A deterministic event calculus for effective runtime verification

Davide Ancona[1], Luca Franceschini[1],
Angelo Ferrando[2], and Viviana Mascardi[1]

[1] DIBRIS, Università di Genova, Italy
{davide.ancona,viviana.mascardi}@unige.it,
luca.franceschini@dibris.unige.it
[2] University of Liverpool, UK angelo.ferrando@liverpool.ac.uk

Abstract. Runtime verification (RV) is an effective technique for dynamically monitoring, even after deployment, properties that could be hardly verified statically. To this aim, specification formalims for RV have to reconcile expressive power and monitoring efficiency.
We present an event calculus which provides a basis for the semantics and the implementation of RML, a domain specific language (DSL) for RV. The semantics of the calculus is based on a deterministic reduction strategy which allows concise specifications of non context-free properties, and their efficient verification at runtime.

Keywords: Runtime verification · event calculi · specification languages · effective monitoring.

1 Introduction

Runtime Verification (RV) [10,8] is a technique concerned with dynamic monitoring of the traces of events generated by a system under scrutiny (SUS); this is typically achieved through a monitor synthesized from a specification which is written in either a DSL or a programming language, and which defines the expected correct behavior of the SUS.

Roughly, RV amounts to deal with a word problem, since the aim of the monitor is to check that a trace of events (the word) belongs to the set of valid traces defined by the specification; however, this is an oversimplification for several reasons. RV is also concerned with the problem, typically addressed by *code instrumentation*, of generating the events that have to be observed and consumed by the monitor. For what concerns the monitoring activity, monitors can perform either *offline*, after the execution of the SUS has produced a necessarily finite trace, or *online*, during the execution of a possibly non-terminating SUS, hence the word to be checked can be either finite or infinite. The *verdict* of the monitor is usually modeled by a many-valued logic to take into account inconclusive responses which may occur especially for online monitoring. Finally, if monitoring occurs after deployment, then the monitor is also expected to send a feedback to the SUS to allow for error recovery.

RV can be successfully integrated [1] with other verification techniques because of its distinguishing features: as static verification, it is based on formal specifications, but allows checking of properties that can be hardly verified statically; as software testing, it scales well to real complex systems, and is not exhaustive, but is better for dealing with control-oriented properties and non deterministic behavior, and may offer opportunities for runtime contract enforcement, fault protection and automatic program repair.

In order to be effective, RV DSLs have to reconcile expressive power and monitoring efficiency; for this reason, they are usually more suitable to monitor control-oriented properties, while they exhibit limitations in dealing with data-oriented properties which often require more expressive power; several RV DSLs proposed in literature are based on regular expressions or temporal logics which are not able to express context-free (CF) properties, some support CF grammars, but, anyway, fail to define properties beyond that expressive power; for instance, verifying the correctness of a simple but widely used data type as FIFO queues requires non-CF properties.

RML is a rewriting-based [2] and system agnostic RV DSL which decouples monitoring from instrumentation by allowing users[3] to write specifications and to synthesize monitors from them, independently of the SUS and its instrumentation. The expressive power of RML goes beyond CF grammars, and the language has been successfully employed to verify complex properties in multi-agent systems [6,7], and for Node.js and IoT applications [3,9].

This paper is focused on the problem of non-deterministic specifications that typically has to be faced when designing and implementing RV DSLs: non deterministic monitors are more difficult to be fully implemented and may introduce serious performance penalties that make them unusable in practice; furthermore, detecting whether a specification is not deterministic can be difficult, if not undecidable, and the problem of automatically transforming a non deterministic specification into an equivalent deterministic one is even harder.

In RML specifications can be deterministic by construction if the semantics is based on a deterministic reduction strategy; by exploring this possibility, we have discovered that, besides supporting a simpler and more efficient implementation, a deterministic semantics still allows RML users to express certain non-CF data-oriented properties in a simple way and to monitor them efficiently.

Section 2 briefly introduces RML and examples of specifications of data-oriented properties which benefit from the adoption of a deterministic semantics; the semantics of the language is formalized by an event calculus in Section 3, where its basic properties are proved. Section 4 draws conclusions, discusses limitations and outlines opportunities for future work.

[3] The implementation of RML is available at `https://github.com/LucaFranceschini/RML`.

2 RML with deterministic reduction strategy

This section briefly introduces RML, a runtime monitoring language with motivating examples; the presentation is deliberately informal, the semantics of the language is defined in Section 3.

2.1 Events and event types

RML is based on a general model where events are object literals.Such a model can be instantiated in more specific ones, where events may include several property keys, with their types and intended meaning. Throughout the whole paper, we will consider a simple model where events can have the property keys `event`, `name`, `args`, and `res`; `event` is associated with a string specifying only two possible kinds of event: `'func_pre'` or `'func_post'` for the events 'function has been called' and 'function has returned', respectively; `name` specifies the name of the function, `args` the list of its arguments, and `res` (only expected for `'func_post'` events) its result.

For instance, for a certain data value `val`, the following events

```
{event:'func_pre',name:'enqueue',args:[val]}
{event:'func_post',name:'dequeue',args:[],res:val}
```

denote 'function `enqueue` has been called with argument `val`', and 'function `dequeue` has returned value `val`, after being called with no arguments', respectively.

Event types define possibly infinite sets of events and coincide with what is often referred as symbolic events in the context of RV [5]. In RML, event types are terms built on top of (possibly overloaded) symbols with different arities, and subterms representing data values of primitive, array, or object type; event types are defined by pattern matching as in the following example:

```
enq(val) matches {event:'func_pre',name:'enqueue',args:[val]};
deq(val) matches  {event:'func_post',name:'dequeue',res:val};
enq matches enq(_);
deq matches deq(_);
```

Listing 1.1. Definition of event types.

According to the definition, `enq(val)` matches all calls to function `enqueue` with argument `val`, `deq(val)` all returns from function `dequeue` with result `val`, and `enq` and `deq` match all events matching `enq(val)` and `deq(val)`, respectively, for any[4] value `val`.

A match succeeds if the event has all specified properties with the expected values, and possibly other properties; RML supports two basic predefined event types denoted by the keywords `none` and `any`, which match no event and all events, respectively. Other features that allow more flexible definitions of event types, including negation, are not presented here, for sake of brevity.

[4] Wild-cards are used for unspecified property values.

2.2 Basic operators

In RML specifications denote sets of event traces (i.e., sequence of events), obtained by combining event types with the following basic binary operators, listed in decreasing order of precedence: concatenation (juxtaposition), intersection ($\wedge$), union ($\vee$), and shuffle ($|$); recursion is supported, and the keyword `empty` denotes the singleton set containing the empty trace.

The formal semantics of the basic operators is defined in the calculus presented in Section 3; to the aim of this section, it is important to observe that, while intersection always requires both operands to consume the current event, for the others only one operand at time is needed, and there are cases where both could be selected; hence, for intersection there exists a unique reduction strategy, which is also deterministic, but for the remaining operators one can opt for either a non deterministic (ND) or a deterministic strategy.

Let us consider for instance the following specification which uses the shuffle operator:

```
enq | (enq deq)  // shuffle enq with enq concatenated to deq
```

This specification requires the first event of the trace to match event type `enq`; with the ND strategy, the event can be consumed by either the right or left operand of the shuffle; accordingly, the specification reduces to either `empty | (enq deq)` or `enq | deq`. This strategy corresponds to the conventional semantics of the shuffle operator in formal languages: traces $en\,en'\,de$ and $en'\,en\,de$ and $en'\,de\,en$ are accepted, where en and en' are events matching the left-most and right-most event type `enq`, respectively, and de matches `deq`. Conversely, only $en\,en'\,de$ is accepted with the left-to-right (LR) deterministic strategy.

Similar considerations apply also for union and concatenation. In the specification below, both operands of the union operator accept a first event matching `enq` with the ND strategy, thus the specification reduces to either `empty` or `deq`, and both traces en and $en'\,de$ are accepted, while only en is accepted with the LR strategy.

```
enq ∨ (enq deq) // union of enq with enq concatenated to deq
```

In the following specification both `(enq ∨ empty)` and `(enq deq)` expect a first event matching `enq`; because `(enq ∨ empty)` accepts the empty trace, the specification reduces to either `empty (enq deq)` or `deq` with the ND strategy, and both $en\,en'\,de$ and $en'\,de$ are accepted, while only $en\,en'\,de$ is accepted with the LR strategy.

```
(enq ∨ empty) (enq deq) // concatenate enq ∨ empty to enq deq
```

2.3 Derived operators

Besides the previously introduced operators, several derived operators can be used for conciseness. Standard postfix operators are borrowed from regular expressions: for any expression exp, `(exp)?` is equivalent to `empty ∨ (exp)`, while `(exp)*` and `(exp)+` correspond to the following specifications:

```
Star = empty ∨ (exp) Star // (exp)*
Plus = (exp) Star // (exp)+
```

The empty set of traces can be easily represented by `none`, while the constant `all` denotes the universe of all traces, which can be expressed by `any*`.

The filter operator is useful when only some kinds of events are relevant for verifying a certain property: in ev_ty≫exp$_1$:exp$_2$, expression exp$_1$ consumes events matching ev_ty, while exp$_2$ the others; ev_ty≫exp is an abbreviated version corresponding to ev_ty≫exp:`all` (that is, only events matching ev_ty must be checked). The filter operator is expressible in terms of the intersection and shuffle operators: (ev_ty*∧exp$_1$)|(not_ev_ty*∧exp$_2$), where not_ev_ty is the negation of the event type ev_ty.

2.4 Parametric specifications

A specification is called parametric if it expresses a property which depends on the data values carried by the events; linguistic support for parametric specifications is of paramount importance to guarantee that the expressive power of the specification formalism is high enough to make runtime verification effective.

Let us consider randomized queues, where enqueued elements can be dequeued in any order, and let us assume that queues are initially empty; to be correctly specified, a parametric specification is required.

For instance, `RQ = (enq ( deq | RQ ))?` does not work properly, because it accepts incorrect traces as $en_1\ de_2$, where en_1 and de_2 match `enq(1)` and `deq(2)`, respectively,

RML supports parametric specifications by allowing users to declare variables which hold data values and are dynamically bound when events are matched.

```
RQ = {let val; enq(val) ( deq(val) | RQ)}?;
```

Listing 1.2. Specification of randomized queues.

The specification is parametric in the enqueued and dequeued values, thanks to the declaration of the variable `val`, whose scope is delimited by the curly braces; a value is dynamically associated with `val` when an event successfully matches `enq(val)`, so that the subsequent event type `deq(val)` can only match events corresponding to dequeuing the same value. Since the specification is recursively defined, the declaration of `val` can be arbitrarily nested; enqueuing of different values can be correctly managed because at each level the new declaration hides the outer ones.

With the shuffle operator it is possible to express concisely the random behavior of the `dequeue` operation; in this case the reduction strategy does not affect the semantics, because event types `enq(val)` and `deq(val)` are disjoint and queues are randomized, hence elements can be dequeued in any order, and it does not matter whether the corresponding event is consumed by the left or right operand of the shuffle.

For sake of simplicity, the specification above, and those that follow, require that traces can end only when the queue is empty; in this case the condition can

be relaxed by adding `?` at the end of `deq(val)`; similar comments apply to the other examples.

2.5 Examples with the LR reduction strategy

The following examples show how the LR reduction strategy can help specifying non trivial properties concisely and efficiently.

Randomized queues with no repetitions: we first consider the variation of randomized queues with no repetitions.

```
RQNR = {let val; enq(val) (enq(val)* deq(val) | RQNR)}?;
```

Listing 1.3. Specification of randomized queues with no repetitions.

The specification is obtained by slightly changing the example in Listing 1.2: `enq(val)*` is added before `deq(val)`, to ensure that after a value `val` is enqueued, subsequent additions of the same element will not change the queue before the element is dequeued. In this case, the ND and the LR strategy yield different semantics.

Let us consider the incorrect trace $en_1\ en_1\ de_1\ de_1$, where en_1 and de_1 match `enq(1)` and `deq(1)`, respectively; the first occurrence of en_1 can only be consumed by the left-most occurrence of `enq(val)`; accordingly, the specification reduces into `enq(1)* deq(1) | RQNR`; the second occurrence of en_1 can be consumed by either the left or the right operand of the shuffle, but the LR strategy forces the reduction of `enq(val)* deq(val)` into itself (by definition of the star and concatenation operators). Then, the first occurrence of de_1 can be consumed by `enq(1)* deq(1)` (but not by `RQNR`), and the specification reduces to `empty | RQNR`; hence, the second occurrence of de_1 cannot be accepted, and, the whole trace is correctly rejected. Conversely, with the ND strategy the second occurrence of en_1 can be consumed by `RQNR` yielding the specification `enq(1)* deq(1) | enq(1)* deq(1) | RQNR` which can accept both occurrences of de_1, and, thus, fails to reject the whole trace.

FIFO queues: the specification in Listing 1.4 defines the correct behavior of standard FIFO queues.

```
Q = {let val; enq(val) ((deq | Q) ∧ (deq≫deq(val) all))}?;
```

Listing 1.4. Specification of FIFO queues.

The specification is obtained from `RQ` in Listing 1.2 by adding, through the intersection operator, the constraint specified by `deq≫(deq(val) all)`, requiring that, in all traces, the first event matching `deq` must actually match `deq(val)`; indeed, this addition allows refinement of randomized queues into FIFO queues. In this case `(deq | Q)` works equally well as `(deq(val) | Q)`, since the fact that a call to `dequeue` must return the value `val` is ensured by the right operand of the intersection operator.

Also in this case, the specification is correct only if we adopt the LR strategy; indeed, with the ND strategy, for any trace t in `Q`, `(deq | Q)` contains all traces obtained by inserting in t an event matching `deq` at arbitrary position; from this we can derive that, for instance, the incorrect trace $en_1\ en_1\ en_2\ de_1\ de_2\ de_1$ can be accepted with the ND strategy, if en_i and de_i match `enq(`i`)` and `deq(`i`)`, respectively. This counter-example also works when `(deq | Q)` is replaced with `(deq(val) | Q)`.

FIFO queues with no repetitions: similarly as done for randomized queues, the specification of FIFO queues with no repetitions in Listing 1.5 is obtained from the specification in Listing 1.4 by adding `enq(val)*` before `deq` in the left operand of the shuffle operator.

```
QNR={let val; enq(val) ((enq(val)* deq | QNR)
     ∧ (deq≫deq(val) all))}?;
```

Listing 1.5. Specification of FIFO queues with no repetitions.

Analogously as done for the specification in Listing 1.4, one can show that the specification is correct only with the LR strategy.

3 Event calculus

In this section we define the formal semantics of RML through an event calculus which is also at the basis of the language implementation: RML specifications are translated into the calculus, and the corresponding reduction rules, defining a labeled transition system, are implemented in SWI-Prolog[5] to monitor event traces.

v	$::= x \mid \kappa$	(data value)
θ	$::= \tau \mid \tau(v_1, \ldots, v_n)$	(event type)
t	$::= \epsilon$	(empty trace)
	$\mid \theta : t$	(prefix)
	$\mid t_1 \cdot t_2$	(concatenation)
	$\mid t_1 \wedge t_2$	(intersection)
	$\mid t_1 \vee t_2$	(union)
	$\mid t_1 \mid t_2$	(shuffle)
	$\mid \{x;\ t\}$	(parametric expression)

Fig. 1. Syntax of trace calculus.

Syntax. As in the RML examples, event types are built on top of names τ and data value variables x and literals κ. The calculus does not cover event type definitions, which are assumed to be provided separately; we only require that the set of event types is not empty, and that definitions of event types are closed

[5] `http://www.swi-prolog.org/`

w.r.t. negation: for any event type θ, its negation $\bar{\theta}$ is definable. This assumption is needed to ensure that operators in Section 2.3 can be actually derived in the calculus.

For conciseness ϵ is used for `empty`; the two constants 1 and 0 denote the universe and the empty set of all event traces, respectively. They are introduced for convenience, and they do not belong to the syntax of the calculus, since they are both derivable, by virtue of our hypotheses on event type definitions: 1 is equivalent to the term defined by $t = \epsilon \vee (\theta : t) \vee (\bar{\theta} : t)$, while 0 corresponds to $(\theta : \epsilon) \wedge (\bar{\theta} : \epsilon)$, for any event type θ.

In the calculus, RML concatenation is expressed with the prefix and the concatenation operators, and the empty trace; for instance, `(enq ∨ empty) (enq deq)` is translated into the term `((enq:`ϵ`)∨`ϵ`)·(enq:deq:`ϵ`)`. The remaining basic binary operators are supported with the same syntax of RML; a parametric expression $\{x;\ t\}$ corresponds to the RML expression `{let` x`; `t`}`. As shown in Section 2, all other operators can be derived from the basic ones.

The terms of the calculus are allowed to be regular (a.k.a. rational) [4], to support recursion: they correspond to trees with possibly infinite depth, but finite set of subtrees; equivalently, they are those terms that can always be defined as the unique solutions of a finite set of syntactic equations. For instance, the equation $t =$ `enq:`t defines a unique tree, whose depth is infinite and whose set of subtrees only contains `enq` and t itself.

Semantics. The semantics of the calculus depends on four predicates, inductively defined by the inference rules in Figure 2. Events e range over a fixed universe of events $\mathcal{E}$. The predicate $t_1 \xrightarrow{e} t_2; \sigma$ 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 negation of the predicate, denoted by $t \not\xrightarrow{e}$, is derivable iff there are no reduction steps for event e starting from expression t; the predicate is needed to enforce the LR reduction strategy.

Substitutions are finite partial maps from variables to data values which are produced by successful matches of event types; 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 expression t is denoted by σt, and defined by coinductionon t; in particular,

$$\sigma x = \sigma(x) \text{ if } x \in dom(\sigma),\ \sigma(x) = x \text{ otherwise}$$
$$\sigma\{x;\ t\} = \{x;\ \sigma_{\setminus x} t\}$$

while σt is the homomorphic extension for the remaining cases.

Since the calculus does not cover event type definitions, the semantics of event types is provided by the external 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

$$\text{(pre)}\ \frac{}{\theta : t \xrightarrow{e} t;\sigma}\ \sigma{=}match(e,\theta) \qquad \text{(or-l)}\ \frac{t_1 \xrightarrow{e} t_1';\sigma}{t_1 \vee t_2 \xrightarrow{e} t_1';\sigma} \qquad \text{(or-r)}\ \frac{t_1 \overset{e}{\not\to} \quad t_2 \xrightarrow{e} t_2';\sigma}{t_1 \vee t_2 \xrightarrow{e} t_2';\sigma}$$

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

$$\text{(shuffle-r)}\ \frac{t_1 \overset{e}{\not\to} \quad t_2 \xrightarrow{e} t_2';\sigma}{t_1 \mid t_2 \xrightarrow{e} t_1 \mid t_2';\sigma} \qquad \text{(cat-l)}\ \frac{t_1 \xrightarrow{e} t_1';\sigma}{t_1 \cdot t_2 \xrightarrow{e} t_1' \cdot t_2;\sigma}$$

$$\text{(cat-r)}\ \frac{t_1 \overset{e}{\not\to} \quad E(t_1) \quad t_2 \xrightarrow{e} t_2';\sigma}{t_1 \cdot t_2 \xrightarrow{e} t_2';\sigma} \qquad \text{(par-t)}\ \frac{t \xrightarrow{e} t';\sigma}{\{x;\ t\} \xrightarrow{e} \sigma_{|x} t';\sigma_{\setminus\{x\}}}\ x \in dom(\sigma)$$

$$\text{(par-f)}\ \frac{t \xrightarrow{e} t';\sigma}{\{x;\ t\} \xrightarrow{e} \{x;\ t'\};\sigma}\ x \notin dom(\sigma) \qquad \text{(n-}\epsilon\text{)}\ \frac{}{\epsilon \overset{e}{\not\to}} \qquad \text{(n-prefix)}\ \frac{}{\theta : t \overset{e}{\not\to}}\ match(e,\theta)\ \text{fails}$$

$$\text{(n-or)}\ \frac{t_1 \overset{e}{\not\to} \quad t_2 \overset{e}{\not\to}}{t_1 \vee t_2 \overset{e}{\not\to}} \qquad \text{(n-and-l)}\ \frac{t_1 \overset{e}{\not\to}}{t_1 \wedge t_2 \overset{e}{\not\to}} \qquad \text{(n-and-r)}\ \frac{t_2 \overset{e}{\not\to}}{t_1 \wedge t_2 \overset{e}{\not\to}}$$

$$\text{(n-and)}\ \frac{t_1 \xrightarrow{e} t_1';\sigma_1 \quad t_2 \xrightarrow{e} t_2';\sigma_2}{t_1 \wedge t_2 \overset{e}{\not\to}}\ \sigma_1 \cup \sigma_2\ \text{fails} \qquad \text{(n-shuffle)}\ \frac{t_1 \overset{e}{\not\to} \quad t_2 \overset{e}{\not\to}}{t_1 \mid t_2 \overset{e}{\not\to}}$$

$$\text{(n-cat-l)}\ \frac{t_1 \overset{e}{\not\to} \quad NE(t_1)}{t_1 \cdot t_2 \overset{e}{\not\to}} \qquad \text{(n-cat-r)}\ \frac{t_1 \overset{e}{\not\to} \quad t_2 \overset{e}{\not\to}}{t_1 \cdot t_2 \overset{e}{\not\to}} \qquad \text{(n-par)}\ \frac{t \overset{e}{\not\to}}{\{x;\ t\} \overset{e}{\not\to}} \qquad \text{(e-}\epsilon\text{)}\ \frac{}{E(\epsilon)}$$

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

$$\text{(e-par)}\ \frac{E(t)}{E(\{x;\ t\})} \qquad \text{(ne-pre)}\ \frac{}{NE(\theta : t)} \qquad \text{(ne-or)}\ \frac{NE(t_1) \quad NE(t_1)}{NE(t_1 \vee t_2)}$$

$$\text{(ne-all)}\ \frac{\exists\, i \in \{1,2\}\ NE(t_i)}{NE(t_1\ op\ t_2)}\ op \in \{|,\cdot,\wedge\} \qquad \text{(ne-par)}\ \frac{NE(t)}{NE(\{x;\ t\})}$$

Fig. 2. Small-step operational semantics for the event calculus.

substitution σ for which e matches $\sigma\theta$. The substitution is expected to be the most general one, that is, its domain coincides with the set of variables in θ.

As an example of how *match* could be derived from the RML event type definitions, if $e =$`{event:'func_pre',name:'enqueue',args:[42]}`, then the following specification

```
enq(val) matches {event:'func_pre',name:'enqueue',args:[val]};
```

determines the function *match* s.t. $match(e, enq(v)) = \sigma$, and $match(e, enq(3))$ fails, if $dom(\sigma) = \{v\}$, $\sigma(v) = 42$.

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 fails otherwise.

The predicate E is needed in rule (cat-r): 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 $t_1 \stackrel{e}{\not\to}$ forcing the LR reduction strategy), and the empty trace is accepted by t_1 (premise $E(t_1)$).

Rule (par-t) can be applied when variable x is in the domain of the substitution σ generated during the reduction step from t to t': σ is applied to t', 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 removal is performed.

Since monitors can be on line, and can check non terminating programs, the semantics of a specification may include also infinite traces; an event trace $\bar{e} \in \mathcal{E}^* \cup \mathcal{E}^\omega$ is a possibly infinite sequence of events; the empty trace is denoted by ϵ, while $e\bar{e}$ denotes the trace consisting of the first event e, followed by the rest of the trace $\bar{e}$. The semantics $[\![t]\!]$ of a specification t is the set of traces coinductively defined as follows:

– $\epsilon \in [\![t]\!]$ iff $E(t)$ is derivable;
– $e\bar{e} \in [\![t]\!]$ iff $t \stackrel{e}{\to} t'; \sigma$ is derivable, and $\bar{e} \in [\![\sigma t']\!]$.

3.1 Examples of reductions

Before providing some examples of reductions, we show how an RML specification compiles down to the event calculus; in particular, we focus on `RQNR` defined in Listing 1.3, which is translated into the rational term t_{rqnr} defined by the following equations:

$$t_{\text{rqnr}} = \{v;\ enq(v) : ((s \cdot (deq(v) : \epsilon)) \mid t_{\text{rqnr}})\} \vee \epsilon \qquad s = \epsilon \vee (enq(v) : s)$$

Let us assume that function *match* corresponds to the definitions of the event types provided in Listing 1.1, and that en_i and de_i denote any event matching $enq(i)$ and $deq(i)$, respectively, for any integer value i.

The following reduction steps can be derived: $t_{\text{rqnr}} \xrightarrow{en_1} t_1 \xrightarrow{en_1} t_2 \xrightarrow{en_2} t_3 \xrightarrow{de_1} t_4 \xrightarrow{de_2} t_5$ where

$$t_1 = t_2 = (s_1 \cdot (deq(1) : \epsilon)) \mid t_{\text{rqnr}} \qquad t_4 = \epsilon \mid ((s_2 \cdot (deq(2) : \epsilon)) \mid t_{\text{rqnr}})$$
$$t_3 = (s_1 \cdot (deq(1) : \epsilon)) \mid ((s_2 \cdot (deq(2) : \epsilon)) \mid t_{\text{rqnr}}) \qquad t_5 = \epsilon \mid (\epsilon \mid t_{\text{rqnr}})$$

and s_i denotes the term obtained by applying the substitution $\{v \mapsto i\}$ to s. By rules (e-al), (e-or-r) and (e-ϵ), $E(t_5)$ is derivable, therefore we can deduce that $en_1\, en_1\, en_2\, de_1\, de_2 \in [\![t_{\text{rqnr}}]\!]$, as expected.

3.2 Identity and optimizations

In order to keep the time and space complexity of the verification procedure linear, it is crucial to shrink the calculus term during the reduction whenever possible. This can be done by exploiting some laws of the calculus which can be proved by virtue of the operational semantics:

$$[\![\epsilon \mid t]\!] = [\![t \mid \epsilon]\!] = [\![t]\!] \qquad [\![1 \wedge t]\!] = [\![t \wedge 1]\!] = [\![t]\!] \qquad [\![\theta \gg 1]\!] = [\![1]\!]$$

Let us consider Listing 1.4, formalizing the correct behavior of standard FIFO queues. Such RML specification can be translated to the following rational term in the event calculus:

$$t_q = \epsilon \vee \{val; enq(val) : (((deq : \epsilon) \mid t_q) \wedge (deq \gg deq(val) : 1\}$$

After enqueuing and dequeuing an element e, the following term is obtained:

$$t_q \xrightarrow{enq(e)} ((deq : \epsilon) \mid t_q) \wedge (deq \gg deq(e) : 1) \xrightarrow{deq(e)} (\epsilon \mid t_q) \wedge (deq \gg 1)$$

By identity of the shuffle operator and idempotence of 1 w.r.t. the filter, the resulting term can be simplified to $t_q \wedge 1$. Finally, since 1 is the identity element of intersection, the term is equivalent to the initial one, t_q.

The laws presented in this section are crucial to ensure good monitoring performance. The implementation tries to apply them after each step, to keep the size of the term as low as possible. The computational complexity of a single reduction step is a linear function of the term size. This can be understood by looking at the inference system inductively defining the small-step operational semantics. The complexity of the verification algorithm for the queue specifications given in this work, for instance, benefits from the optimizations driven by the laws. At any point in the reduction sequence, the time complexity of a single step (and the space complexity of the term) goes down from $O(n)$ to $O(s)$, with n and s being the *total* number of elements enqueued so far and the *current* size of the queue, respectively. Note that, in the long run, s could be much lower than n, and more importantly, only the former can decrease during execution. Without the ability to shrink the term when possible, the monitor would, at some point, run out of memory.

3.3 Formal results

In this section we prove that the semantics of the calculus is deterministic; in terms of the previous example, this means that there are no other reduction steps starting from t_{rqnr} for the trace $en_1 en_1 en_2 de_1 de_2$; furthermore, for each reduction there exists a unique derivation.

Lemma 1. *For all t, $E(t)$ is derivable if and only if $NE(t)$ is not derivable.*

Proof. (Sketch) The two implications can be proved simultaneously by induction on the derivations of $E(t)$ and $NE(t)$, and by case analysis on the shape of t.

Lemma 2. *For all t_1 and e, there exist t_2 and σ s.t. $t_1 \xrightarrow{e} t_2; \sigma$ is derivable if and only if $t_1 \overset{e}{\not\to}$ is not derivable.*

Proof. (Sketch) The proof depends on Lemma 1; except for this detail, the proof proceeds in a similar way as for lemma 1.

Theorem 1. *For all t, e, t_1, t_2, σ_1 and σ_2, if $t \xrightarrow{e} t_1; \sigma_1$ and $t \xrightarrow{e} t_2; \sigma_2$ are derivable, then $t_1 = t_2$ and $\sigma_1 = \sigma_2$.*

Proof. The proof proceeds by induction on the sum of the depths of the two derivations for $t \xrightarrow{e} t_1; \sigma_1$ and $t \xrightarrow{e} t_2; \sigma_2$, and by case analysis on the shape of t.

- if $t = \epsilon$, then the claim holds vacuously, since there cannot be reduction steps.
- if $t = \theta : t'$, then the derivations can only be obtained by applying rule (prefix), and the claim can be derived directly from the assumption that *match* is a partial function, as stated at the beginning of this section.
- if $t = \{x;\ t'\}$, then the derivations can only be obtained by first applying either rule (par-t) or (par-f); since the two rules share the same hypothesis, we have two derivations for $t' \xrightarrow{e} t''; \sigma'$ and $t' \xrightarrow{e} t'''; \sigma''$ where the sum of their depths is strictly less than the sum of the depths of the derivations for $t \xrightarrow{e} t_1; \sigma_1$ and $t \xrightarrow{e} t_2; \sigma_2$. Hence, by induction we deduce that $t'' = t'''$ and $\sigma' = \sigma''$; from the equality $\sigma' = \sigma''$ and the two side conditions of rules (par-t) or (par-f), we deduce that both derivations have been built by applying the same rule, hence the claim can be derived.
- the proofs for all remaining shapes are similar: they all exploit the fact that by Lemmas 1 and 2 there always exists at most one applicable rule for fixed t and e, and then conclude the claim by induction.

4 Conclusion

We have presented an event calculus which provides a basis for the semantics and the implementation of RML, a domain specific language for RV; the semantics is based on a deterministic reduction strategy, therefore RML specifications are deterministic by construction. We have shown that, besides supporting a simpler and more efficient implementation of RML, a deterministic semantics allows RML users to express certain non-CF data-oriented properties in a simple way and to monitor them efficiently. Such examples are all based on the deterministic semantics of the shuffle operator, but there are others, that have been omitted for space reasons, which exploit the deterministic semantics of concatenation as well, while for union we conjecture that the deterministic version can be derived from the ND one. Even though it has its advantages, considering a deterministic reduction strategy does not come at no costs, because the semantics of union, concatenation and shuffle is no longer the conventional one, as in the case of the ND semantics; for instance, while we have shown that there are some laws that still hold and can be exploited for optimization purposes, others, as commutativity of union and shuffle, no longer hold for the deterministic semantics.

We reserve for future work a deeper study of the properties of the deterministic version of these operators, the relationship between the expressive power of the calculus with the ND and the deterministic semantics (respectively), and the monitorability [11] of RML specifications for finite prefixes of traces.

References

1. Wolfgang Ahrendt, Jesús Mauricio Chimento, Gordon J. Pace, and Gerardo Schneider. Verifying data- and control-oriented properties combining static and runtime

verification: theory and tools. *Formal Methods in System Design*, 51(1):200–265, 2017.

2. Davide Ancona, Angelo Ferrando, and Viviana Mascardi. 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*, pages 1457–1459, 2017.
3. Davide Ancona, Luca Franceschini, Giorgio Delzanno, Maurizio Leotta, Marina Ribaudo, and Filippo Ricca. Towards runtime monitoring of node.js and its application to the internet of things. In *Proceedings First Workshop on Architectures, Languages and Paradigms for IoT, ALP4IoT@iFM 2017, Turin, Italy, September 18, 2017.*, pages 27–42, 2017.
4. Bruno Courcelle. Fundamental properties of infinite trees. *Theor. Comput. Sci.*, 25:95–169, 1983.
5. Yliès Falcone, Klaus Havelund, and Giles Reger. A tutorial on runtime verification. In *Engineering Dependable Software Systems*, pages 141–175. 2013.
6. Angelo Ferrando, Davide Ancona, and Viviana Mascardi. Decentralizing MAS monitoring with decamon. In *Proceedings of the 16th Conference on Autonomous Agents and MultiAgent Systems, AAMAS 2017, São Paulo, Brazil, May 8-12, 2017*, pages 239–248, 2017.
7. Angelo Ferrando, Louise A. Dennis, Davide Ancona, Michael Fisher, and Viviana Mascardi. Recognising assumption violations in autonomous systems verification. In *Proceedings of the 17th International Conference on Autonomous Agents and MultiAgent Systems, AAMAS 2018, Stockholm, Sweden, July 10-15, 2018*, pages 1933–1935, 2018.
8. Adrian Francalanza, Jorge A. Pérez, and César Sánchez. Runtime verification for decentralised and distributed systems. In *Lectures on Runtime Verification - Introductory and Advanced Topics*, pages 176–210. 2018.
9. Maurizio Leotta, Diego Clerissi, Dario Olianas, Filippo Ricca, Davide Ancona, Giorgio Delzanno, Luca Franceschini, and Marina Ribaudo. An acceptance testing approach for internet of things systems. *IET Software*, 12(5):430–436, 2018.
10. Martin Leucker and Christian Schallhart. A brief account of runtime verification. *The Journal of Logic and Algebraic Programming*, 78(5):293–303, 2009.
11. Amir Pnueli and Aleksandr Zaks. PSL model checking and run-time verification via testers. In *FM 2006: Formal Methods, 14th International Symposium on Formal Methods, Hamilton, Canada, August 21-27, 2006, Proceedings*, pages 573–586, 2006.