{e}{\rightarrowtail} \tau_2';\sigma}$$

$$\text{(and)}\ \frac{\tau_1 \overset{e}{\rightarrowtail} \tau_1';\sigma_1 \quad \tau_2 \overset{e}{\rightarrowtail} \tau_2';\sigma_2}{\tau_1 \wedge \tau_2 \overset{e}{\rightarrowtail} \tau_1' \wedge \tau_2';\sigma}\ \sigma{=}\sigma_1 \cup \sigma_2 \qquad \text{(shuffle-l)}\ \frac{\tau_1 \overset{e}{\rightarrowtail} \tau_1';\sigma}{\tau_1 \mid \tau_2 \overset{e}{\rightarrowtail} \tau_1' \mid \tau_2;\sigma} \qquad \text{(shuffle-r)}\ \frac{\tau_2 \overset{e}{\rightarrowtail} \tau_2';\sigma}{\tau_1 \mid \tau_2 \overset{e}{\rightarrowtail} \tau_1 \mid \tau_2';\sigma}$$

$$\text{(cat-l)}\ \frac{\tau_1 \overset{e}{\rightarrowtail} \tau_1';\sigma}{\tau_1 \cdot \tau_2 \overset{e}{\rightarrowtail} \tau_1' \cdot \tau_2;\sigma} \qquad \text{(cat-r)}\ \frac{\tau_2 \overset{e}{\rightarrowtail} \tau_2';\sigma}{\tau_1 \cdot \tau_2 \overset{e}{\rightarrowtail} \tau_2';\sigma}\ \epsilon(\tau_1) \qquad \text{(var-t)}\ \frac{\tau \overset{e}{\rightarrowtail} \tau';\sigma}{<\mathsf{x}\,;\tau> \overset{e}{\rightarrowtail} \sigma\tau';\sigma_{\setminus \mathsf{x}}}\ \mathsf{x}{\in}dom(\sigma)$$

$$\text{(var-f)}\ \frac{\tau \overset{e}{\rightarrowtail} \tau';\sigma}{<\mathsf{x}\,;\tau> \overset{e}{\rightarrowtail} <\mathsf{x}\,;\tau'>;\sigma}\ \mathsf{x}{\notin}dom(\sigma) \qquad \text{(ϵ-empty)}\ \frac{}{\epsilon(\epsilon)} \qquad \text{(ϵ-var)}\ \frac{\epsilon(\tau)}{\epsilon(<\mathsf{x}\,;\tau>)} \qquad \text{(ϵ-or-l)}\ \frac{\epsilon(\tau_1)}{\epsilon(\tau_1 \vee \tau_2)}$$

$$\text{(ϵ-or-r)}\ \frac{\epsilon(\tau_2)}{\epsilon(\tau_1 \vee \tau_2)} \qquad \text{(ϵ-others)}\ \frac{\epsilon(\tau_1) \quad \epsilon(\tau_2)}{\epsilon(\tau_1\, op\, \tau_2)}\ op{\in}\{|,\cdot,\wedge\}$$

Figure 1. Transition system for parametric trace expressions

The operational semantics of parametric trace expressions is defined by the labeled transition system in Figure 1. The transition relation $\tau \overset{e}{\rightarrow} \tau'$ is defined by rule (main) and depends on the auxiliary relation $\tau \overset{e}{\rightarrowtail} \tau';\sigma$, which returns the substitution σ generated during the transition step; in this way the definition in Figure 1 can be easily turned into an algorithm for event trace recognition; a whole trace expression cannot contain free variables, and the main transition is fired only if the computed substitution is empty.

In rule (prefix) a transition step is possible only when the current event e matches the event type ϑ; the computed substitution is returned by the $match$ function.

Rules for union, shuffle, and concatenation are straightforward.

In rule (and) the side condition requires that the substitutions σ_1 and σ_2 computed for the two subexpressions must coincide on the intersection of their domains; the final substitution σ is obtained by merging σ_1 and σ_2.

Two rules are required for the $<\mathsf{x}\,;\tau>$ construct. (var-t) is applied when variable x is contained in the domain of the computed substitution σ; σ is applied to the trace expression τ' in which τ rewrites to, the binder is removed and x is removed from the domain of the computed substitution ($\sigma_{\setminus \mathsf{x}}$). (var-f) is applied when variable x is not contained in the domain of the computed substitution σ; the binder is not removed, and the computed substitution coincides with σ.

The auxiliary predicate $\epsilon(\_)$ checks whether the semantics of a trace expression contains the empty trace, and is required in the side condition of rule (cat-r) for $\tau_1 \cdot \tau_2$: an event trace is allowed to continue with τ_2 only if τ_1 is allowed to terminate, that is, the semantics of τ_1 contains the empty trace. Rules for $\epsilon(\_)$ are straightforward.

The semantics $[\![\tau]\!]$ of a trace expression τ is defined in terms of the transition relation $\overset{e}{\rightarrow}$, and the predicate $\epsilon(\_)$. Since $[\![\tau]\!]$ may contain infinite traces, its definition is coinductive.

Def. 3.1. *For all possibly infinite event traces $\bar{e} \in \mathcal{E}^* \cup \mathcal{E}^\omega$ and trace expressions τ, $\bar{e} \in [\![\tau]\!]$ is coinductively defined as follows:*

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

Derived operators. Several useful operators can be derived from the primitive operators defined above.

We assume that the language of event types contains the universe event type any and the empty event type $none$, that is, for all $e \in \mathcal{E}$, $match(e, any) = \emptyset$, and there is no substitution σ, s.t. $match(e, none) = \sigma$; then the constant 1 denoting the set of all possible traces over $\mathcal{E}$ can be defined by the trace expression $T = \epsilon \vee any : T$, and the constant 0 denoting the empty set of traces can be defined by $none : \epsilon$.

Another useful derived operator is the filter operator.

An event trace $\bar{e}$ belongs to $[\![\vartheta \gg \tau]\!]$ iff by removing from it all events that do not match ϑ we get a subtrace $\bar{e}'$ s.t. $\bar{e}' \in [\![\tau]\!]$. The $\vartheta \gg \tau$ construct can be derived in the following way, if we assume that event types are closed by complementation: $(T_t \wedge \tau) \mid T_f$, where $T_t = \epsilon \vee \vartheta : T_t$, $T_f = \epsilon \vee \overline{\vartheta} : T_f$, and $\overline{\vartheta}$ is the complement event type of ϑ, that is, for all $e \in \mathcal{E}$, $match(e, \vartheta)$ holds iff $match(e, \overline{\vartheta})$ fails.

The filter operator can be generalized to obtain the conditional operator $cond$ used in Section 2: $cond(\vartheta, \tau_1, \tau_2)$ can be defined by the trace expression $(T_t \wedge \tau_1) | (T_f \wedge \tau_2)$, where, again, $T_t = \epsilon \vee \vartheta : T_t$, $T_f = \epsilon \vee \overline{\vartheta} : T_f$.

The corresponding rules for the transition system and the auxiliary function $\epsilon(\_)$ can be easily derived, see Figure 3.

4. Properties of trace expression operators

In this section we formally investigate the properties of the operators of parametric trace expressions; the main aim is to provide the foundations for a study of the equational theory of parametric trace expressions.

$$\text{(filter-t)}\ \frac{\tau \overset{e}{\rightarrowtail} \tau'; \sigma_2}{\vartheta \gg \tau \overset{e}{\rightarrow} \vartheta \gg \tau'; \sigma_1 \cup \sigma_2}\ match(e,\vartheta)=\sigma_1$$

$$\text{(filter-f)}\ \frac{}{\vartheta \gg \tau \overset{e}{\rightarrowtail} \vartheta \gg \tau; \emptyset}\ match(e,\vartheta)\ \text{fails}$$

$$\text{(cond-t)}\ \frac{\tau_1 \overset{e}{\rightarrowtail} \tau_1'; \sigma_2}{cond(\vartheta,\tau_1,\tau_2) \overset{e}{\rightarrow} cond(\vartheta,\tau_1',\tau_2); \sigma_1 \cup \sigma_2}\ match(e,\vartheta)=\sigma_1$$

$$\text{(cond-f)}\ \frac{\tau_2 \overset{e}{\rightarrowtail} \tau_2'; \sigma}{cond(\vartheta,\tau_1,\tau_2) \overset{e}{\rightarrow} cond(\vartheta,\tau_1,\tau_2'); \sigma}\ match(e,\vartheta)\ \text{fails}$$

$$\text{(ϵ-filter)}\ \frac{\epsilon(\tau)}{\epsilon(\vartheta \gg \tau)} \qquad \text{(ϵ-cond)}\ \frac{\epsilon(\tau_1)\quad \epsilon(\tau_2)}{\epsilon(cond(\vartheta,\tau_1,\tau_2))}$$

Figure 2. Rules for derived operators (filter and conditional).

$$[\![\epsilon]\!] = \{\epsilon\} \tag{1}$$
$$[\![\vartheta : \tau]\!] = [\![\vartheta]\!] \cdot [\![\tau]\!] \tag{2}$$
$$[\![\tau_1 \cdot \tau_2]\!] = [\![\tau_1]\!]_\omega \cup ([\![\tau_1]\!]_* \cdot [\![\tau_2]\!]) \tag{3}$$
$$[\![\tau_1 \wedge \tau_2]\!] = [\![\tau_1]\!] \cap [\![\tau_2]\!] \tag{4}$$
$$[\![\tau_1 \vee \tau_2]\!] = [\![\tau_1]\!] \cup [\![\tau_2]\!] \tag{5}$$
$$[\![\tau_1 \mid \tau_2]\!] = [\![\tau_1]\!]_\omega \cup [\![\tau_2]\!]_\omega \cup ([\![\tau_1]\!] \mid [\![\tau_2]\!]) \tag{6}$$
$$[\![<\mathsf{x}\,;\,\tau>]\!] = \bigcup_{v \in \mathcal{V}} [\![\{\mathsf{x} \mapsto v\}\tau]\!] \tag{7}$$

Figure 3. Trace expressions semantics.

4.1 Preliminaries

Concatenation between sets of traces is defined on top of trace concatenation as $[\![\tau_1]\!] \cdot [\![\tau_2]\!] = \{\, \bar{e}_1\bar{e}_2 \mid \bar{e}_1 \in [\![\tau_1]\!] \wedge \bar{e}_2 \in [\![\tau_2]\!] \,\}$, assuming $[\![\tau_1]\!]$ does *not* contain infinite traces, as it is customary in the formal languages literature when defining concatenation of (ω-)languages. The semantics of a (ground) event type ϑ is $[\![\vartheta]\!] = \{\, e \mid match(e,\vartheta) = \emptyset \,\}$. $[\![\tau]\!]_*$ and $[\![\tau]\!]_\omega$ denote the restriction of the semantics to finite and infinite traces, respectively, i.e. $[\![\tau]\!]_* = \mathcal{E}^* \cap [\![\tau]\!]$ and $[\![\tau]\!]_\omega = \mathcal{E}^\omega \cap [\![\tau]\!]$. Shuffle between set of traces is defined as follows: $[\![\tau_1]\!] \mid [\![\tau_2]\!] = \bigcup_{\bar{e}_1 \in [\![\tau_1]\!], \bar{e}_2 \in [\![\tau_2]\!]} \bar{e}_1 \mid \bar{e}_2$. To handle possibly infinite traces, the shuffle of two event traces is coinductively defined by the following equations:

$$\bar{e} \mid \epsilon = \epsilon \mid \bar{e} = \{\bar{e}\}$$
$$e_1\bar{e}_1 \mid e_2\bar{e}_2 = e_1(\bar{e}_1 \mid e_2\bar{e}_2) \cup e_2(e_1\bar{e}_1 \mid \bar{e}_2)$$

4.2 Basic properties

The basic properties of trace expression operators are captured by the equations in Figure 3. Proofs are only sketched here; more detailed proofs can be found in the Appendix.

(1) is a direct consequence of Definition 3.1 and $\epsilon(\epsilon)$.

(2) follows from the definition of the membership judgement and rule (prefix), which allows the rewriting step $\vartheta : \tau \overset{e}{\rightarrow} \tau; \emptyset$ whenever $match(e,\vartheta) = \emptyset$.

In (3) the infinitary part of the left trace expression needs to be handled separately. Recall that concatenation of sets of traces is empty if one of the two sets is empty. However, given a trace expression $\tau_1 \cdot \tau_2$ such that $[\![\tau_2]\!] = \emptyset$, it could still be possible to have an infinite reduction on τ_1. The $\subseteq$ inclusion can be proved by considering different cases: if a trace $\bar{e} = ee' \ldots$ is infinite and there is an infinite reduction $\tau_1 \overset{e}{\rightarrow} \tau_1' \overset{e'}{\rightarrow} \cdots$, then $\bar{e} \in [\![\tau_1]\!]_\omega$; otherwise, $\bar{e} = \bar{e}_1\bar{e}_2$, $\bar{e}_1$ is finite and both $\bar{e}_1 \in [\![\tau_1]\!]_*$ and $\bar{e}_2 \in [\![\tau_2]\!]$ hold. The opposite implication can be split into $[\![\tau_1]\!]_\omega \subseteq [\![\tau_1 \cdot \tau_2]\!]$ and $[\![\tau_1]\!]_* \cdot [\![\tau_2]\!] \subseteq [\![\tau_1 \cdot \tau_2]\!]$, which are easier to prove.

Both properties (4) and (5) match the intuitive meaning of union and intersection.

Property (4) can be conveniently proved exploiting the coinduction principle. Consider the inclusion $[\![\tau_1 \wedge \tau_2]\!] \subseteq [\![\tau_1]\!]$, and some $\bar{e} \in [\![\tau_1 \wedge \tau_2]\!]$; we prove the inclusion by coinduction on $\bar{e}$ and τ_1. If $\bar{e} = \epsilon$ the proof is trivial. If $\bar{e} = e\bar{e}'$, by Definition 3.1 $\tau_1 \wedge \tau_2 \overset{e}{\rightarrow} \tau_1' \wedge \tau_2'$, which can only hold if $\tau_1 \overset{e}{\rightarrow} \tau_1'$. Thus, it is possible to derive $\bar{e} \in [\![\tau_1]\!]$ from $\bar{e}' \in [\![\tau_1']\!]$, and since there exists τ such that $\bar{e}' \in [\![\tau_1' \wedge \tau]\!]$ (namely, τ_2') we conclude. Similarly, $[\![\tau_1 \wedge \tau_2]\!] \subseteq [\![\tau_2]\!]$ and $[\![\tau_1]\!] \cap [\![\tau_2]\!] \subseteq [\![\tau_1 \wedge \tau_2]\!]$ can both be proved by coinduction.

It is straightforward to prove (5) by reasoning on the rules (or-l) and (or-r) defining the transition $\tau_1 \vee \tau_2 \rightarrow \tau'$.

In (6), even if $[\![\tau_1]\!] | [\![\tau_2]\!]$ correctly handles possibly infinite traces from both the sets, we still need to take into account the corner case in which one trace expression does not admit any event while the other one accepts some infinite traces. This is similar to what happens with concatenation, since it is possible to have an infinite reduction $\tau_1 \mid \tau_2 \overset{e}{\rightarrow} \tau_1' \mid \tau_2 \overset{e'}{\rightarrow} \cdots$ not involving τ_2 at all (the same thing can happen in the symmetric case). $[\![\tau_1 \mid \tau_2]\!] \subseteq [\![\tau_1]\!]_\omega \cup [\![\tau_2]\!]_\omega \cup ([\![\tau_1]\!] \mid [\![\tau_2]\!])$ can be proved by considering the three different cases we mentioned. On the other hand, the opposite inclusion is proved exploiting again the coinduction principle, since, intuitively, a reduction on a shuffle trace expression never removes the shuffle operator.

Finally, property (7) deals with the $<\mathsf{x}\,;\,\tau>$ construct. Since no assumptions can be made about the value that will be computed at runtime for a variable x, the set of all valid traces must take into account every possible value. In a real-world scenario, only some of them will be valid; for the others, the semantics of the trace expression resulting from the substitution will be empty.

4.3 Derived properties

From the basic properties of Figure 3, other interesting algebraic properties can be derived. For instance, though concatenation of trace expressions is not as simple as formal language concatenation, it is nonetheless associative:

$$
\begin{aligned}
[\![(\tau_1 \cdot \tau_2) \cdot \tau_3]\!] &= [\![\tau_1 \cdot \tau_2]\!]_\omega \cup ([\![\tau_1 \cdot \tau_2]\!]_* \cdot [\![\tau_3]\!]) \\
&= [\![\tau_1]\!]_\omega \cup ([\![\tau_1]\!]_* \cdot [\![\tau_2]\!]_\omega) \cup ([\![\tau_1]\!]_* \cdot [\![\tau_2]\!]_* \cdot [\![\tau_3]\!]) \\
&= [\![\tau_1]\!]_\omega \cup ([\![\tau_1]\!]_* \cdot ([\![\tau_2]\!]_\omega \cup ([\![\tau_2]\!]_* \cdot [\![\tau_3]\!]))) \\
&= [\![\tau_1]\!]_\omega \cup ([\![\tau_1]\!]_* \cdot [\![\tau_2 \cdot \tau_3]\!]) \\
&= [\![\tau_1 \cdot (\tau_2 \cdot \tau_3)]\!]
\end{aligned}
$$

In the steps above we exploited the distributivity of concatenation over union of trace expressions: $[\![\tau_1 \cdot (\tau_2 \vee \tau_3)]\!] = [\![(\tau_1 \cdot \tau_2) \vee (\tau_1 \cdot \tau_3)]\!]$. Clearly, $\{\epsilon\}$ is the identity element.

Since union and intersection of trace expressions directly correspond to their set-theoretic counterparts, many properties come for free: both operators are commutative and associative, and distribute over each other. More precisely, union and intersection form a Boolean algebra with respect to the set of all (possibly infinite) event traces $\mathcal{E}^* \cup \mathcal{E}^\omega$, thus they also have many other properties.

Finally, the shuffle operator is commutative and associative as well, and it distributes over union: $[\![(\tau_1 \vee \tau_2) \mid \tau_3]\!] = [\![(\tau_1 \mid \tau_3) \vee (\tau_2 \mid \tau_3)]\!]$.

5. Conclusion

We have shown how parametric trace expressions can be usefully adopted for specifying the behavior of a dynamically evolving community of objects to perform RV of a Java-like program.

We have formally studied the basic properties of the primitive operators of parametric trace expressions to provide the foundations of an equational theory of parametric trace expressions useful for reasoning on equivalence of behavioral specifications, and for validating optimization transformations to speed up runtime monitoring.

In literature there exist several proposals for RV of object-oriented languages (Allan et al. 2005; Martin et al. 2005; Brörkens and Möller 2002; Colombo et al. 2009; Chen and Rosu 2007; Luo et al. 2014; de Boer and de Gouw 2014). Even though many of them are based on specification formalisms (regular expressions, context-free grammars, and LTL) which are less expressive than trace expressions in the context of RV, a deeper comparison with related work is matter for future investigation.

We are also planning to experiment our prototype tool developed for RV of multiagent systems (Ancona et al. 2017) in the context of object-oriented programming, to verify how it can scale to more complex systems and specifications; besides Java-like languages, object-oriented RV can be usefully adopted for dynamic languages as JavaScript, and widespread platforms based on them, as Node.js.

Finally, we are considering to exploit the equational theory of parametric trace expressions developed here to optimize our prototype RV tool by simplifying trace expressions that are dynamically generated by the monitoring system driven by the reduction semantics of trace expressions.

References

C. Allan, P. Avgustinov, A. S. Christensen, L. Hendren, S. Kuzins, O. Lhoták, O. de Moor, D. Sereni, G. Sittampalam, and J. Tibble. Adding trace matching with free variables to AspectJ. In *OOPSLA '05*, pages 345–364. ACM, 2005.

D. Ancona, S. Drossopoulou, and V. Mascardi. Automatic generation of self-monitoring MASs from multiparty global session types in Jason. In *DALT 2012*, volume 7784 of *LNAI*, pages 76–95. Springer, 2012.

D. Ancona, D. Briola, A. El Fallah Seghrouchni, V. Mascardi, and P. Taillibert. Efficient Verification of MASs with Projections. In *EMAS 2014, Revised Selected Papers*, pages 246–270, 2014.

D. Ancona, V. Bono, M. Bravetti, J. Campos, G. Castagna, P. M. Deniélou, S. J. Gay, N. Gesbert, E. Giachino, R. Hu, E. B. Johnsen, F. Martins, V. Mascardi, F. Montesi, R. Neykova, N. Ng, L. Padovani, V. Vasconcelos, and N. Yoshida. Behavioral types in programming languages. *Foundations and Trends in Programming Languages*, 3(2-3):95–230, 2016a.

D. Ancona, A. Ferrando, and V. Mascardi. Comparing trace expressions and linear temporal logic for runtime verification. In *Theory and Practice of Formal Methods - Essays Dedicated to Frank de Boer on the Occasion of His 60th Birthday*, pages 47–64. Springer Verlag, 2016b.

D. Ancona, A. Ferrando, and V. Mascardi. Parametric runtime verification of multiagent systems. In *AAMAS 2017. Proceedings*, 2017. Extended Abstract. To appear.

D. Briola, V. Mascardi, and D. Ancona. Distributed runtime verification of JADE multiagent systems. In *IDC*, Studies in Computational Intelligence. Springer, 2014.

M. Brörkens and M. Möller. Dynamic event generation for runtime checking using the JDI. *Electr. Notes Theor. Comput. Sci.*, 70 (4):21–35, 2002.

F. Chen and G. Rosu. Mop: an efficient and generic runtime verification framework. In *OOPSLA 2007*, pages 569–588, 2007.

C. Colombo, G. J. Pace, and G. Schneider. LARVA — Safer Monitoring of Real-Time Java Programs (Tool Paper). In *SEFM 2009*, pages 33–37, 2009.

B. Courcelle. Fundamental properties of infinite trees. *Theoretical Comput. Sci.*, 25:95–169, 1983.

F. S. de Boer and C. P. T. de Gouw. Combining Monitoring With Run-Time Assertion Checking. In *SFM 14*, pages 217 – 262. Springer, 2014.

Q. Luo, Y. Zhang, C. Lee, D. Jin, P. O. Meredith, T. F. Serbanuta, and G. Rosu. Rv-monitor: Efficient parametric runtime verification with simultaneous properties. In *RV'14*, volume 8734, pages 285–300. Springer, 2014.

M. C. Martin, V. B. Livshits, and M. S. Lam. Finding application errors and security flaws using PQL: a program query language. In *OOPSLA 2005*, pages 365–383, 2005.

C. Vasconcelos and A. Ravara. From object-oriented code with assertions to behavioural types. In *SAC 2017*, 2017.

A. Proofs

This section is devoted to the proofs of the equations describing trace expressions semantics in Figure 3. Most of the times we will prove the two inclusions separately.

It is convenient to explicitly rewrite Definition 3.1 as an inference system:

$$(\in\text{-empty})\ \frac{}{\bar{e} \in [\![\tau]\!]}\ \epsilon(\tau) \qquad (\in\text{-non-empty})\ \frac{\bar{e} \in [\![\tau']\!]}{e\bar{e} \in [\![\tau]\!]}\ \tau \xrightarrow{e} \tau'$$

Some of the proofs go by coinduction. We assume familiarity with the coinduction proof principle, especially in the case of inference systems.

Proposition A.1 (Empty trace)**.** *For all event traces $\bar{e}$,*

$$\bar{e} \in [\![\epsilon]\!] \iff \bar{e} = \epsilon$$

Proof. It follows from rules ($\in$-empty) and (ϵ-empty). □

Proposition A.2 (Prefix)**.** *For all event traces $\bar{e}$, event type ϑ and trace expression τ,*

$$\bar{e} \in [\![\vartheta : \tau]\!] \iff \exists e, \bar{e}'.\, \bar{e} = e\bar{e}' \text{ and } e \in [\![\vartheta]\!] \text{ and } \bar{e}' \in [\![\tau]\!]$$

Proof. It follows from rules ($\in$-non-empty) and (prefix). □

Proposition A.3 (Concatenation $\subseteq$)**.** *For all event traces $\bar{e}$ and trace expressions $\tau_1 \cdot \tau_2$,*

$$\begin{aligned}\bar{e} \in [\![\tau_1 \cdot \tau_2]\!] \implies\ & \text{either } (\bar{e} \text{ infinite and } \bar{e} \in [\![\tau_1]\!]) \\ & \text{or } (\exists \bar{e}_1 \in [\![\tau_1]\!], \bar{e}_2 \in [\![\tau_2]\!].\\ & \quad \bar{e}_1 \text{ finite and } \bar{e} = \bar{e}_1\bar{e}_2)\end{aligned}$$

Proof. We distringuish different cases. If there is a (possibly infinite) reduction $\tau_1 \cdot \tau_2 \xrightarrow{e_1} \tau_1' \cdot \tau_2 \xrightarrow{e_1'} \cdots$, then by definition of rule (cat-l) we can also prove $\tau_1 \xrightarrow{e_1} \tau_1' \xrightarrow{e_1'} \cdots$. If such a reduction is infinite, then $\bar{e}$ must be infinite as well and $\bar{e} \in \tau_1$. Otherwise the reduction ends after n steps and $\tau_1^{(n)} \cdot \tau_2 \xrightarrow{e_2} \tau_2' \xrightarrow{e_2'} \cdots$ by rule (cat-r). Thus $\bar{e} = \bar{e}_1\bar{e}_2$, $\bar{e}_1 \in [\![\tau_1]\!]$ and $\bar{e}_2 \in [\![\tau_2]\!]$. □

Proposition A.4 (Concatenation $\supseteq$-1)**.** *For all event traces $\bar{e}$ and trace expressions τ_1, τ_2,*

$$\bar{e} \text{ infinite and } \bar{e} \in [\![\tau_1]\!] \implies \bar{e} \in [\![\tau_1 \cdot \tau_2]\!]$$

Proof. By coinduction. $\bar{e} = e\bar{e}'$. From $\bar{e} \in [\![\tau_1]\!]$, $\tau_1 \xrightarrow{e} \tau_1'$. Given $\bar{e} \in [\![\tau_1 \cdot \tau_2]\!]$, it can be shown there exists a rule to derive the judgement by means of rule (cat-l) and $\tau_1 \xrightarrow{e} \tau_1'$. Since $\bar{e}$ is infinite, $\bar{e}'$ is infinite as well and $\bar{e}' \in [\![\tau_1' \cdot \tau_2]\!]$. □

Proposition A.5 (Concatenation $\supseteq$-2)**.** *For all event traces $\bar{e}$ and trace expressions τ_1, τ_2,*

$$\begin{aligned}\exists \bar{e}_1, \bar{e}_2.\, \bar{e}_1 \text{ finite and } \bar{e} = \bar{e}_1\bar{e}_2 \text{ and } \bar{e}_2 \in [\![\tau_1]\!] \text{ and } \bar{e}_2 \in [\![\tau_2]\!]\\ \implies \bar{e} \in [\![\tau_1 \cdot \tau_2]\!]\end{aligned}$$

Proof. From $\bar{e}_1 \in [\![\tau_1]\!]$ we know there is a finite reduction $\tau_1 \to \cdots \to \tau_1'$ such that $\epsilon(\tau_1')$. Similarly, since $\bar{e}_2 \in [\![\tau_2]\!]$, there is a (possibly infinite) reduction starting from τ_2. Then by rule (cat-l) (for the first part) and (cat-r) (for the second part) $\bar{e} \in [\![\tau_1 \cdot \tau_2]\!]$. □

Proposition A.6 (Intersection $\subseteq$)**.** *For all event traces $\bar{e}$ and trace expressions τ_1, τ_2,*

$$\bar{e} \in [\![\tau_1 \wedge \tau_2]\!] \implies \bar{e} \in [\![\tau_1]\!] \text{ and } \bar{e} \in [\![\tau_2]\!]$$

Proof. By coinduction we prove the implication for τ_1, but the same reasoning holds for τ_2 as well. The case $\bar{e} = \epsilon$ is a trivial consequence of rules ($\in$-empty) and (ϵ-and). Let us assume $\bar{e} = e\bar{e}'$. Since $\bar{e} \in [\![\tau_1 \wedge \tau_2]\!]$, by definition of rule ($\in$-non-empty) $\bar{e}' \in [\![\tau_1' \wedge \tau_2']\!]$ and $\tau_1 \wedge \tau_2 \xrightarrow{e} \tau_1' \wedge \tau_2'$, and by definition of rule (and) $\tau_1 \xrightarrow{e} \tau_1'$. Then there exists a rule to derive $\bar{e} \in [\![\tau_1]\!]$ from the premise $\bar{e}' \in [\![\tau_1']\!]$. We conclude showing there is a τ such that $\bar{e}' \in [\![\tau_1' \wedge \tau]\!]$, namely τ_2'. □

Proposition A.7 (Intersection $\supseteq$)**.** *For all event traces $\bar{e}$ and trace expressions τ_1, τ_2,*

$$\bar{e} \in [\![\tau_1]\!] \text{ and } \bar{e} \in [\![\tau_2]\!] \implies \bar{e} \in [\![\tau_1 \wedge \tau_2]\!]$$

Proof. By coinduction. Again, since the case $\bar{e} = \epsilon$ is trivial, we assume $\bar{e} = e\bar{e}'$. From the assumptions, $\tau_1 \xrightarrow{e} \tau_1'$ and $\tau_2 \xrightarrow{e} \tau_2'$. From these, applying rule (and) we get $\tau_1 \wedge \tau_2 \xrightarrow{e} \tau_1' \wedge \tau_2'$. Then there exists a rule to derive $\bar{e} \in [\![\tau_1 \wedge \tau_2]\!]$, with the premise $\bar{e}' \in [\![\tau_1' \wedge \tau_2']\!]$. We conclude since both $\bar{e}' \in [\![\tau_1']\!]$ and $\bar{e}' \in [\![\tau_2']\!]$ hold. □

Proposition A.8 (Union $\subseteq$)**.** *For any trace $\bar{e}$ and trace expression τ_1, τ_2,*

$$\bar{e} \in [\![\tau_1 \vee \tau_2]\!] \implies \text{either } \bar{e} \in [\![\tau_1]\!] \text{ or } \bar{e} \in [\![\tau_2]\!]$$

Proof. If $\bar{e} = \epsilon$, it follows from rules ($\in$-empty), (ϵ-or-l) and (ϵ-or-r). If $\bar{e} = e\bar{e}'$, $\tau_1 \vee \tau_2 \xrightarrow{e} \tau_i'$ for some $i \in \{1, 2\}$, and $\bar{e}' \in [\![\tau_i']\!]$. By the definition of rules (or-l) and (or-r), $\tau_i \xrightarrow{e} \tau_1'$, thus $\bar{e} \in [\![\tau_i]\!]$. □

Proposition A.9 (Union $\supseteq$)**.** *For all event traces $\bar{e}$ and trace expressions τ_1, τ_2,*

$$\text{either } \bar{e} \in [\![\tau_1]\!] \text{ or } \bar{e} \in [\![\tau_2]\!] \implies \bar{e} \in [\![\tau_1 \vee \tau_2]\!]$$

Proof. Without loss of generality, assume that $\bar{e} \in [\![\tau_1]\!]$ holds (the same reasoning works for $\bar{e} \in [\![\tau_2]\!]$). If $\bar{e} = \epsilon$, the statement is a trivial consequence of rules ($\in$-empty) and (ϵ-or-l). If $\bar{e} = e\bar{e}'$, $\tau_1 \xrightarrow{e} \tau_1'$ and $\bar{e}' \in [\![\tau_1']\!]$. By rule (or-l) $\tau_1 \vee \tau_2 \xrightarrow{e} \tau_1'$, thus $\bar{e} \in [\![\tau_1 \vee \tau_2]\!]$. □

Proposition A.10 (Shuffle $\subseteq$)**.** *For all event traces $\bar{e}$ and trace expressions τ_1, τ_2,*

$$\begin{aligned}\bar{e} \in [\![\tau_1 \mid \tau_2]\!] \implies\ & \text{either } (\exists \bar{e}_1 \in [\![\tau_1]\!], \bar{e}_2 \in [\![\tau_2]\!].\, \bar{e} \in \bar{e}_1 \mid \bar{e}_2)\\ & \text{or } (\bar{e} \text{ infinite and } \bar{e} \in [\![\tau_1]\!])\\ & \text{or } (\bar{e} \text{ infinite and } \bar{e} \in [\![\tau_2]\!])\end{aligned}$$

Proof. We consider three different cases, corresponding to the three possible scenarios above. If $\tau_1 \mid \tau_2$ comes from an infinite reduction on τ_1 ($\tau_1 \mid \tau_2 \xrightarrow{e} \tau_1' \mid \tau_2 \xrightarrow{e'} \cdots$) then it is easy to see that $\bar{e}$ is infinite and $\bar{e} \in [\![\tau_1]\!]$; the same reasoning holds for an infinite reduction on τ_2. If both rules (shuffle-l) and (shuffle-r) are used in the derivation, we consider all events e_1 such that $\tau_1^{(i)} \xrightarrow{e_1} \tau_1^{(i+1)}$, as well as events e_2 such that $\tau_2^{(j)} \xrightarrow{e_2} \tau_2^{(j+1)}$. By definition of the shuffle operator $\bar{e} \in \bar{e}_1 \mid \bar{e}_2$, and both $\bar{e}_1 \in [\![\tau_1]\!]$ and $\bar{e}_2 \in [\![\tau_2]\!]$ hold. □

Proposition A.11 (Shuffle $\supseteq$)**.** *For all event traces $\bar{e}$ and trace expressions τ_1, τ_2,*

$$\exists \bar{e}_1 \in [\![\tau_1]\!], \bar{e}_2 \in [\![\tau_2]\!]. \bar{e} \in \bar{e}_1 \mid \bar{e}_2 \implies \bar{e} \in [\![\tau_1 \mid \tau_2]\!]$$

Proof. By coinduction. If $\bar{e} = \epsilon$ then also $\bar{e}_1 = \bar{e}_2 = \epsilon$ and the proof is a consequence of rule (ϵ-shuffle). If $\bar{e} = e\bar{e}'$, by definition of the shuffle operator either $\bar{e}_1 = e\bar{e}_1'$ or $\bar{e}_2 = e\bar{e}_2'$. Without loss of generality let us assume the former. From the assumptions, $\tau_1 \xrightarrow{e} \tau_1'$, and by rule (shuffle-l) $\tau_1 \mid \tau_2 \xrightarrow{e} \tau_1' \mid \tau_2$. Then there exists a rule to derive $\bar{e} \in [\![\tau_1 \mid \tau_2]\!]$ from a premise $\bar{e} \in [\![\tau_1' \mid \tau_2]\!]$. We can conclude since $\bar{e}' \in \bar{e}_1' \mid \bar{e}_2$ (by definition of shuffle), $\bar{e}_1' \in [\![\tau_1']\!]$ and $\bar{e}_2 \in [\![\tau_2]\!]$. □

Proposition A.12 (Binder $\subseteq$)**.** *For all event traces $\bar{e}$, variables $\mathsf{x} \in \mathcal{X}$ and trace expressions τ,*

$$\bar{e} \in [\![<\mathsf{x} ; \tau>]\!] \implies \exists v \in \mathcal{V}. \bar{e} \in [\![\{\mathsf{x} \mapsto v\}\tau]\!]$$

Proof. If $\bar{e} = \epsilon$, then $\epsilon(<\mathsf{x} ; \tau>)$ and the substitution has no effect since it can only change event types, which are not taken into account by $\epsilon(\_)$; in this case we can conclude choosing any $v \in \mathcal{V}$. Now let us assume $\bar{e} = e\bar{e}'$, and consider the different rules from which the judgement $\bar{e} \in [\![<\mathsf{x} ; \tau>]\!]$ can be derived.

If $<\mathsf{x} ; \tau> \xrightarrow{e} \sigma\tau'$ by means of rule (var-t), then $\bar{e}' \in [\![\sigma\tau']\!]$, $\tau \overset{e}{\rightarrowtail} \tau'; \sigma$ and the computed substitution σ has the shape $\{\mathsf{x} \mapsto v\}$. By induction on the inference system defining $\rightarrowtail$ it can be proved that $\tau \overset{e}{\rightarrowtail} \tau'; \sigma \implies \sigma\tau \overset{e}{\rightarrowtail} \sigma\tau'; \emptyset$. From this we derive $\bar{e} \in [\![\sigma\tau]\!]$.

If there is an infinite reduction $<\mathsf{x};\tau> \xrightarrow{e} <\mathsf{x};\tau'> \xrightarrow{e'} \cdots$ using rule (var-f) alone, then $\tau \xrightarrow{e} \tau' \xrightarrow{e'} \cdots$. It can be proved by induction that $\sigma\tau \xrightarrow{e} \sigma\tau' \xrightarrow{e'} \cdots$ hold as well for any $\sigma = \{\mathsf{x} \mapsto v\}$, thus $\bar{e} \in [\![\sigma\tau]\!]$.

Every other case can be handled using the reasoning above. □

Proposition A.13 (Binder $\supseteq$)**.** *For all event traces $\bar{e}$, variables x and trace expressions τ,*

$$\exists v \in \mathcal{V}. \bar{e} \in [\![\{\mathsf{x} \mapsto v\}\tau]\!] \implies \bar{e} \in [\![<\mathsf{x} ; \tau>]\!]$$

Proof. In this case the substitution $\{\mathsf{x} \mapsto v\}$ can be used together with rule (var-t) to prove the conclusion, since it is exactly the substitution needed to proceed with the reduction sequence. □