Can determinism and compositionality coexist in RML?

Davide Ancona   Viviana Mascardi   Angelo Ferrando

DIBRIS, University of Genova, Italy   University of Manchester, UK

{Davide.Ancona,Viviana.Mascardi}@unige.it   angelo.ferrando@manchester.ac.uk

Runtime verification (RV) consists in dynamically verifying that the event traces generated by single runs of a system under scrutiny (SUS) are compliant with the formal specification of its expected properties. RML (Runtime Monitoring Language) is a simple but expressive Domain Specific Language for RV; its semantics is based on a trace calculus formalized by a deterministic rewriting system which drives the implementation of the interpreter of the monitors generated by the RML compiler from the specifications. While determinism of the trace calculus ensures better performances of the generated monitors, it makes the semantics of its operators less intuitive. In this paper we move a first step towards a compositional semantics of the RML trace calculus, by interpreting its basic operators as operations on sets of instantiated event traces and by proving that such an interpretation is equivalent to the operational semantics of the calculus.

1 Introduction

RV [35, 27, 13] consists in dynamically verifying that the event traces generated by single runs of a SUS are compliant with the formal specification of its expected properties.

The RV process needs as inputs the SUS and the specification of the properties to be verified, usually defined with either a domain specific (DSL) or a programming language, to denote the set of valid event traces; RV is performed by monitors, automatically generated from the specification, which consume the observed events of the SUS, emit verdicts and, in case they work online while the SUS is executing, feedback useful for error recovery.

RV is complimentary to other verification methods: analogously to formal verification, it uses a specification formalism, but, as opposite to it, scales well to real systems and complex properties and it is not exhaustive as happens in software testing; however, it also exhibits several distinguishing features: it is quite useful to check control-oriented properties [2], and offers opportunities for fault protection when the monitor runs online. Many RV approaches adopt a DSL language to specifiy properties to favor portability and reuse of specifications and interoperability of the generated monitors and to provide stronger correctness guarantees: monitors automatically generated from a higher level DSL are more reliable than ad hoc code implemented in a ordinary programming language to perform RV.

RML[1] [28] is a simple but expressive DSL for RV which can be used in practice for RV of complex non Context-Free properties, as FIFO properties, which can be verified by the generated monitors in time linear in the size of the inspected trace; the language design and implementation is based on previous work on trace expressions and global types [7, 17, 4, 9], which have been adopted for RV in several contexts. Its semantics is based on a trace calculus formalized by a rewriting system which drives the implementation of the interpreter of the monitors generated by the RML compiler from the specifications; to allow better performances, the rewriting system is fully deterministic [11] by adopting a left-preferential evaluation strategy for binary operators and, thus, no monitor backtracking is needed and exponential

---

[1]https://rmlatdibris.github.io

explosion of the space allocated for the states of the monitor is avoided. A similar strategy is followed by mainstream programming languages in predefined libraries for regular expressions for efficient incremental matching of input sequences, to avoid the issue of Regular expression Denial of Service (ReDoS) [24]: for instance, given the regular expression `a?(ab)?` (optionally `a` concatenated with optionally `ab`) and the input sequence `ab`, the Java method `lookingAt()` of class `java.util.regex.Matcher` matches `a` instead of the entire input sequence `ab` because the evaluation of concatenation is deterministically left-preferential.

As explained more in details in Section 5, with respect to other existing RV formalisms, RML has been designed as an extension of regular expressions and deterministic context-free grammars, which are widely used in RV because they are well-understood among software developers as opposite to other more sophisticated approaches, as temporal logics. As shown in previous papers [7, 17, 4, 9], the calculus at the basis of RML allows users to define and efficiently check complex parameterized properties and it has been proved to be more expressive than LTL [8].

Unfortunately, while determinism ensures better performances, it makes the compositional semantics of its operators less intuitive; for instance, the example above concerning the regular expression `a?(ab)?` with deterministic left-preferential concatenation applies also to RML, which is more expressive than regular expressions: the compositional semantics of concatenation does not correspond to standard language concatenation, because `a?` and `(ab)?` denote the formal languages $\{\lambda, a\}$ and $\{\lambda, ab\}$, respectively, where λ denotes the empty string, while, if concatenation is deterministically left-preferential, then the semantics of `a?(ab)?` is $\{\lambda, a, aab\}$ which does not coincide with the language $\{\lambda, a, ab, aab\}$ obtained by concatenating $\{\lambda, a\}$ with $\{\lambda, ab\}$. In Section 4 we show that the semantics of left-preferential concatenation can still be given compositionally, although the corresponding operator is more complicate than standard language concatenation. Similar results follow for the other binary operators of RML (union, intersection and shuffle); in particular, the compositional semantics of left-preferential shuffle is more challenging. Furthermore, the fact that RML supports parametricity makes the compositional semantics more complex, since traces must be coupled with the corresponding substitutions generated by event matching. To this aim, as a first step towards a compositional semantics of the RML trace calculus, we provide an interpretation of the basic operators of the RML trace calculus as operations on sets of instantiated event traces, that is, pairs of trace of events and substitutions computed to bind the variables occurring in the event type patterns used in the specifications and to associate them with the data values carried by the matched events. Furthermore we prove that such an interpretation is equivalent to the original operational semantics of the calculus based on the deterministic rewriting system.

The paper is structured as follows: Section 2 introduces the basic definitions which are used in the subsequent technical sections, Section 3 formalizes the RML trace calculus and its operational semantics, while Section 4 introduces the semantics based on sets of instantiated event traces and formally proves its equivalence with the operational semantics; finally, Section 5 is devoted to the related work and Section 6 draws conclusions and directions for further work. For space limitations, some proof details can be found in the extended version [10] of this paper.

2   Technical background

This section introduces some basic definitions and propositions used in the next technical sections.

Partial functions: Let $f : D \to C$ be a partial function; then $dom(f) \subseteq D$ denotes the set of elements $d \in D$ s.t. $f(d)$ is defined (hence, $f(d) \in C$).

A partial function over natural numbers $f : \mathbb{N} \to N$, with $N \subseteq \mathbb{N}$, is *strictly increasing* iff for all $n_1, n_2 \in dom(f)$, $n_1 < n_2$ implies $f(n_1) < f(n_2)$. From this definition one can easily deduce that a strictly increasing partial function over natural numbers is always injective, and, hence, it is bijective iff it is surjective.

Proposition 2.1 *Let $f : \mathbb{N} \to N$, with $N \subseteq \mathbb{N}$, be a strictly increasing partial function. Then for all $n_1, n_2 \in dom(f)$, if $f(n_1) < f(n_2)$, then $n_1 < n_2$.*

Proposition 2.2 *Let $f : \mathbb{N} \to N$, with $N \subseteq \mathbb{N}$, be a strictly increasing partial function satisfying the following conditions:*

1. *f is surjective (hence, bijective);*

2. *for all $n \in \mathbb{N}$, if $n + 1 \in dom(f)$, then $n \in dom(f)$;*

3. *for all $n \in \mathbb{N}$, if $n + 1 \in N$, then $n \in N$;*

Then, for all $n \in \mathbb{N}$, if $n \in dom(f)$, then $f(n) = n$, hence f is the identity over $dom(f)$, and $dom(f) = N$.

Event traces: Let $\mathscr{E}$ denotes a possibly infinite set $\mathscr{E}$ of events, called the *event universe*. An event trace over the event universe $\mathscr{E}$ is a partial function $\bar{e} : \mathbb{N} \to \mathscr{E}$ s.t. for all $n \in \mathbb{N}$, if $n + 1 \in dom(\bar{e})$, then $n \in dom(\bar{e})$. We call $\bar{e}$ *finite/infinite* iff $dom(\bar{e})$ is finite/infinite, respectively; when $\bar{e}$ is finite, its length $|\bar{e}|$ coincides with the cardinality of $dom(\bar{e})$, while $|\bar{e}|$ is undefined for infinite traces $\bar{e}$. From the definitions above one can easily deduce that if $\bar{e}$ is finite, then $dom(\bar{e}) = \{n \in \mathbb{N} \mid n < |\bar{e}|\}$. We denote with λ the unique trace over $\mathscr{E}$ s.t. $|\lambda| = 0$; when not ambiguous, we denote with e the trace $\bar{e}$ s.t. $|\bar{e}| = 1$ and $\bar{e}(0) = e$.

For simplicity, in the rest of the paper we implicitly assume that all considered event traces are defined over the same event universe.

Concatenation: The concatenation $\bar{e}_1 \cdot \bar{e}_2$ of event trace $\bar{e}_1$ and $\bar{e}_2$ is the trace $\bar{e}$ s.t.

- if $\bar{e}_1$ is infinite, then $\bar{e} = \bar{e}_1$;

- if $\bar{e}_1$ is finite, then $\bar{e}(n) = \bar{e}_1(n)$ for all $n \in dom(\bar{e}_1)$, $\bar{e}(n + |\bar{e}_1|) = \bar{e}_2(n)$ for all $n \in dom(\bar{e}_2)$, and if $\bar{e}_2$ is finite, then $dom(\bar{e}) = \{n \mid n < |\bar{e}_1| + |\bar{e}_2|\}$.

From the definition above one can easily deduce that λ is the identity of $\cdot$, and that $\bar{e}_1 \cdot \bar{e}_2$ is infinite iff $\bar{e}_1$ or $\bar{e}_2$ is infinite. The trace $\bar{e}_1$ is a prefix of $\bar{e}_2$, denoted with $\bar{e}_1 \lhd \bar{e}_2$, iff there exists $\bar{e}$ s.t. $\bar{e}_1 \cdot \bar{e} = \bar{e}_2$. If T_1 and T_2 are two sets of event traces over $\mathscr{E}$, then $T_1 \cdot T_2$ is the set $\{\bar{e}_1 \cdot \bar{e}_2 \mid \bar{e}_1 \in T_1, \bar{e}_2 \in T_2\}$. We write $\bar{e}_1 \lhd T$ to mean that there exists $\bar{e}_2 \in T$ s.t. $\bar{e}_1 \lhd \bar{e}_2$.

Shuffle: The shuffle $\bar{e}_1 \mid \bar{e}_2$ of event trace $\bar{e}_1$ and $\bar{e}_2$ is the set of traces T s.t. $\bar{e} \in T$ iff $dom(\bar{e})$ can be partitioned into N_1 and N_2 in such a way that there exist two strictly increasing and bijective[2] partial functions $f_1 : dom(\bar{e}_1) \to N_1$ and $f_2 : dom(\bar{e}_2) \to N_2$ s.t.

$\bar{e}_1(n_1) = \bar{e}(f_1(n_1))$ and $\bar{e}_2(n_2) = \bar{e}(f_2(n_2))$, for all $n_1 \in dom(\bar{e}_1)$, $n_2 \in dom(\bar{e}_2)$.

From the definition above, the definition of λ and Proposition 2.2 one can deduce that $\lambda \mid \bar{e} = \bar{e} \mid \lambda = \{\bar{e}\}$; it is easy to show that for all $\bar{e} \in \bar{e}_1 \mid \bar{e}_2$, $\bar{e}$ is infinite iff $\bar{e}_1$ or $\bar{e}_2$ is infinite, and $|\bar{e}| = n$ iff $|\bar{e}_1| = n_1$, $|\bar{e}_2| = n_2$ and $n = n_1 + n_2$.

If T_1 and T_2 are two sets of event traces over $\mathscr{E}$, then $T_1 \mid T_2$ is the set $\bigcup_{\bar{e}_1 \in T_1, \bar{e}_2 \in T_2} (\bar{e}_1 \mid \bar{e}_2)$.

---

[2]Actually, the sufficient condition is surjectivity, but bijectivity can be derived from the fact that the functions are strictly increasing over natural numbers.

Left-preferential shuffle: The left-preferential shuffle $\bar{e}_1 \leftarrow | \bar{e}_2$ of event trace $\bar{e}_1$ and $\bar{e}_2$ is the set of traces $T \subseteq \bar{e}_1 \,|\, \bar{e}_2$ s.t. $\bar{e} \in T$ iff $dom(\bar{e})$ can be partitioned into N_1 and N_2 in such a way that there exist two strictly increasing and bijective partial functions $f_1 : dom(\bar{e}_1) \rightarrow N_1$ and $f_2 : dom(\bar{e}_2) \rightarrow N_2$ s.t.

- $\bar{e}_1(n_1) = \bar{e}(f_1(n_1))$ and $\bar{e}_2(n_2) = \bar{e}(f_2(n_2))$, for all $n_1 \in dom(\bar{e}_1)$, $n_2 \in dom(\bar{e}_2)$;

- for all $n_2 \in dom(\bar{e}_2)$, if $m = \min\{n_1 \in dom(\bar{e}_1) \mid f_2(n_2) < f_1(n_1)\}$, then $\bar{e}_1(m) \neq \bar{e}_2(n_2)$.

In the definition above, if[3] $\{n_1 \in dom(\bar{e}_1) \mid f_2(n_2) < f_1(n_1)\} = \emptyset$, then the second condition trivially holds.

As an example, if we have two traces of events $\bar{e}_1 = e_1 \cdot e_2$, and $\bar{e}_2 = e_2 \cdot e_3$, by applying the left-preferential shuffle we obtain the set of traces $\bar{e}_1 \leftarrow | \bar{e}_2 = \{e_1 \cdot e_2 \cdot e_2 \cdot e_3, e_2 \cdot e_3 \cdot e_1 \cdot e_2, e_2 \cdot e_1 \cdot e_3 \cdot e_2, e_2 \cdot e_1 \cdot e_2 \cdot e_3\}$. With respect to $\bar{e}_1 \,|\, \bar{e}_2$, the trace $e_1 \cdot e_2 \cdot e_3 \cdot e_2$ has been excluded, since this can be obtained only when the first occurrence of e_2 belongs to $\bar{e}_2$; formally, this correponds to the functions $f_1 : \{0,1\} \rightarrow \{0,3\}$ and $f_2 : \{0,1\} \rightarrow \{1,2\}$ s.t. $f_1(0) = 0, f_1(1) = 3, f_2(0) = 1, f_2(1) = 2$, which satisfy the first item of the definition, but not the second, because $\min\{n_1 \in \{0,1\} \mid f_2(0) = 1 < f_1(n_1)\} = 1$ and $\bar{e}_1(1) = e_2 = \bar{e}_2(0)$; the functions f_1' and f_2' s.t. $f_1'(0) = 0, f_1'(1) = 1, f_2'(0) = 3, f_2'(1) = 2$ satisfy both items, but f_2' is **not** strictly increasing.

Generalized left-preferential shuffle: Given a set of event traces T, the generalized left-preferential shuffle $\bar{e}_1 \leftarrow |_T \bar{e}_2$ of event trace $\bar{e}_1$ and $\bar{e}_2$ w.r.t. T is the set of traces $T' \subseteq \bar{e}_1 \leftarrow | \bar{e}_2$ s.t. $\bar{e} \in T'$ iff $dom(\bar{e})$ can be partitioned into N_1 and N_2 in such a way that there exist two strictly increasing and bijective partial functions $f_1 : dom(\bar{e}_1) \rightarrow N_1$ and $f_2 : dom(\bar{e}_2) \rightarrow N_2$ s.t.

- $\bar{e}_1(n_1) = \bar{e}(f_1(n_1))$ and $\bar{e}_2(n_2) = \bar{e}(f_2(n_2))$, for all $n_1 \in dom(\bar{e}_1)$, $n_2 \in dom(\bar{e}_2)$;

- for all $n_2 \in dom(\bar{e}_2)$, if $m = \min\{n_1 \in dom(\bar{e}_1) \mid f_2(n_2) < f_1(n_1)\}$, then $\bar{e}'(m) \neq \bar{e}_2(n_2)$ for all $\bar{e}' \in T$ s.t. $m \in dom(\bar{e}')$.

From the definitions of the shuffle operators above one can easily deduce that $\bar{e}_1 \leftarrow |_\emptyset \bar{e}_2 = \bar{e}_1 \,|\, \bar{e}_2$ and $\bar{e}_1 \leftarrow |_{\{\bar{e}_1\}} \bar{e}_2 = \bar{e}_1 \leftarrow | \bar{e}_2$, for all event traces $\bar{e}_1$, $\bar{e}_2$. This generalisation of the left-preferential shuffle is needed to define the compositional semantics of the shuffle in Section 4. Let us consider $T_1 = \{e_1 \cdot e_2, e_3 \cdot e_4\}$ and $T_2 = \{e_1 \cdot e_5\}$; one might be tempted to define $T_1 \leftarrow | T_2$ as the set $\{\bar{e} \mid \bar{e}_1 \in T_1, \bar{e}_2 \in T_2, \bar{e} \in \bar{e}_1 \leftarrow | \bar{e}_2\}$, which corresponds to $\{e_1 \cdot e_2 \cdot e_1 \cdot e_5, e_1 \cdot e_1 \cdot e_2 \cdot e_5, e_1 \cdot e_1 \cdot e_5 \cdot e_2, e_3 \cdot e_4 \cdot e_1 \cdot e_5, e_3 \cdot e_1 \cdot e_4 \cdot e_5, e_3 \cdot e_1 \cdot e_5 \cdot e_4, e_1 \cdot e_5 \cdot e_3 \cdot e_4, e_1 \cdot e_3 \cdot e_4 \cdot e_5, e_1 \cdot e_3 \cdot e_5 \cdot e_4\}$. But, the last three traces, where e_1 is consumed from T_2 as first event, are not correct, because the event e_1 in T_1 must take the precedence. Thus, the correct definition is given by $\{\bar{e} \mid \bar{e}_1 \in T_1, \bar{e}_2 \in T_2, \bar{e} \in \bar{e}_1 \leftarrow |_{T_1} \bar{e}_2\}$, which does not contain the three traces mentioned above.

3   The RML trace calculus

In this section we define the operational semantics of the trace calculus on which RML is based on. An RML specification is compiled into a term of the trace calculus, which is used as an Intermediate Representation, and then a SWI-Prolog[4] monitor is generated; its execution employs the interpreter of the trace calculus, whose SWI-Prolog implementation is directly driven by the reduction rules defining the labeled transition system of the calculus.

Syntax.   The syntax of the calculus is defined in Figure 1. The main basic building block of the calculus

---

[3]This happens iff in $\bar{e}$ all events of $\bar{e}_1$ precede position n_2, hence, event $\bar{e}_2(n_2)$.
[4]http://www.swi-prolog.org/

$$
\begin{array}{llll}
v & ::= & l \mid \{k_1{:}v_1,\ldots,k_n{:}v_n\} \mid [v_1,\ldots,v_n] & \text{(data value)} \\
b & ::= & x \mid l \mid \{k_1{:}b_1,\ldots,k_n{:}b_n\} \mid [b_1,\ldots,b_n] & \text{(basic data expression)} \\
\theta & ::= & \tau(b_1,\ldots,b_n) & \text{(event type pattern)} \\
t & ::= & \varepsilon & \text{(empty trace)} \\
& & \theta & \text{(single event)} \\
& & \mid t_1 \cdot t_2 & \text{(concatenation)} \\
& & \mid t_1 \wedge t_2 & \text{(intersection)} \\
& & \mid t_1 \vee t_2 & \text{(union)} \\
& & \mid t_1 \mid t_2 & \text{(shuffle)} \\
& & \mid \{\text{let } x; t\} & \text{(parametric expression)}
\end{array}
$$

Figure 1: Syntax of the RML trace calculus: θ is defined inductively, t is defined coinductively on the set of cyclic terms.

is provided by the notion of *event type pattern*, an expression consisting of a name τ of an *event type*, applied to arguments which are *basic data expressions* denoting either variables or the data values (of primitive, array, or object type) associated with the events perceived by the monitor. An event type is a predicate which defines a possibly infinite set of events; an event type pattern specifies the set of events that are expected to occur at a certain point in the event trace; since event type patterns can contain variables, upon a successful match a substitution is computed to bind the variables of the pattern with the data values carried by the matched event.

RML is based on a general object model where events are represented as JavaScript object literals; for instance, the event type open(*fd*) of arity 1 may represent all events stating 'function call `fs.open` has returned file descriptor *fd*' and having shape {event:'func_post', name:'fs.open', res:*fd*}. The argument *fd* consists of the file descriptor (an integer value) returned by a call to `fs.open`. The definition is parametric in the variable *fd* which can be bound only when the corresponding event is matched with the information of the file descriptor associated with the property `res`; for instance, open(42) matches all events of shape {event:'func_post', name:'fs.open', res:42}, that is, all returns from call to `fs.open` with value 42.

Despite RML offers to the users the possibility to define the event types that are used in the specification, for simplicity the calculus is independent of the language used to define event types; correspondingly, the definition of the rewriting system of the calculus is parametric in the relation *match* assigning a semantics to event types (see below).

A specification is represented by a trace expression t built on top of the constant ε (denoting the singleton set with the empty trace), event type patterns θ (denoting the sets of all traces of length 1 with events matching θ), the binary operators (able to combine together sets of traces) of concatenation (juxtaposition), intersection ($\wedge$), union ($\vee$) and shuffle ($\mid$), and a let-construct to define the scope of variables used in event type patterns.

Differently from event type patterns, which are inductively defined terms, trace expressions are assumed to be cyclic (a.k.a. regular or rational) [23, 29, 5, 6] to provide an abstract support to recursion, since no explicit constructor is needed for it: the depth of a tree corresponding to a trace expression is allowed to be infinite, but the number of its different subtrees must be finite. This condition is proved to be equivalent [23] to requiring that a trace expression can always be defined by a *finite* set[5] of possibly recursive syntactic equations.

---

[5]The internal representation of cyclic terms in SWI-Prolog is indeed based on such approach.

$$(\text{e-}\varepsilon)\frac{}{\vdash E(\varepsilon)} \qquad (\text{e-al})\frac{\vdash E(t_1) \quad \vdash E(t_2)}{\vdash E(t_1 \, op \, t_2)} \, op \in \{|,\cdot,\wedge\} \qquad (\text{e-or-l})\frac{\vdash E(t_1)}{\vdash E(t_1 \vee t_2)} \qquad (\text{e-or-r})\frac{\vdash E(t_2)}{\vdash E(t_1 \vee t_2)}$$

$$(\text{e-par})\frac{\vdash E(t)}{\vdash E(\{\text{let } x; t\})} \qquad (\text{single})\frac{}{\theta \xrightarrow{e} \varepsilon; \sigma} \, \sigma = match(e,\theta) \qquad (\text{or-l})\frac{t_1 \xrightarrow{e} t_1'; \sigma}{t_1 \vee t_2 \xrightarrow{e} t_1'; \sigma} \qquad (\text{or-r})\frac{t_1 \xrightarrow{e}\!\!\!\!\!/ \quad t_2 \xrightarrow{e} t_2'; \sigma}{t_1 \vee t_2 \xrightarrow{e} t_2'; \sigma}$$

$$(\text{and})\frac{t_1 \xrightarrow{e} t_1'; \sigma_1 \quad t_2 \xrightarrow{e} t_2'; \sigma_2}{t_1 \wedge t_2 \xrightarrow{e} t_1' \wedge t_2'; \sigma} \, \sigma = \sigma_1 \cup \sigma_2 \qquad (\text{shuffle-l})\frac{t_1 \xrightarrow{e} t_1'; \sigma}{t_1 \,|\, t_2 \xrightarrow{e} t_1' \,|\, t_2; \sigma} \qquad (\text{shuffle-r})\frac{t_1 \xrightarrow{e}\!\!\!\!\!/ \quad t_2 \xrightarrow{e} t_2'; \sigma}{t_1 \,|\, t_2 \xrightarrow{e} t_1 \,|\, t_2'; \sigma}$$

$$(\text{cat-l})\frac{t_1 \xrightarrow{e} t_1'; \sigma}{t_1 \cdot t_2 \xrightarrow{e} t_1' \cdot t_2; \sigma} \qquad (\text{cat-r})\frac{t_1 \xrightarrow{e}\!\!\!\!\!/ \quad t_2 \xrightarrow{e} t_2'; \sigma}{t_1 \cdot t_2 \xrightarrow{e} t_2'; \sigma} \vdash E(t_1) \qquad (\text{par-t})\frac{t \xrightarrow{e} t'; \sigma}{\{\text{let } x; t\} \xrightarrow{e} \sigma_{|x} t'; \sigma_{\backslash x}} \, x \in dom(\sigma)$$

$$(\text{par-f})\frac{t \xrightarrow{e} t'; \sigma}{\{\text{let } x; t\} \xrightarrow{e} \{\text{let } x; t'\}; \sigma} \, x \notin dom(\sigma) \qquad (\text{n-}\varepsilon)\frac{}{\varepsilon \xrightarrow{e}\!\!\!\!\!/} \qquad (\text{n-single})\frac{}{\theta \xrightarrow{e}\!\!\!\!\!/} \, match(e,\theta) \text{ undef} \qquad (\text{n-or})\frac{t_1 \xrightarrow{e}\!\!\!\!\!/ \quad t_2 \xrightarrow{e}\!\!\!\!\!/}{t_1 \vee t_2 \xrightarrow{e}\!\!\!\!\!/}$$

$$(\text{n-and-l})\frac{t_1 \xrightarrow{e}\!\!\!\!\!/}{t_1 \wedge t_2 \xrightarrow{e}\!\!\!\!\!/} \qquad (\text{n-and-r})\frac{t_2 \xrightarrow{e}\!\!\!\!\!/}{t_1 \wedge t_2 \xrightarrow{e}\!\!\!\!\!/} \qquad (\text{n-and})\frac{t_1 \xrightarrow{e} t_1'; \sigma_1 \quad t_2 \xrightarrow{e} t_2'; \sigma_2}{t_1 \wedge t_2 \xrightarrow{e}\!\!\!\!\!/} \, \sigma_1 \cup \sigma_2 \text{ undef}$$

$$(\text{n-shuffle})\frac{t_1 \xrightarrow{e}\!\!\!\!\!/ \quad t_2 \xrightarrow{e}\!\!\!\!\!/}{t_1 \,|\, t_2 \xrightarrow{e}\!\!\!\!\!/} \qquad (\text{n-cat-l})\frac{t_1 \xrightarrow{e}\!\!\!\!\!/}{t_1 \cdot t_2 \xrightarrow{e}\!\!\!\!\!/} \not\vdash E(t_1) \qquad (\text{n-cat-r})\frac{t_1 \xrightarrow{e}\!\!\!\!\!/ \quad t_2 \xrightarrow{e}\!\!\!\!\!/}{t_1 \cdot t_2 \xrightarrow{e}\!\!\!\!\!/} \qquad (\text{n-par})\frac{t \xrightarrow{e}\!\!\!\!\!/}{\{\text{let } x; t\} \xrightarrow{e}\!\!\!\!\!/}$$

Figure 2: Transition system for the trace calculus.

Since event type patterns are inductive terms, the definition of free variables for them is standard.

Definition 3.1 *The set of free variables $pfv(\theta)$ occurring in an event type pattern θ is inductively defined as follows:*

$$pfv(x) = \{x\} \qquad pfv(l) = \emptyset$$
$$pfv(\tau(b_1,\ldots,b_n)) = pfv(\{k_1{:}b_1,\ldots,k_n{:}b_n\}) = pfv([b_1,\ldots,b_n]) = \bigcup_{i=1\ldots n} pfv(b_i)$$

Given their cyclic nature, a similar inductive definition of free variables for trace expressions does not work; for instance, if $t = \text{open}(fd) \cdot t$, a definition of fv given by induction on trace expressions would work only for non-cyclic terms and would be undefined for $fv(t)$. Unfortunately, neither a coinductive definition could work correctly since the set S returned by $fv(t)$ has to satisfy the equation $S = \{fd\} \cup S$ which has infinitely many solutions; hence, while an inductive definition of fv leads to a partial function which is undefined for all cyclic terms, a coinductive definition results in a non-functional relation fv; luckily, such a relation always admits the "least solution" which corresponds to the intended semantics.

Fact 3.1 *Let p be the predicate on trace expressions and set of variables, coinductively defined as follows:*

$$\frac{}{p(\varepsilon,\emptyset)} \qquad \frac{}{p(\theta,S)} \, pfv(\theta) = S \qquad \frac{p(t,S)}{p(\{\text{let } x; t\},S \setminus \{x\})} \qquad \frac{p(t_1,S_1) \quad p(t_2,S_2)}{p(t_1 \, op \, t_2, S_1 \cup S_2)} \, op \in \{|,\cdot,\wedge,\vee\}$$

Then, for any trace expression t, if $L = \bigcap\{S \mid p(t,S) \text{ holds}\}$, then $p(t,L)$ holds.

Proof: By case anaysis on t and coinduction on the definition of $p(t,S)$. □

Definition 3.2 *The set of free variables $fv(t)$ occurring in a trace expression is defined by $fv(t) = \bigcap\{S \mid p(t,S) \text{ holds}\}$.*

Semantics.   The semantics of the calculus depends on three judgments, inductively defined by the inference rules in Figure 2. Events e range over a fixed universe of events $\mathscr{E}$. The judgment $\vdash E(t)$ is derivable iff t accepts the empty trace λ and is auxiliary to the definition of the other two judgments $t_1 \xrightarrow{e} t_2; \sigma$ and $t \xrightarrow{e}\!\!\!\!\!/\,$; the rules defining it are straightforward and are independent from the remaining judgments, hence a stratified approach is followed and $\vdash E(t)$ and its negation $\nvdash E(t)$ are safely used in the side conditions of the rules for $t_1 \xrightarrow{e} t_2; \sigma$ and $t \xrightarrow{e}\!\!\!\!\!/\,$ (see below).

The judgment $t_1 \xrightarrow{e} t_2; \sigma$ defines the single reduction steps of the labeled transition system on which the semantics of the calculus is based; $t_1 \xrightarrow{e} t_2; \sigma$ is derivable iff the event e can be consumed, with the generated substitution σ, by the expression t_1, which then reduces to t_2. The judgment $t \xrightarrow{e}\!\!\!\!\!/\,$ is derivable iff there are no reduction steps for event e starting from expression t and is needed to enforce a deterministic semantics and to guarantee that the rules are monotonic and, hence, the existence of the least fixed-point; the definitions of the two judgments are mutually recursive.

Substitutions are finite partial maps from variables to data values which are produced by successful matches of event type patterns; the domain of σ and the empty substitution are denoted by $dom(\sigma)$ and $\emptyset$, respectively, while $\sigma_{|x}$ and $\sigma_{\setminus x}$ denote the substitutions obtained from σ by restricting its domain to $\{x\}$ and removing x from its domain, respectively. We simply write $t_1 \xrightarrow{e} t_2$ to mean $t_1 \xrightarrow{e} t_2; \emptyset$. Application of a substitution σ to an event type patter θ is denoted by $\sigma\theta$, and defined by induction on θ:

$$\sigma x = \sigma(x) \text{ if } x \in dom(\sigma), \ \sigma x = x \text{ otherwise} \qquad \sigma l = l$$
$$\sigma\{k_1{:}b_1, \ldots, k_n{:}b_n\} = \{k_1{:}\sigma b_1, \ldots, k_n{:}\sigma b_n\} \qquad \sigma[b_1, \ldots, b_n] = [\sigma b_1, \ldots, \sigma b_n]$$
$$\sigma\tau(b_1, \ldots, b_n) = \tau(\sigma b_1, \ldots, \sigma b_n)$$

Application of a substitution σ to a trace expression t is denoted by σt, and defined by coinduction on t:

$$\sigma\varepsilon = \varepsilon \qquad \sigma\theta = \sigma\tau(b_1, \ldots, b_n) \text{ if } \theta = \tau(b_1, \ldots, b_n)$$
$$\sigma(t_1 \ op \ t_2) = \sigma t_1 \ op \ \sigma t_2 \text{ for } op \in \{\cdot, \wedge, \vee, |\} \qquad \sigma\{\text{let } x; t\} = \{\text{let } x; \ \sigma_{\setminus x} t\}$$

Since the calculus does not cover event type definitions, the semantics of event types is parametric in the auxiliary partial function *match*, used in the side condition of rules (prefix) and (n-prefix): $match(e, \theta)$ returns the substitution σ iff event e matches event type $\sigma\theta$ and fails (that is, is undefined) iff there is no substitution σ for which e matches $\sigma\theta$. The substitution is expected to be the most general one and, hence, its domain to be included in the set of free variables in θ (see Def. 3.1).

As an example of how *match* could be derived from the definitions of event types in RML, if we consider again the event type open(*fd*) and $e=\{\text{event:'func\_post'}, \text{ name:'fs.open'}, \text{ res:42}\}$, then $match(e, \text{open(fd)}) = \{\text{fd} \mapsto 42\}$, while $match(e, \text{open(23)})$ is undefined.

Except for intersection, which is intrinsically deterministic since both operands need to be reduced, the rules defining the semantics of the other binary operators depend on the judgment $t \xrightarrow{e}\!\!\!\!\!/\,$ to force determinism; in particular, the judgment is used to ensure a left-to-right evaluation strategy: reduction of the right operand is possible only if the left hand side cannot be reduced.

The side condition of rule (and) uses the partial binary operator $\cup$ to merge substitutions: $\sigma_1 \cup \sigma_2$ returns the union of σ_1 and σ_2, if they coincide on the intersection of their domains, and is undefined otherwise.

Rule (cat-r) uses the judgment $E(t_1)$ in its side condition: event e consumed by t_2 can also be consumed by $t_1 \cdot t_2$ only if e is not consumed by t_1 (premise $t_1 \xrightarrow{e}\!\!\!\!\!/\,$ forcing left-to-right deterministic reduction), and the empty trace is accepted by t_1 (side condition $\vdash E(t_1)$).

Rule (par-t) can be applied when variable x is in the domain of the substitution σ generated by the reduction step from t to t': the substitution $\sigma_{|x}$ restricted to x is applied to t', and x is removed from the

domain of σ, together with its corresponding declaration. If x is not in the domain of σ (rule (par-f)), no substitution and no declaration removal is performed.

The rules defining $t \overset{e}{\nrightarrow}$ are complementary to those for $t \overset{e}{\rightarrow} t'$, and the definition of $t \overset{e}{\nrightarrow}$ depends on the judgment $t \overset{e}{\rightarrow} t'$ because of rule (n-and): there are no reduction steps for event e starting from expression $t_1 \wedge t_2$, even when $t_1 \overset{e}{\rightarrow} t_1'; \sigma_1$ and $t_2 \overset{e}{\rightarrow} t_2'; \sigma_2$ are derivable, if the two generated substitutions σ_1 and σ_2 cannot be successfully merged together; this happens when there are two event type patterns that match event e for two incompatible values of the same variable.

Let us consider an example of a cyclic term with the let-construct: $t = \{\text{let } fd; open(fd) \cdot close(fd) \cdot t\}$. The trace expression declares a local variable fd (the file descriptor), and requires that two immediately subsequent open and close events share the same file descriptor. Since the recursive occurrence of t contains a nested let-construct, the subsequent open and close events can involve a different file descriptor, and this can happen an infinite number of times. In terms of derivation, starting from t, if the event `{event:'func_post', name:'fs.open', res:42}` is observed, which matches $open(42)$, then the substitution $\{fd \mapsto 42\}$ is computed. As a consequence, the residual term $close(42) \cdot t$ is obtained, by substituting fd with 42 and removing the let-block. After that, the only valid event which can be observed is `{event:'func_pre',name:'close',args:[42]}`, matching $close(fd)$. Thus, after this rewriting step we get t again; the behavior continues as before, but a different file descriptor can be matched because of the let-block which hides the outermost declaration of fd; indeed, the substitution is not propagated inside the nested let-block. Differently from t, the term $\{\text{let } fd; t'\}$ with $t' = open(fd) \cdot close(fd) \cdot t'$ would require all open and close events to match a unique global file descriptor. As further explained in Section 5, such example shows how the let-construct is a solution more flexible than the mechanism of trace slicing used in other RV tools to achieve parametricty.

The following lemma can be proved by induction on the rules defining $t \overset{e}{\rightarrow} t'; \sigma$.

Lemma 3.1 *If $t \overset{e}{\rightarrow} t'; \sigma$ is derivable, then $dom(\sigma) \cup fv(t') \subseteq fv(t)$.*

Since trace expressions are cyclic, they can only contain a finite set of free variables, therefore the domains of all substitutions generated by a possibly infinite sequence of consecutive reduction steps starting from t are all contained in $fv(t)$.

3.1 Semantics based on the transition system

The reduction rules defined above provide the basis for the semantics of the calculus; because of computed substitutions and free variables, the semantics of a trace expression is not just a set of event traces: every accepted trace must be equipped with a substitution specifying how variables have been instantiated during the reduction steps. We call it an *instantiated event trace*; this can be obtained from the pairs of event and substitution traces yield by the possibly infinite reduction steps, by considering the disjoint union of all returned substitutions. Such a notion is needed[6] to allow a compositional semantics. The notion of substitution trace can be given in an analogous way as done for event traces in Section 2. By the considerations related to Lemma 3.1, the substitution associated with an instantiated event trace has always a finite domain, even when the trace is infinite; this means that the substitution is always fully defined after a finite number of reduction steps.

Definition 3.3 *A concrete instantiated event trace over the event universe $\mathcal{E}$ is a pair $(\bar{e}, \overline{\sigma})$ of event traces over $\mathcal{E}$, and substitution traces s.t. either $\bar{e}$ and $\overline{\sigma}$ are both infinite, or they are both finite and have the same length, all the substitutions in $\overline{\sigma}$ have mutually disjoint domains and $\bigcup\{dom(\sigma') \mid \sigma' \in \bar{\sigma}\}$ is finite.*

---

[6]See the example in Section 4.

An abstract instantiated event trace *(instantiated event trace, for short) over $\mathscr{E}$ is a pair $(\bar{e}, \sigma)$ where $\bar{e}$ is an event trace over $\mathscr{E}$ and σ is a substitution. We say that $(\bar{e}, \sigma)$ is derived from the concrete instantiated event trace $(\bar{e}, \bar{\sigma})$, written $(\bar{e}, \bar{\sigma}) \rightsquigarrow (\bar{e}, \sigma)$, iff $\sigma = \bigcup \{\sigma' \mid \sigma' \in \bar{\sigma}\}$.*

In the rest of the paper we use the meta-variable $\mathscr{I}$ to denote sets of instantiated event traces. We use the notations $\mathscr{I}\downarrow_1$ and $\mathscr{I}\downarrow_2$ to denote the two projections $\{\bar{e} \mid (\bar{e}, \sigma) \in \mathscr{I}\}$ and $\{\sigma \mid (\bar{e}, \sigma) \in \mathscr{I}\}$, respectively; we write $\bar{e} \triangleleft \mathscr{I}$ to mean $\bar{e} \triangleleft \mathscr{I}\downarrow_1$. The notation $\mathscr{I}\downarrow_\omega$ denotes the set $\{(\bar{e}, \sigma) \mid (\bar{e}, \sigma) \in \mathscr{I}, \bar{e}\ infinite\}$ restricted to infinite traces.

We can now define the semantics of trace expressions.

Definition 3.4 *The concrete semantics $[\![t]\!]_c$ of a trace expression t is the set of concrete instantiated event traces coinductively defined as follows:*

- *$(\lambda, \lambda) \in [\![t]\!]_c$ iff $\vdash E(t)$ is derivable;*

- *$(e \cdot \bar{e}, \sigma \cdot \bar{\sigma}) \in [\![t]\!]_c$ iff $t \xrightarrow{e} t'$; σ is derivable and $(\bar{e}, \bar{\sigma}) \in [\![\sigma t']\!]_c$.*

The (abstract) semantics $[\![t]\!]$ of a trace expression t is the set of instantiated event traces $\{(\bar{e}, \sigma) \mid (\bar{e}, \bar{\sigma}) \in [\![t]\!]_c, (\bar{e}, \bar{\sigma}) \rightsquigarrow (\bar{e}, \sigma)\}$.

The following propositions show that the concrete semantics of a trace expression t as given in Definition 3.4 is always well-defined.

Proposition 3.1 *If $(\bar{e}, \bar{\sigma}) \in [\![t]\!]_c$ and $\bar{e}$ is finite, then $|\bar{e}| = |\bar{\sigma}|$.*

Proposition 3.2 *If $(\bar{e}, \bar{\sigma}) \in [\![t]\!]_c$ and $\bar{e}$ is infinite, then $\bar{\sigma}$ is infinite as well.*

Proposition 3.3 *If $(\bar{e}, \bar{\sigma}) \in [\![t]\!]_c$, then for all $n, m \in \mathbb{N}$, $n \neq m$ implies $dom(\bar{\sigma}(n)) \cap dom(\bar{\sigma}(m)) = \emptyset$.*

Proposition 3.4 *If $(\bar{e}, \bar{\sigma}) \in [\![t]\!]_c$, then for all $n \in \mathbb{N}$ $dom(\bar{\sigma}(n)) \subseteq fv(t)$.*

4 Towards a compositional semantics

In this section we show how each basic trace expression operator can be interpreted as an operation over sets of instantiated event traces and we formally prove that such an interpretation is equivalent to the semantics derived from the transition system of the calculus in Definition 3.4, if one considers only contractive terms.

4.1 Composition operators

Left-preferential union: The left-preferential union $\mathscr{I}_1 \overset{\leftarrow}{\bigvee} \mathscr{I}_2$ of sets of instantiated event traces $\mathscr{I}_1$ and $\mathscr{I}_2$ is defined as follows: $\mathscr{I}_1 \overset{\leftarrow}{\bigvee} \mathscr{I}_2 = \mathscr{I}_1 \bigcup \{(\bar{e}, \sigma) \in \mathscr{I}_2 \mid \bar{e} = \lambda\ \text{or}\ (\bar{e} = e \cdot \bar{e}', e \not\triangleleft \mathscr{I}_1)\}$.

In the deterministic left-preferential version of union, instantiated event traces in $\mathscr{I}_2$ are kept only if they start with an event which is not the first element of any of the traces in $\mathscr{I}_1$ (the condition vacuously holds for the empty trace); since reduction steps can involve only one of the two operands at time, no restriction on the substitutions of the instantiated event traces is required.

Left-preferential concatenation: The left-preferential concatenation $\mathscr{I}_1 \stackrel{\leftarrow}{\cdot} \mathscr{I}_2$ of sets of instantiated event traces $\mathscr{I}_1$ and $\mathscr{I}_2$ is defined as follows: $\mathscr{I}_1 \stackrel{\leftarrow}{\cdot} \mathscr{I}_2 = \mathscr{I}_1{\downarrow}_\omega \cup \{(\bar{e}_1 \cdot \bar{e}_2, \sigma) \mid (\bar{e}_1, \sigma_1) \in \mathscr{I}_1, (\bar{e}_2, \sigma_2) \in \mathscr{I}_2, \sigma = \sigma_1 \cup \sigma_2, (\bar{e}_2 = \lambda \text{ or } (\bar{e}_2 = e \cdot \bar{e}_3, (\bar{e}_1 \cdot e) \not\prec \mathscr{I}_1))\}$.

The left operand $\mathscr{I}_1{\downarrow}_\omega$ of the union corresponds to the fact that in the deterministic left-preferential version of concatenation, all infinite instantiated event traces in $\mathscr{I}_1$ belong to the semantics of concatenation. The right operand of the union specifies the behavior for all finite instantiated event traces $\bar{e}_1$ in $\mathscr{I}_1$; in such cases, the trace in $\mathscr{I}_1 \stackrel{\leftarrow}{\cdot} \mathscr{I}_2$ can continue with $\bar{e}_2$ in $\mathscr{I}_2$ if $\bar{e}_1$ is not allowed to continue in $\mathscr{I}_1$ with the first event e of $\bar{e}_2$ ($(\bar{e}_1 \cdot e) \not\prec \mathscr{I}_1$, the condition vacuously holds if $\bar{e}_2$ is the empty trace). Since the reduction steps corresponding to $\bar{e}_2$ follow those for $\bar{e}_1$, the overall substitution σ must meet the constraint $\sigma = \sigma_1 \cup \sigma_2$ ensuring that σ_1 and σ_2 match on the shared variables of the two operands.

Intersection: The intersection $\mathscr{I}_1 \bigwedge \mathscr{I}_2$ of sets of instantiated event traces $\mathscr{I}_1$ and $\mathscr{I}_2$ is defined as follows: $\mathscr{I}_1 \bigwedge \mathscr{I}_2 = \{(\bar{e}, \sigma) \mid (\bar{e}, \sigma_1) \in \mathscr{I}_1, (\bar{e}, \sigma_2) \in \mathscr{I}_2, \sigma = \sigma_1 \cup \sigma_2\}$.

Since intersection is intrinsically deterministic, its semantics throws no surprise.

Left-preferential shuffle: The left-preferential shuffle $\mathscr{I}_1 \stackrel{\leftarrow}{\mid} \mathscr{I}_2$ of sets of instantiated event traces $\mathscr{I}_1$ and $\mathscr{I}_2$ is defined as follows: $\mathscr{I}_1 \stackrel{\leftarrow}{\mid} \mathscr{I}_2 = \{(\bar{e}, \sigma) \mid (\bar{e}_1, \sigma_1) \in \mathscr{I}_1, (\bar{e}_2, \sigma_2) \in \mathscr{I}_2, \sigma = \sigma_1 \cup \sigma_2, \bar{e} \in \bar{e}_1 \leftarrow|_{\mathscr{I}_1{\downarrow}_1} \bar{e}_2\}$.

The definition is based on the generalized left-preferential shuffle defined in Section 2; an event in $\bar{e}_2$ at a certain position n can contribute to the shuffle only if no trace in $\mathscr{I}_1{\downarrow}_1$ could contribute with the same event at the same position n. Since the reduction steps corresponding to $\bar{e}_1$ and $\bar{e}_2$ are interleaved, the overall substitution σ must meet the constraint $\sigma = \sigma_1 \cup \sigma_2$ ensuring that σ_1 and σ_2 match on the shared variables of the two operands.

Variable deletion: The deletion $\mathscr{I}_{\backslash x}$ of x from the set of instantiated event traces $\mathscr{I}$ is defined as follows: $\mathscr{I}_{\backslash x} = \{(\bar{e}, \sigma_{\backslash x}) \mid (\bar{e}, \sigma) \in \mathscr{I}\}$.

As expected, variable deletion only affects the domain of the computed substitution.

The definitions above show that instantiated event traces are needed to allow a compositional semantics; let us consider the following simplified variation of the example given in Section 3: $t' = \{\text{let } fd; open(fd) \cdot close(fd)\}$. If we did not keep track of substitutions, then the compositional semantics of $open(fd)$ and $close(fd)$ would contain all traces of length 1 matching $open(fd)$ and $close(fd)$, respectively, for any value fd, and, hence, the semantics of $open(fd) \cdot close(fd)$ could not constrain $open$ and $close$ events to be on the same file descriptor. Indeed, such a constraint is obtained by checking that the substitution of the event trace matching $open(fd)$ can be successfully merged with the substitution of the event trace matching $close(fd)$, so that the two substitutions agree on fd.

4.2 Contractivity

Contractivity is a condition on trace expressions which is statically enforced by the RML compiler; such a requirement avoids infinite loops when an event does not match the specification and the generated monitor would try to build an infinite derivation. Although the generated monitors could dynamically check potential loops dynamically, a syntactic condition enforced statically by the compiler allows monitors to be relieved of such a check, and, thus, to be more efficient.

Contractivity can be seen as a generalization of absence of left recursion in grammars [37]; loops in cyclic terms are allowed only if they are all guarded by a concatenation where the left operand t

cannot contain the empty trace (that is, $\nvdash E(t)$ holds), and the loop continues in the right operand of the concatenation. If such a condition holds, then it is not possible to build infinite derivations for $t_1 \xrightarrow{e} t_2$.

Interestingly enough, such a condition is also needed to prove that the interpretation of operators as given in Section 4.1 is equivalent to the semantics given in Definition 3.4. Indeed, the equivalence result proved in Theorem 4.1 is based on Lemma 4.1 stating that for all contractive term t_1 and event e, there exist t_2 and σ s.t. $t_1 \xrightarrow{e} t_2; \sigma$ is derivable if and only if $t_1 \xrightarrow{e}\!\!\!\!/$ is not derivable; such a claim does not hold for a non contractive term as $t = t \vee t$, because for all e, t' and σ, $t \xrightarrow{e} t'; \sigma$ and $t \xrightarrow{e}\!\!\!\!/$ are not derivable. This is due to the fact that both judgments are defined by an inductive inference system. Intuitively, from a contractive term we cannot derive a new term without passing through at least one concatenation. For instance, considering the term $t = e \cdot t$, we have contractivity because we have to consume e before going inside the loop. But, if we swap the operands, we obtain instead $t = t \cdot e$, where contractivity does not hold; in fact, deriving the concatenation we go first inside the head, but it is cyclic. Since the $\rightarrow$ and $\nrightarrow$ judgements are defined inductively, both are not derivable because a finite derivation tree cannot be derived for neither of them.

Definition 4.1 *Syntactic contexts $\mathscr{C}$ are inductively defined as follows:*

$$\mathscr{C} \quad ::= \quad \Box \mid \mathscr{C} \, op \, t \mid t \, op \, \mathscr{C} \mid \{\text{let } x; \mathscr{C}\} \quad \text{with } op \in \{\wedge, \vee, |, \cdot\}$$

Definition 4.2 *A syntactic context $\mathscr{C}$ is contractive if one of the following conditions hold:*

- $\mathscr{C} = \{\text{let } x; \mathscr{C}'\}$ *and $\mathscr{C}'$ is contractive;*
- $\mathscr{C} = \mathscr{C}' \, op \, t$, *$\mathscr{C}'$ is contractive and $op \in \{\cdot, \wedge, \vee, |\}$;*
- $\mathscr{C} = t \, op \, \mathscr{C}'$, *$\mathscr{C}'$ is contractive and $op \in \{\wedge, \vee, |\}$;*
- $\mathscr{C} = t \cdot \mathscr{C}'$, *$\vdash E(t)$ and $\mathscr{C}'$ is contractive;*
- $\mathscr{C} = t \cdot \mathscr{C}'$ *and $\nvdash E(t)$.*

Definition 4.3 *A term is part of t iff it belongs to the least set $partof(t)$ matching the following definition:*

$$partof(\varepsilon) = partof(\theta) = \emptyset \quad partof(\{\text{let } x; t\}) = \{t\} \cup partof(t)$$
$$partof(t_1 \, op \, t_2) = \{t_1, t_2\} \cup partof(t_1) \cup partof(t_2) \, for \, op \in \{|, \cdot, \wedge, \vee\}$$

Because trace expressions can be cyclic, the definition of *partof* follows the same pattern adopted for *fv*. One can prove that a term t is cyclic iff there exists $t' \in partof(t)$ s.t. $t' \in partof(t')$.

Definition 4.4 *A term t is* contractive *iff the following conditions old:*

- *for any syntactic context $\mathscr{C}$, if $t = \mathscr{C}[t]$ then $\mathscr{C}$ is contractive;*
- *for any term t', if $t' \in partof(t)$, then t' is contractive.*

4.3 Main Theorem

We first list all the auxiliary lemmas used to prove Theorem 4.1.

Lemma 4.1 *For all contractive term t_1 and event e, there exist t_2 and σ s.t. $t_1 \xrightarrow{e} t_2; \sigma$ is derivable if and only if $t_1 \xrightarrow{e}\!\!\!\!/$ is not derivable.*

Lemma 4.2 *If $(\bar{e}, \bar{\sigma}) \rightsquigarrow (\bar{e}, \sigma)$, then $(\bar{e}, \bar{\sigma}_{\backslash x}) \rightsquigarrow (\bar{e}, \sigma_{\backslash x})$.*

Where $\bar{\sigma}_{\backslash x}$ denotes the substitution sequence where x is removed from the domain of each substitution in $\bar{\sigma}$.

Lemma 4.3 *Given a substitution function σ and a term t, we have that $\sigma t = \sigma_{\backslash x}\sigma_{|x}t = \sigma_{|x}\sigma_{\backslash x}t$, for every $x \in dom(\sigma)$.*

Lemma 4.4 *Let t be a term, σ_1 be a substitution function s.t. $dom(\sigma_1) = \{x\}$; we have that:*

$$\forall_{(\bar{e},\sigma_2)\in[\![t]\!]}.((\sigma_1 \cup \sigma_2 \text{ is defined}) \implies (\bar{e},\sigma_{2\backslash x}) \in [\![\sigma_1 t]\!])$$

Lemma 4.5 *Let t be a term, σ_1 be a substitution function s.t. $dom(\sigma_1) = \{x\}$; we have that:*

$$\forall_{(\bar{e},\sigma_2)\in[\![\sigma_1 t]\!]}.((\sigma_1 \cup \sigma_2 \text{ is defined}) \implies (\bar{e},\sigma_2) \in [\![t]\!])$$

Lemma 4.6 $t \xrightarrow{e} \!\!\!\!/ \iff e \not\triangleleft [\![t]\!]$.

Lemma 4.7 *If $(\bar{e},\sigma) \in [\![t]\!]$, then $(\bar{e},\emptyset) \in [\![\sigma t]\!]$.*

Lemma 4.8 *If $(\bar{e},\bar{\sigma}) \in [\![t]\!]_c$ and $\bar{e}$ is infinite, then $(\bar{e},\bar{\sigma}) \in [\![t \cdot t']\!]_c$ for every t'.*

Lemma 4.9 *If $e \cdot \bar{e} \in \bar{e}_1 \leftarrow|_T \bar{e}_2$, then $\bar{e}_1 = e \cdot \bar{e}'_1$, or $\bar{e}_2 = e \cdot \bar{e}'_2$ and $e \not\triangleleft \bar{e}_1$.*

Lemma 4.10 *If $(\bar{e},\bar{\sigma}) \in [\![t]\!]_c$ and $E(t')$, then $(\bar{e},\bar{\sigma}) \in [\![t \cdot t']\!]_c$.*

Lemma 4.11 *Given $(\bar{e}_1,\bar{\sigma}_1) \in [\![t_1]\!]_c$, $t_2 \xrightarrow{e_2} t_2^1;\sigma_2^1$ and $(\bar{e}_2,\bar{\sigma}_2^2) \in [\![\sigma_2^1 t_2^1]\!]_c$ with $\bar{\sigma}_2 = \sigma_2^1 \cdot \bar{\sigma}_2^2$. If $\bar{e}_1 = e_1 \cdot \ldots \cdot e_n$ is finite, $t_1 \xrightarrow{e_1} t_1^1;\sigma_1^1$, $t_1^1 \xrightarrow{e_2} t_1^2;\sigma_1^2$, ..., $t_1^{n-1} \xrightarrow{e_n} t_1^n;\sigma_1^n$, with $\sigma_1 = \sigma_1^1 \cdot \ldots \cdot \sigma_1^n$ and $t_1^n \xrightarrow{e_2} \!\!\!\!/$, then $(\bar{e}_1 \cdot e_2 \cdot \bar{e}_2, \bar{\sigma}_1 \cdot \bar{\sigma}_2) \in [\![t_1 \cdot t_2]\!]_c$.*

In Theorem 4.1 we claim that for every operator of the trace calculus, the compositional semantics is equivalent to the abstract semantics. To prove such claim, we need to show that, for each operator, every trace belonging to the compositional semantics belongs to the abstract semantics, which means we only consider correct traces (*soundness*); and, every trace belonging to the abstract semantics belongs to the compositional semantics, which means we consider all the correct traces (*completeness*).

Each operator requires a customised proofs, but in principle, all the proofs follow the same reasoning. Both soundness and completeness proof start expanding the compositional semantics definition in terms of its concrete semantics, which in turn is rewritten in terms of the operational semantics. At this point, the compositional operator's operands can be separately analysed in order to be recombined with the corresponding trace calculus operator. Finally, the proofs are concluded going backwards from the operational semantics to the abstract one, through the concrete semantics. For all the operators, except $\vee$ and $\wedge$, the proofs are given by coinduction over the terms structure. In every proof which is not analysed separately ($\iff$ cases), we implicitly apply Lemma 4.1.

Theorem 4.1 *The following claims hold for all contractive terms t_1 and t_2:*

- $[\![t_1 \vee t_2]\!] = [\![t_1]\!] \overset{\leftarrow}{\vee} [\![t_2]\!]$
- $[\![t_1 \cdot t_2]\!] = [\![t_1]\!] \overset{\leftarrow}{\cdot} [\![t_2]\!]$
- $[\![t_1 \wedge t_2]\!] = [\![t_1]\!] \wedge [\![t_2]\!]$
- $[\![t_1 \,|\, t_2]\!] = [\![t_1]\!] \overset{\leftarrow}{|} [\![t_2]\!]$
- $[\![\{\text{let } x; t_1\}]\!] = [\![t_1]\!]_{\backslash x}$

The proofs for the union, intersection, shuffle and let cases are omitted and can be found in the extended version [10]. We decided not to report them due to space constraints. In the proofs that follow, we prove composed implications such as $A_1 \vee \ldots \vee A_n \implies B$, by splitting them into n separate implications $A_1 \implies^1 B, \ldots, A_n \implies^n B$.

The first operator we are going to analyse is the concatenation, where we are going to show that $(\bar{e}, \sigma) \in [\![t_1 \cdot t_2]\!] \iff (\bar{e}, \sigma) \in [\![t_1]\!] \overset{\leftarrow}{\cdot} [\![t_2]\!]$.

The proof for the empty trace is trivial, and is constructed on top of the definition of the E predicate.

$$
\begin{aligned}
(\lambda, \emptyset) \in [\![t_1 \cdot t_2]\!] &\iff (\lambda, \lambda) \in [\![t_1 \cdot t_2]\!]_c \wedge (\lambda, \lambda) \rightsquigarrow (\lambda, \emptyset) \text{ (by definition of } [\![t]\!]) \\
&\iff E(t_1 \cdot t_2) \text{ is derivable (by definition of } [\![t]\!]_c) \\
&\iff E(t_1) \text{ is derivable } \wedge E(t_2) \text{ is derivable (by definition of } E(t)) \\
&\iff (\lambda, \lambda) \in [\![t_1]\!]_c \wedge (\lambda, \lambda) \in [\![t_2]\!]_c \wedge (\lambda, \lambda) \rightsquigarrow (\lambda, \emptyset) \text{ (by definition of } [\![t]\!]_c) \\
&\iff (\lambda, \emptyset) \in [\![t_1]\!] \wedge (\lambda, \emptyset) \in [\![t_2]\!] \text{ (by definition of } [\![t]\!]) \\
&\iff (\lambda, \emptyset) \in ([\![t_1]\!] \overset{\leftarrow}{\cdot} [\![t_2]\!]) \text{ (by definition of } \overset{\leftarrow}{\cdot})
\end{aligned}
$$

When the trace is not empty, we present the procedure to prove *completeness* ($\implies$) and *soundness* ($\impliedby$), separately.

Let us start with *completeness*. To prove it, we have to show that the abstract semantics $[\![t_1 \cdot t_2]\!]$ (based on the original operational semantics) is included in the composition of the abstract semantics $[\![t_1]\!]$ and $[\![t_2]\!]$, using $\overset{\leftarrow}{\cdot}$ operator. More specifically, in the first part of the proof ($\implies^1$), the first event of the trace belongs to the head of the concatenation. Thus, the head is expanded through operational semantics, causing the term to be rewritten into a concatenation, where the head is substituted with a new term. Since the concrete semantics has been defined coinductively, we can conclude that the proof is satisfied by the so derived concatenation by coinduction. Finally, the proof is concluded recombining the new concatenation in terms of $\overset{\leftarrow}{\cdot}$. The second part of the proof ($\implies^2$) does not require coinduction, since the trace belongs to the tail of the concatenation. Through the operational semantics, the concatenation is rewritten into the new tail, and the proof is straightforwardly concluded following the abstract semantics.

$$
\begin{aligned}
(e \cdot \bar{e}, \sigma) \in [\![t_1 \cdot t_2]\!] &\implies (e \cdot \bar{e}, \bar{\sigma}) \in [\![t_1 \cdot t_2]\!]_c \wedge (e \cdot \bar{e}, \bar{\sigma}) \rightsquigarrow (e \cdot \bar{e}, \sigma) \text{ (by definition of } [\![t]\!]) \\
&\implies t_1 \cdot t_2 \overset{e}{\rightarrow} t'; \sigma' \text{ is derivable } \wedge (\bar{e}, \bar{\sigma}') \in [\![\sigma' t']\!]_c \text{ (by definition of } [\![t]\!]_c) \\
&\implies (t_1 \overset{e}{\rightarrow} t_1'; \sigma' \text{ is derivable } \wedge t_1 \cdot t_2 \overset{e}{\rightarrow} t_1' \cdot t_2; \sigma' \text{ is derivable } \wedge \\
&\quad (\bar{e}, \bar{\sigma}') \in [\![\sigma'(t_1' \cdot t_2)]\!]_c) \vee \\
&\quad (t_1 \overset{e}{\not\rightarrow} \wedge E(t_1) \wedge t_2 \overset{e}{\rightarrow} t_2'; \sigma' \text{ is derivable } \wedge t_1 \cdot t_2 \overset{e}{\rightarrow} t_2'; \sigma' \text{ is derivable } \wedge \\
&\quad (\bar{e}, \bar{\sigma}') \in [\![\sigma' t_2']\!]_c) \text{ (by operational semantics)} \\
&\implies^1 t_1 \overset{e}{\rightarrow} t_1'; \sigma' \text{ is derivable } \wedge t_1 \cdot t_2 \overset{e}{\rightarrow} t_1' \cdot t_2; \sigma' \text{ is derivable } \wedge \\
&\quad (\bar{e}, \sigma'') \in [\![\sigma'(t_1' \cdot t_2)]\!] \wedge (\bar{e}, \bar{\sigma}') \rightsquigarrow (\bar{e}, \sigma'') \wedge \sigma = \sigma'' \cup \sigma' \\
&\quad \text{(by definition of } [\![t]\!]) \\
&\implies^1 t_1 \overset{e}{\rightarrow} t_1'; \sigma' \text{ is derivable } \wedge t_1 \cdot t_2 \overset{e}{\rightarrow} t_1' \cdot t_2 \text{ is derivable } \wedge \\
&\quad (\bar{e}, \sigma'') \in [\![\sigma' t_1']\!] \overset{\leftarrow}{\cdot} [\![\sigma' t_2]\!] \wedge (\bar{e}, \bar{\sigma}') \rightsquigarrow (\bar{e}, \sigma'') \wedge \sigma = \sigma'' \cup \sigma' \\
&\quad \text{(by coinduction over } [\![t]\!]) \\
&\implies^1 t_1 \overset{e}{\rightarrow} t_1'; \sigma' \text{ is derivable } \wedge t_1 \cdot t_2 \overset{e}{\rightarrow} t_1' \cdot t_2 \text{ is derivable } \wedge
\end{aligned}
$$

$$(\bar{e}_1, \sigma_1'') \in [\![\sigma' t_1']\!] \wedge (\bar{e}_2, \sigma_2'') \in [\![\sigma' t_2]\!] \wedge \bar{e} = \bar{e}_1 \cdot \bar{e}_2 \wedge$$
$$(\bar{e}_2 = \lambda \vee (\bar{e}_2 = e' \cdot \bar{e}_3 \wedge \bar{e}_1 \cdot e' \not\preceq [\![\sigma' t_1']\!])) \text{ (by definition of } \overset{\leftarrow}{\cdot})$$
$$\implies^1 \quad t_1 \overset{e}{\to} t_1'; \sigma' \text{ is derivable } \wedge (\bar{e}_1, \bar{\sigma}_1) \in [\![\sigma' t_1']\!]_c \wedge (\bar{e}_1, \bar{\sigma}_1) \rightsquigarrow (\bar{e}_1, \sigma_1'') \wedge$$
$$(\bar{e}_2, \sigma_2'') \in [\![\sigma' t_2]\!] \wedge \bar{e} = \bar{e}_1 \cdot \bar{e}_2 \wedge$$
$$(\bar{e}_2 = \lambda \vee (\bar{e}_2 = e' \cdot \bar{e}_3 \wedge \bar{e}_1 \cdot e' \not\preceq [\![\sigma' t_1']\!])) \text{ (by definition of } [\![t]\!])$$
$$\implies^1 \quad (e \cdot \bar{e}_1, \sigma' \cdot \bar{\sigma}_1) \in [\![t_1]\!]_c \wedge (\bar{e}_1, \bar{\sigma}_1) \rightsquigarrow (\bar{e}_1, \sigma_1'') \wedge$$
$$(\bar{e}_2, \sigma_2'') \in [\![\sigma' t_2]\!] \wedge \bar{e} = \bar{e}_1 \cdot \bar{e}_2 \wedge$$
$$(\bar{e}_2 = \lambda \vee (\bar{e}_2 = e' \cdot \bar{e}_3 \wedge \bar{e}_1 \cdot e' \not\preceq [\![\sigma' t_1']\!])) \text{ (by definition of } [\![t]\!]_c)$$
$$\implies^1 \quad (e \cdot \bar{e}_1, \sigma_1'') \in [\![\sigma' t_1]\!] \wedge (\bar{e}_2, \sigma_2'' \cup \sigma') \in [\![t_2]\!] \wedge \bar{e} = \bar{e}_1 \cdot \bar{e}_2 \wedge$$
$$(\bar{e}_2 = \lambda \vee (\bar{e}_2 = e' \cdot \bar{e}_3 \wedge \bar{e}_1 \cdot e' \not\preceq [\![\sigma' t_1']\!]))$$
$$\text{(by definition of } [\![t]\!] \text{ and Lemma 4.7)}$$
$$\implies^1 \quad (e \cdot \bar{e}, \sigma) \in [\![t_1]\!] \overset{\leftarrow}{\cdot} [\![t_2]\!] \text{ (by definition of } \overset{\leftarrow}{\cdot})$$
$$\implies^2 \quad (e \cdot \bar{e}, \bar{\sigma}) \in [\![t_2]\!]_c \wedge (\lambda, \lambda) \in [\![t_1]\!]_c \wedge t_1 \overset{e}{\not\to} \text{ (by definition of } [\![t]\!]_c)$$
$$\implies^2 \quad (e \cdot \bar{e}, \sigma) \in [\![t_2]\!] \wedge (\lambda, \emptyset) \in [\![t_1]\!] \wedge t_1 \overset{e}{\not\to} \text{ (by definition of } [\![t]\!])$$
$$\implies^2 \quad (e \cdot \bar{e}, \sigma) \in [\![t_2]\!] \wedge (\lambda, \emptyset) \in [\![t_1]\!] \wedge (\lambda \cdot e) \not\preceq [\![t_1]\!] \text{ (by Lemma 4.6)}$$
$$\implies^2 \quad (e \cdot \bar{e}, \sigma) \in [\![t_1]\!] \overset{\leftarrow}{\cdot} [\![t_2]\!] \text{ (by definition of } \overset{\leftarrow}{\cdot})$$

We now prove *soundness*. To prove it, we show that the composition of abstract semantics $[\![t_1]\!]$ and $[\![t_2]\!]$ using the $\overset{\leftarrow}{\cdot}$ operator is included in the abstract semantics of the related concatenation term $[\![t_1 \cdot t_2]\!]$. The resulting proof is splitted in four separated cases. When the trace belongs to $[\![t_1]\!]$ is infinite ($\implies^1$). The proof is based on the fact that an infinite trace concatenated to another trace is always equal to itself. In all the other cases, the proof can be fully derived by a direct application of the operational semantics.

$$(e \cdot \bar{e}, \sigma) \in [\![t_1]\!] \overset{\leftarrow}{\cdot} [\![t_2]\!] \implies (e \cdot \bar{e}) \in [\![t_1]\!] \downarrow_\omega \vee$$
$$(e \cdot \bar{e} = \bar{e}_1 \cdot \bar{e}_2 \wedge (\bar{e}_1, \sigma_1) \in [\![t_1]\!] \wedge (\bar{e}_2, \sigma_2) \in [\![t_2]\!] \wedge \sigma = \sigma_1 \cup \sigma_2 \wedge$$
$$(\bar{e}_2 = \lambda \vee (\bar{e}_2 = e' \cdot \bar{e}_3 \wedge \bar{e}_1 \cdot e' \not\preceq [\![t_1]\!]))) \text{ (by definition of } \overset{\leftarrow}{\cdot})$$
$$\implies (e \cdot \bar{e}) \in [\![t_1]\!] \downarrow_\omega \vee$$
$$(\bar{e}_1 = \lambda \wedge (\lambda, \emptyset) \in [\![t_1]\!] \wedge (e \cdot \bar{e}, \sigma) \in [\![t_2]\!] \wedge e \not\preceq [\![t_1]\!]) \vee$$
$$(\bar{e}_2 = \lambda \wedge (e \cdot \bar{e}) \in [\![t_1]\!] \wedge (\lambda, \emptyset) \in [\![t_2]\!]) \vee$$
$$(\bar{e}_1 = e \cdot \bar{e}_1' \wedge \bar{e}_2 = e' \cdot \bar{e}_3 \wedge \bar{e}_1 \cdot e' \not\preceq [\![t_1]\!] \wedge$$
$$(e \cdot \bar{e}_1, \sigma_1) \in [\![t_1]\!] \wedge (e' \cdot \bar{e}_3) \in [\![t_2]\!] \wedge \sigma = \sigma_1 \cup \sigma_2)$$

$$(e \cdot \bar{e}, \sigma) \in [\![t_1]\!] \downarrow_\omega \implies^1 (e \cdot \bar{e}, \sigma) \in [\![t_1]\!] \wedge \bar{e} \text{ infinite (by definition of } \downarrow_\omega)$$
$$\implies^1 (e \cdot \bar{e}, \bar{\sigma}) \in [\![t_1]\!]_c \wedge (e \cdot \bar{e}, \bar{\sigma}) \rightsquigarrow (e \cdot \bar{e}, \sigma) \wedge$$
$$\bar{e} \text{ infinite (by definition of } [\![t]\!])$$
$$\implies^1 t_1 \overset{e}{\to} t_1'; \sigma' \text{ is derivable } \wedge (\bar{e}, \bar{\sigma}') \in [\![\sigma' t_1']\!]_c \wedge$$
$$(e \cdot \bar{e}, \bar{\sigma}) \rightsquigarrow (e \cdot \bar{e}, \sigma) \wedge \bar{e} \text{ infinite (by definition of } [\![t]\!]_c)$$
$$\implies^1 t_1 \overset{e}{\to} t_1'; \sigma' \text{ is derivable } \wedge (\bar{e}, \bar{\sigma}') \in [\![\sigma'(t_1' \cdot t_2)]\!]_c \wedge$$

$$(e \cdot \bar{e}, \bar{\sigma}) \rightsquigarrow (e \cdot \bar{e}, \sigma) \wedge \bar{e} \text{ infinite } \text{ (by Lemma 4.8)}$$
$$\implies^1 \quad t_1 \cdot t_2 \xrightarrow{e} t'_1 \cdot t_2; \sigma' \text{ is derivable } \wedge (\bar{e}, \bar{\sigma}') \in [\![\sigma'(t'_1 \cdot t_2)]\!]_c \wedge$$
$$(e \cdot \bar{e}, \bar{\sigma}) \rightsquigarrow (e \cdot \bar{e}, \sigma) \wedge \bar{e} \text{ infinite } \text{ (by operational semantics)}$$
$$\implies^1 \quad (e \cdot \bar{e}, \bar{\sigma}) \in [\![t_1 \cdot t_2]\!]_c \wedge (e \cdot \bar{e}, \bar{\sigma}) \rightsquigarrow (e \cdot \bar{e}, \sigma) \text{ (by definition of } [\![t]\!]_c)$$
$$\implies^1 \quad (e \cdot \bar{e}, \sigma) \in [\![t_1 \cdot t_2]\!] \text{ (by definition of } [\![t]\!])$$

$$(\bar{e}_1 = \lambda \wedge (\lambda, \emptyset) \in [\![t_1]\!] \wedge$$
$$(e \cdot \bar{e}, \sigma) \in [\![t_2]\!] \wedge e \not\preceq [\![t_1]\!]) \quad \implies^2 \quad E(t_1) \text{ is derivable } \wedge (e \cdot \bar{e}, \sigma) \in [\![t_2]\!] \wedge$$
$$e \not\preceq [\![t_1]\!] \text{ (by definition of } [\![t]\!])$$
$$\implies^2 \quad E(t_1) \text{ is derivable } \wedge t_2 \xrightarrow{e} t'_2; \sigma' \text{ is derivable } \wedge$$
$$(\bar{e}, \bar{\sigma}') \in [\![\sigma't'_2]\!]_c \wedge e \not\preceq [\![t_1]\!] \wedge (e \cdot \bar{e}, \bar{\sigma}) \rightsquigarrow (e \cdot \bar{e}, \sigma)$$
$$\text{(by definition of } [\![t]\!]_c)$$
$$\implies^2 \quad t_1 \cdot t_2 \xrightarrow{e} t'_2; \sigma' \text{ is derivable } \wedge (\bar{e}, \bar{\sigma}') \in [\![\sigma't'_2]\!]_c$$
$$\text{(by operational semantics)}$$
$$\implies^2 \quad (e \cdot \bar{e}, \bar{\sigma}) \in [\![t_1 \cdot t_2]\!]_c \wedge (e \cdot \bar{e}, \bar{\sigma}) \rightsquigarrow (e \cdot \bar{e}, \sigma)$$
$$\text{(by definition of } [\![t]\!]_c)$$
$$\implies^2 \quad (e \cdot \bar{e}, \sigma) \in [\![t_1 \cdot t_2]\!] \text{ (by definition of } [\![t]\!])$$

$$(\bar{e}_2 = \lambda \wedge (e \cdot \bar{e}) \in [\![t_1]\!] \wedge (\lambda, \emptyset) \in [\![t_2]\!]) \quad \implies^3 \quad (e \cdot \bar{e}, \sigma) \in [\![t_1 \cdot t_2]\!] \text{ (by Lemma 4.10)}$$

$$(\bar{e}_1 = e \cdot \bar{e}'_1 \wedge \bar{e}_2 = e' \cdot \bar{e}_3 \wedge$$
$$\bar{e}_1 \cdot e' \not\preceq [\![t_1]\!] \wedge (e \cdot \bar{e}_1, \sigma_1) \in [\![t_1]\!] \wedge$$
$$(e' \cdot \bar{e}_3, \sigma_2) \in [\![t_2]\!] \wedge \sigma = \sigma_1 \cup \sigma_2) \quad \implies^4 \quad t_1 \xrightarrow{e} t'_1; \sigma'_1 \text{ is derivable } \wedge (\bar{e}_1, \bar{\sigma}_1) \in [\![\sigma'_1 t_1]\!]_c \wedge$$
$$(\bar{e}_1, \bar{\sigma}_1) \rightsquigarrow (\bar{e}_1, \sigma''_1) \wedge t_2 \xrightarrow{e'} t'_2; \sigma'_2 \text{ is derivable } \wedge$$
$$(\bar{e}_3, \bar{\sigma}'_2) \in [\![\sigma'_2 t'_2]\!] \wedge (\bar{e}_2, \bar{\sigma}_2) \rightsquigarrow (\bar{e}_2, \sigma''_2) \wedge t_1 \xrightarrow{e'} \text{ is derivable}$$
$$\wedge \sigma_1 = \sigma'_1 \cup \sigma''_1 \wedge \sigma_2 = \sigma'_2 \cup \sigma''_2 \text{ (by operational semantics)}$$
$$\implies^4 \quad (\bar{e}_1 \cdot \bar{e}_2, \bar{\sigma}) \in [\![t_1 \cdot t_2]\!]_c \text{ (by Lemma 4.11)}$$

5 Related Work

Compositionality, determinism and events-based semantics are topics very central to concurrent systems. Winskel has introduced the notion of event structure [44] to model computational processes as sets of event occurrences together with relations representing their causal dependencies. Vaandrager [43] has proved that for concurrent deterministic systems it is sufficient to observe the beginning and end of events to derive its causal structure. Lynch and Tuttle have introduced input/output automata [36] to model concurrent and distributed discrete event systems with a trace semantics consisting of both finite and infinite sequences of actions.

   The rest of this section describes some of the main RV techniques and state-of-the-art tools and compares them with respect to RML; more comprehensive surveys on RV can be found in literature [25, 30, 35, 41, 26, 13, 31] which mention formalisms for parameterised runtime verification that have not deliberately presented here for space limitation.

Monitor-oriented programming: Similarly as RML, which does not depend on the monitored system and its instrumentation, other proposals introduce different levels of separation of concerns. *Monitor-oriented programming* (*MOP* [19]) is an infrastructure for RV that is neither tied to any particular programming language nor to a single specification language. In order to add support for new logics, one has to develop an appropriate plug-in converting specifications to one of the format supported by the MOP instance of the language of choice; the main formalisms implemented in existing MOP include finite state machines, extended regular expressions, context-free grammars and temporal logics. Finite state machines (or, equivalently, regular expressions) can be easily translated to RML, have limited expressiveness, but are widely used in RV because they are well-understood among software developers as opposite to other more sophisticated approaches, as temporal logics. Extended regular expressions include intersection and complement; although such operators allow users to write more compact specifications, they do not increase the formal expressive power since regular languages are closed under both. Deterministic Context-Free grammars (that is, deterministic pushdown automata) can be translated in RML using recursion, concatenation, union, and the empty trace, while the relationship with Context-Free grammars (that is, pushdown automata) has not been fully investigated yet; as stated in the introduction, RML can express several non Context-Free properties, hence RML cannot be less expressive than Context-Free grammars, but we do not know whether Context-Free grammars are less expressive than RML.

Temporal logics: Since RV has its roots in model checking, it is not surprising that logic-based formalism previously introduced in the context of the latter have been applied to the former. *Linear Temporal Logic* (*LTL*) [38], is one of the most used formalism in verification.

Since the standard semantics of LTL is defined on infinite traces only, and RV monitors can only check finite trace prefixes (as opposed to static formal verification), a three-valued semantics for LTL, named *LTL$_3$* has been proposed [15]. Beyond the basic "true" and "false" truth values, a third "inconclusive" one is considered (LTL specification syntax is unchanged, only the semantics is modified to take into account the new value). This allows one to distinguish the satisfaction/violation of the desired property ("false") from the lack of sufficient evidence among the events observed so far ("inconclusive"), making this semantics more suited to RV. Differently from LTL, the semantics of LTL$_3$ is defined on finite prefixes, making it more suitable for comparison with other RV formalisms. Further development of LTL$_3$ led to *RV-LTL* [14], a 4-valued semantics on which RML monitor verdicts are based on.

The expressive power of LTL is the same as of star-free ω-regular languages [39]. When restricted to finite traces, RML is much more expressive than LTL as any regular expression can be trivially translated to it; however, on infinite traces, the comparison is slightly more intricate since RML and LTL$_3$ have incomparable expressiveness [8]. There exist many extensions of LTL that deal with time in a more quantitative way (as opposed to the strictly qualitative approach of standard LTL) without increasing the expressive power, like *interval temporal logic* [18], *metric temporal logic* [42] and *timed LTL* [15]. Other proposals go beyond regularity [3] and even context-free languages [16].

Several temporal logics are embeddable in *recHML* [34], a variant of the *modal μ-calculus* [33]; this allows the formal study of monitorability [1] in a general framework, to derive results for free about any formalism that can be expressed in such calculi. It would be interesting to study whether the RML trace calculus could be derivable to get theoretical results that are missing from this presentation. Unfortunately, it is not clear whether our calculus and *recHML* are comparable at all. For instance, *recHML* is a fixed-point logic including both least and greatest fixpoint operators, while our calculus implicitly uses a greatest fixpoint semantics for recursion. Nonetheless, *recHML* does not include a shuffle operator, and we are not aware of a way to derive it from other operators.

Regardless of the formal expressiveness, RML and temporal logics are essentially different: RML is closer to formalisms with which software developers are more familiar, as regular expressions and Context-Free languages, but does not offer direct support for time; however, if the instrumentation provides timestamps, then some time-related properties can still be expressed exploiting parametricity.

State machines: As opposite to the language-based approach, as RML, specifications can be defined using *state machines* (a.k.a. automata or finite-state machines). Though the core concept of a finite set of states and a (possibly input-driven) transition function between them is always there, in the field of automata theory different formalizations and extensions bring the expressiveness anywhere from simple deterministic finite automata to Turing machines.

An example of such formalisms is *DATE* (Dynamic Automata with Timers and Events [21]), an extension of the finite-state automata computational model based on communicating automata with timers and transitions triggered by observed events. This is the basis of *LARVA* [22], a Java RV tool focused on control-flow and real-time properties, exploiting the expressiveness of the underlying system (DATE).

The main feature of LARVA that is missing in RML is the support for temporized properties, as observed events can trigger timers for other expected events. On the other hand, the parametric verification support of RML is more general. LARVA scope mechanism works at the object level, thus parametricity is based on *trace slicing* [31] and implemented by spawning new monitors and associating them with different objects. The RML approach is different as specifications can be parametric with respect to any observed data thanks to event type patterns and the let-construct to control the scope of the variables occurring in them. Limitations of the parametric trace slicing approach described above, as well as possible generalizations to overcome them, have been explored by [20, 12, 40].

Finally, the goals of the two tools are different: while RML strives to be system-independent, LARVA is devoted to Java verification, and the implementation relies on AspectJ [32] as an "instrumentation" layer allowing one to inject code (the monitor) to be executed at specific locations in the program.

6   Conclusion

We have moved a first step towards a compositional semantics of the RML trace calculus, by introducing the notion of instantiated event trace, defining the semantics of trace expressions in terms of sets of instantiated event traces and showing how each basic trace expression operator can be interpreted as an operation over sets of instantiated event traces; we have formally proved that such an interpretation is equivalent to the semantics derived from the transition system of the calculus if one considers only contractive terms.

For simplicity, here we have considered only the core of the calculus, but we plan to extend our result to the full calculus, which includes also the prefix closure operator and a top-level layer with constructs to support generic specifications [28]. Another interesting direction for further investigation consists in studying how the notion of contractivity influences the expressive power of the calculus and, hence, of RML; although we have failed so far to find a non-contractive term whose semantics is not equivalent to a corresponding contractive trace expression, we have not formally proved that contractivity does not limit the expressive power of the calculus.

References

[1] L. Aceto, A. Achilleos, A. Francalanza, A. Ingólfsdóttir & K. Lehtinen (2019): *Adventures in Monitorability: From Branching to Linear Time and Back Again*. Proc. ACM Program. Lang. 3(POPL), pp. 52:1–52:29, doi:10.1145/3290365.

[2] Wolfgang Ahrendt, Jesús Mauricio Chimento, Gordon J. Pace & Gerardo Schneider (2017): *Verifying data- and control-oriented properties combining static and runtime verification: theory and tools*. Formal Methods in System Design 51(1), pp. 200–265, doi:10.1007/s10703-017-0274-y.

[3] Rajeev Alur, Kousha Etessami & P. Madhusudan (2004): *A Temporal Logic of Nested Calls and Returns*. In: Tools and Algorithms for the Construction and Analysis of Systems, 10th International Conference, TACAS 2004, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2004, Barcelona, Spain, March 29 - April 2, 2004, Proceedings, pp. 467–481, doi:10.1007/978-3-540-24730-2_35.

[4] Davide Ancona, Viviana Bono, Mario Bravetti, Joana Campos, Giuseppe Castagna, Pierre-Malo Deniélou, Simon J. Gay, Nils Gesbert, Elena Giachino, Raymond Hu, Einar Broch Johnsen, Francisco Martins, Viviana Mascardi, Fabrizio Montesi, Rumyana Neykova, Nicholas Ng, Luca Padovani, Vasco T. Vasconcelos & Nobuko Yoshida (2016): *Behavioral Types in Programming Languages*. Foundations and Trends in Programming Languages 3(2-3), pp. 95–230, doi:10.1561/2500000031.

[5] Davide Ancona & Andrea Corradi (2014): *Sound and Complete Subtyping between Coinductive Types for Object-Oriented Languages*. In: ECOOP 2014, pp. 282–307, doi:10.1007/978-3-662-44202-9_12.

[6] Davide Ancona & Andrea Corradi (2016): *Semantic subtyping for imperative object-oriented languages*. In: OOPSLA 2016, pp. 568–587, doi:10.1145/2983990.2983992.

[7] Davide Ancona, Sophia Drossopoulou & Viviana Mascardi (2012): *Automatic Generation of Self-monitoring MASs from Multiparty Global Session Types in Jason*. In: Declarative Agent Languages and Technologies X - 10th International Workshop, DALT 2012, Valencia, Spain, June 4, 2012, Revised Selected Papers, pp. 76–95, doi:10.1007/978-3-642-37890-4_5.

[8] Davide Ancona, Angelo Ferrando & Viviana Mascardi (2016): *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, pp. 47–64, doi:10.1007/978-3-319-30734-3_6.

[9] Davide Ancona, Angelo Ferrando & Viviana Mascardi (2017): *Parametric Runtime Verification of Multiagent Systems*. In: Proceedings of the 16th Conference on Autonomous Agents and MultiAgent Systems, AAMAS 2017, São Paulo, Brazil, May 8-12, 2017, pp. 1457–1459, doi:10.5555/3091125.3091328.

[10] Davide Ancona, Angelo Ferrando & Viviana Mascardi (2020): *Can determinism and compositionality coexist in RML? (extende version)*. Available at https://arxiv.org/abs/2008.06453.

[11] Davide Ancona, Luca Franceschini, Angelo Ferrando & Viviana Mascardi (2019): *A Deterministic Event Calculus for Effective Runtime Verification*. In Alessandra Cherubini, Nicoletta Sabadini & Simone Tini, editors: Proceedings of the 20th Italian Conference on Theoretical Computer Science, ICTCS 2019, Como, Italy, September 9-11, 2019, CEUR Workshop Proceedings 2504, CEUR-WS.org, pp. 248–260. Available at http://ceur-ws.org/Vol-2504/paper28.pdf.

[12] Howard Barringer, Yliès Falcone, Klaus Havelund, Giles Reger & David E. Rydeheard (2012): *Quantified Event Automata: Towards Expressive and Efficient Runtime Monitors*. In: FM 2012: Formal Methods - 18th International Symposium, Paris, France, August 27-31, 2012. Proceedings, pp. 68–84, doi:10.1007/978-3-642-32759-9_9.

[13] Ezio Bartocci, Yliès Falcone, Adrian Francalanza & Giles Reger (2018): *Introduction to Runtime Verification*. In: Lectures on Runtime Verification - Introductory and Advanced Topics, pp. 1–33, doi:10.1007/978-3-319-75632-5_1.

[14] Andreas Bauer, Martin Leucker & Christian Schallhart (2007): *The Good, the Bad, and the Ugly, But How Ugly Is Ugly?* In Oleg Sokolsky & Serdar Taşıran, editors: Runtime Verification, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 126–138, doi:10.1007/978-3-540-77395-5_11.

[15] Andreas Bauer, Martin Leucker & Christian Schallhart (2011): *Runtime verification for LTL and TLTL*. ACM Transactions on Software Engineering and Methodology (TOSEM) 20(4), pp. 14:1–14:64, doi:10.1145/2000799.2000800.

[16] Benedikt Bollig, Normann Decker & Martin Leucker (2012): *Frequency Linear-time Temporal Logic*. In: *Sixth International Symposium on Theoretical Aspects of Software Engineering, TASE 2012, 4-6 July 2012, Beijing, China*, pp. 85–92, doi:10.1109/TASE.2012.43.

[17] G. Castagna, M. Dezani-Ciancaglini & L. Padovani (2012): *On Global Types and Multi-Party Session*. Logical Methods in Computer Science 8(1), doi:10.2168/LMCS-8(1:24)2012.

[18] Antonio Cau & Hussein Zedan (1997): *Refining Interval Temporal Logic Specifications*. In: *Transformation-Based Reactive Systems Development, 4th International AMAST Workshop on Real-Time Systems and Concurrent and Distributed Software, ARTS'97, Palma, Mallorca, Spain, May 21-23, 1997, Proceedings*, pp. 79–94, doi:10.1007/3-540-63010-4_6.

[19] Feng Chen & Grigore Rosu (2007): *Mop: an efficient and generic runtime verification framework*. In: *Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2007, October 21-25, 2007, Montreal, Quebec, Canada*, pp. 569–588, doi:10.1145/1297027.1297069.

[20] Feng Chen & Grigore Rosu (2009): *Parametric Trace Slicing and Monitoring*. In: *Tools and Algorithms for the Construction and Analysis of Systems, 15th International Conference, TACAS 2009, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009, York, UK, March 22-29, 2009. Proceedings*, pp. 246–261, doi:10.1007/978-3-642-00768-2_23.

[21] Christian Colombo, Gordon J. Pace & Gerardo Schneider (2008): *Dynamic Event-Based Runtime Monitoring of Real-Time and Contextual Properties*. In: *Formal Methods for Industrial Critical Systems, 13th International Workshop, FMICS 2008, L'Aquila, Italy, September 15-16, 2008, Revised Selected Papers*, pp. 135–149, doi:10.1007/978-3-642-03240-0_13.

[22] Christian Colombo, Gordon J. Pace & Gerardo Schneider (2009): *LARVA – Safer Monitoring of Real-Time Java Programs*. In: *SEFM 2009*, pp. 33–37, doi:10.1109/SEFM.2009.13.

[23] Bruno Courcelle (1983): *Fundamental Properties of Infinite Trees*. Theor. Comput. Sci. 25, pp. 95–169, doi:10.1016/0304-3975(83)90059-2.

[24] James C. Davis, Christy A. Coghlan, Francisco Servant & Dongyoon Lee (2018): *The impact of regular expression denial of service (ReDoS) in practice: an empirical study at the ecosystem scale*. In: *Proceedings of the 2018 ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2018, Lake Buena Vista, FL, USA, November 04-09, 2018*, pp. 246–256, doi:10.1145/3236024.3236027.

[25] Nelly Delgado, Ann Q. Gates & Steve Roach (2004): *A Taxonomy and Catalog of Runtime Software-Fault Monitoring Tools*. IEEE Trans. Software Eng. 30(12), pp. 859–872, doi:10.1109/TSE.2004.91.

[26] Yliès Falcone, Klaus Havelund & Giles Reger (2013): *A Tutorial on Runtime Verification*. In: *Engineering Dependable Software Systems*, pp. 141–175, doi:10.3233/978-1-61499-207-3-141.

[27] Yliès Falcone, Srdan Krstic, Giles Reger & Dmitriy Traytel (2018): *A Taxonomy for Classifying Runtime Verification Tools*. In: *Runtime Verification - 18th International Conference, RV 2018, Limassol, Cyprus, November 10-13, 2018, Proceedings*, pp. 241–262, doi:10.1007/978-3-030-03769-7_14.

[28] Luca Franceschini (March 2020): *RML: Runtime Monitoring Language*. Ph.D. thesis, DIBRIS - University of Genova. Available at `http://hdl.handle.net/11567/1001856`.

[29] A. Frisch, G. Castagna & V. Benzaken (2008): *Semantic subtyping: Dealing set-theoretically with function, union, intersection, and negation types*. J. ACM 55(4), doi:10.1145/1391289.1391293.

[30] Klaus Havelund & Allen Goldberg (2005): *Verify Your Runs*. In: *Verified Software: Theories, Tools, Experiments, First IFIP TC 2/WG 2.3 Conference, VSTTE 2005, Zurich, Switzerland, October 10-13, 2005, Revised Selected Papers and Discussions*, pp. 374–383, doi:10.1007/978-3-540-69149-5_40.

[31] Klaus Havelund, Giles Reger, Daniel Thoma & Eugen Zalinescu (2018): *Monitoring Events that Carry Data*. In: *Lectures on Runtime Verification - Introductory and Advanced Topics*, pp. 61–102, doi:10.1007/978-3-319-75632-5_3.

[32] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm & William G. Griswold (2001): *An Overview of AspectJ*. In: *ECOOP 2001 - Object-Oriented Programming, 15th European Conference, Budapest, Hungary, June 18-22, 2001, Proceedings*, pp. 327–353, doi:10.1007/3-540-45337-7_18.

[33] Dexter Kozen (1983): *Results on the Propositional mu-Calculus*. Theor. Comput. Sci. 27, pp. 333–354, doi:10.1016/0304-3975(82)90125-6.

[34] Kim Guldstrand Larsen (1990): *Proof Systems for Satisfiability in Hennessy-Milner Logic with Recursion*. Theor. Comput. Sci. 72(2&3), pp. 265–288, doi:10.1016/0304-3975(90)90038-J.

[35] Martin Leucker & Christian Schallhart (2009): *A brief account of runtime verification*. The Journal of Logic and Algebraic Programming 78(5), pp. 293–303, doi:10.1016/j.jlap.2008.08.004.

[36] Nancy A. Lynch & Mark R. Tuttle (1987): *Hierarchical Correctness Proofs for Distributed Algorithms*. In Fred B. Schneider, editor: *Proceedings of the Sixth Annual ACM Symposium on Principles of Distributed Computing, Vancouver, British Columbia, Canada, August 10-12, 1987*, ACM, pp. 137–151, doi:10.1145/41840.41852.

[37] RC Moore (2000): *Removing left recursion from context-free grammars*. NAACL 2000: *Proceedings of the 1st North American chapter of the Association for Computational Linguistics conference*. Available at `https://www.aclweb.org/anthology/A00-2033`.

[38] Amir Pnueli (1977): *The temporal logic of programs*. In: *18th Annual Symposium on Foundations of Computer Science, 1977*, IEEE, pp. 46–57, doi:10.1109/SFCS.1977.32.

[39] Amir Pnueli & Lenore D. Zuck (1993): *In and Out of Temporal Logic*. In: *Proceedings of the Eighth Annual Symposium on Logic in Computer Science (LICS '93), Montreal, Canada, June 19-23, 1993*, pp. 124–135, doi:10.1109/LICS.1993.287594.

[40] Giles Reger, Helena Cuenca Cruz & David E. Rydeheard (2015): *MarQ: Monitoring at Runtime with QEA*. In: *Tools and Algorithms for the Construction and Analysis of Systems - 21st International Conference, TACAS 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015. Proceedings*, pp. 596–610, doi:10.1007/978-3-662-46681-0_55.

[41] Oleg Sokolsky, Klaus Havelund & Insup Lee (2012): *Introduction to the special section on runtime verification*. STTT 14(3), pp. 243–247, doi:10.1007/s10009-011-0218-6.

[42] Prasanna Thati & Grigore Rosu (2005): *Monitoring Algorithms for Metric Temporal Logic Specifications*. Electr. Notes Theor. Comput. Sci. 113, pp. 145–162, doi:10.1016/j.entcs.2004.01.029.

[43] Frits W. Vaandrager (1991): *Determinism - (Event Structure Isomorphism = Step Sequence Equivalence)*. Theor. Comput. Sci. 79(2), pp. 275–294, doi:10.1016/0304-3975(91)90333-W.

[44] Glynn Winskel (1986): *Event Structures*. In Wilfried Brauer, Wolfgang Reisig & Grzegorz Rozenberg, editors: *Petri Nets: Central Models and Their Properties, Advances in Petri Nets 1986, Part II, Proceedings of an Advanced Course, Bad Honnef, Germany, 8-19 September 1986, Lecture Notes in Computer Science* 255, Springer, pp. 325–392, doi:10.1007/3-540-17906-2_31.