A SWI-Prolog based implementation of RML

Davide Ancona
DIBRIS, University of Genova, Italy
Davide.Ancona@unige.it

Luca Franceschini
DIBRIS, University of Genova, Italy
Luca.Franceschini@dibris.unige.it

Viviana Mascardi
DIBRIS, University of Genova, Italy
Viviana.Mascardi@unige.it

Angelo Ferrando
University of Liverpool, UK
angelo.ferrando@liverpool.ac.uk

RML

Runtime verification (RV) [3] consists in dynamically checking event traces generated by single runs of a system against a formal specification; such a technique bridges the gap between testing and formal verification: it scales well, as testing, is based on a specification language, as formal verification, but supports another feature which cannot be provided by testing and formal verification: it can monitor and properly handle misbehavior after deployment. Dynamic checking is performed by one or more monitors automatically generated from specifications; the monitors consume the events of the produced trace and emit verdicts on the behavior of the system; the events to be monitored are usually generated with code instrumentation of the system under scrutiny.

RML [4] is an expressive rewriting-based domain specific language (DSL) for RV, designed to be system agnostic to favor reusability of specifications and interoperability of generated monitors; to this aim, it is independent from instrumentation, and adopts a general model where events are uniformly represented as inductive objects whose properties can be associated with primitive values (number, booleans and strings), arrays of values, or other objects. For instance, the object `{event:'fun_ret', name:'req', args:[{headers:{'cont-len':13}}], resId:24}` may represent the event notifying that function `req` has returned an object identified by `24` when called with object `{headers:{'cont-len':13}}` as its only argument. For better interoperability, the data-interchange format JSON [1] is exploited for events.

Event types are an abstraction mechanism that supports flexible adaptability and reuse. An event type is a predicate which defines a possibly infinite set of events; for instance, if the event type `open/1` is defined by `open(fd) matches {event:'fun_ret', name:'open', res:fd}` then the event type pattern[1] `open(`*fd*`)` matches all events of shape `{event:'fun_ret', name:'open', res:`*fd*`, ...}`, where the ellipsis means that more properties are allowed.

An RML specification is built on top of event type patterns and denotes a set of finite and infinite event traces, defined by mutually recursive combinations of sets with the basic binary operators of concatenation (juxtaposition), intersection ($\wedge$), union ($\vee$), shuffle ($|$) and the unary postfix operator !, to compute the prefix closure of a set of traces. RML is more expressive than context-free grammars, for instance it allows verification of FIFO queues.

The semantics of the calculus is deterministic [5] and expressed by a labeled transition system, where labels range over a given universe of events. The transition relation $t_1 \xrightarrow{e} t_2;\sigma$ is derivable iff event e

[1] Since RML is a DSL which does not require users to be familiar with LP, we use a more general terminology, but in fact an event type pattern corresponds to a possibly non-ground Prolog atom.

Submitted to:

triggers the transition from term t_1 to t_2 with computed substitution σ; it is inductively defined by a set of rewriting rules. For instance, the following rule (and) defines transition steps for intersection:

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

For every rewriting step a substitution is computed to bind the variables of the event type patterns that have successfully matched the triggering event.

Implementation

RML is compiled into a trace calculus which serves as intermediate language; the interpreter of the calculus is implemented with SWI-Prolog[2], which is the target language for the monitors automatically generated from RML specifications. The Prolog implementation of the interpreter is straightforward because the rewriting rules can be directly translated to corresponding clauses; for instance, rule (and) above is translated into:

```
next(T1 ∧ T2, E, T, S) :-
  !,next(T1, E, T3, S1),next(T2, E, T4, S2),merge(S1, S2, S),conj(T3, T4, T).
```

The atoms in the clause have the following meaning: the cut is required to guarantee determinism; `next(T1, E, T3, S1)` and `next(T2, E, T4, S2)` correspond to the premises $t_1 \xrightarrow{e} t_1';\sigma_1$ and $t_2 \xrightarrow{e} t_2';\sigma_2$, respectively; `merge(S1, S2, S)` corresponds to the side-condition $\sigma = \sigma_1 \cup \sigma_2$ while `conj(T3, T4, T)` tries to apply optimizations to the term `T3`$\wedge$`T4`, based on the laws of the calculus, to get an equivalent but simplified term `T`.

Data variables occurring in event type patterns cannot be correctly managed by directly translating them into corresponding Prolog logic variables, because in RML they are locally scoped, while Prolog logic variables are globally scoped, and it would not be possible to ensure that the Prolog interpreter computes correct substitutions for data variables, since the SWI-Prolog support for regular coinduction is based on unification (see the details below). For instance, the following two RML specifications `S1` and `S2` are not equivalent:

```
S1 = {let fd; open(fd) S1};          S2 = {let fd; S}; S = open(fd) S;
```

In `S1` the declarations of `fd` are recursively nested, while in `S2` there is a unique declaration for `fd`; as a consequence, `S1` accepts traces of events matching `open(fd)` where `fd` is allowed to be instantiated with different values for each event, while `S2` accepts traces of events matching `open(fd)` where `fd` must be instantiated with the same value for each event.

To correctly deal with locally scoped data variables, an RML data variable identifier *id* is translated to the ground term `var(`*id*`)` and substitutions are implemented with association lists with elements of shape `var(`*id*`)=`*term*; to simplify the translation and guarantee that `var(`*id*`)` is ground, in RML identifiers *id* for data variables are required to start with a lower case letter.

In the trace calculus recursive specifications are represented in a more abstract way with regular (a.k.a. cyclic or rational) terms which can be directly mapped to corresponding Prolog terms thanks to the extremely useful support of SWI-Prolog for cyclic terms: unification is supported by default without occurs checks; for instance, unification `X=f(X)` succeeds and instantiates variable `X` with the (only) regular term satisfying that equation.

Another reason, although strictly related to the previous one, to adopt SWI-Prolog, is its native support for *coinductive logic programming* [6]. Since terms are regular, substitution application is more

conveniently defined coinductively; such a definition can be directly turned into Prolog thanks to the *coinduction* SWI-Prolog library which allows declaration of coinductive predicates; when coinductive predicates are involved in the derivation, after each resolution step the interpreter checks whether the selected atom of the goal unifies with any of the previously resolved ones, and if it is the case, then resolution succeeds by coinduction with the corresponding unification. For instance, substitution application for the case of intersection is defined as follows:

```
:- use_module(library(coinduction)).
:- coinductive apply_sub_trace_exp/3.
apply_sub_trace_exp(S,T1 ∧ T2,T3 ∧ T4) :-
  !,apply_sub_trace_exp(S,T1,T3),apply_sub_trace_exp(S,T2,T4).
```

Finally, dictionaries have been recently introduced in SWI-Prolog to manipulate object-like terms of shape `tag{field1: term1, ..., fieldN: termN}`; in the implemented interpreter events are conveniently represented with dictionaries; they are in turn exploited by the SWI-Prolog standard library `library(http/json)` for JSON deserialization and manipulation, which is used to inspect the events received by the monitor.

Performance evaluation

We briefly report on the experiments conducted to evaluate the performances of the SWI-Prolog monitors generated by the implementation of RML; all experiments have been run on an Intel(R) Core(TM) i7-7700HQ CPU at 2.80GHz with a 16GB RAM, running SWI-Prolog version 8.0.3 for x86_64-linux and Linux Ubuntu 18.04.2 LTS, kernel version 4.15.0-54-generic.

Figure 1 shows the evaluation for the specification of different variations of queues[2]; all corresponding sets of traces cannot be defined by context-free grammars. The left-hand-side chart shows that the time required for monitoring a single event linearly increases with the data size (the maximum number of elements in the queue), while the right-hand-side chart shows that the total time required by the monitors for checking entire traces linearly increases with their length, when the size of queues is bounded.

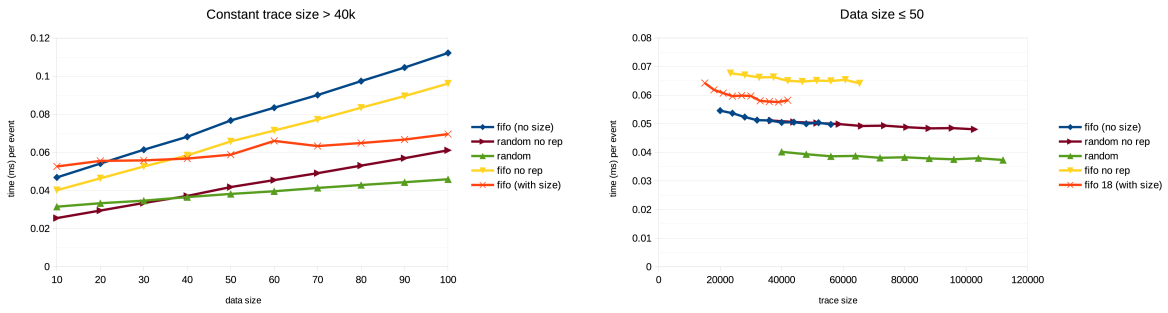

Figure 1: Performance evaluation for several kinds of queues, with variable data and trace size.

[2]See `https://rmlatdibris.github.io/examples/fifo.html` for the details

References

[1] *Introducing JSON*. Available at `https://www.json.org/json-en.html`.

[2] *SWI-Prolog*. Available at `https://www.swi-prolog.org/`.

[3] César Sánchez et al. (2019): *A survey of challenges for runtime verification from advanced application domains (beyond software)*. Formal Methods in System Design 54(3), pp. 279–335, doi:10.1007/s10703-019-00337-w.

[4] D. Ancona, A. Ferrando, L. Franceschini & V. Mascardi (2019): *Runtime Monitoring Language (RML)*. Available at `https://rmlatdibris.github.io/`.

[5] Davide Ancona, Luca Franceschini, Angelo Ferrando & Viviana Mascardi (2019): *A Deterministic Event Calculus for Effective Runtime Verification*. In: Proceedings of the 20th Italian Conference on Theoretical Computer Science, ICTCS 2019, Como, Italy, September 9-11, 2019, pp. 248–260.

[6] Luke Simon, Ajay Mallya, Ajay Bansal & Gopal Gupta (2006): *Coinductive Logic Programming*. In: Logic Programming, 22nd International Conference, ICLP 2006, Seattle, WA, USA, August 17-20, 2006, Proceedings, pp. 330–345.