Runtime Monitoring of Stream Logic Formulae

Sylvain Hallé(✉) and Raphaël Khoury

Laboratoire d'informatique formelle, Université du Québec à Chicoutimi, Chicoutimi, Canada
shalle@acm.org, rkhoury@uqac.ca

Abstract. We introduce a formal notation for the processing of event traces called Stream Logic (SL). A monitor evaluates a Boolean condition over an input trace, while a filter outputs events from an input trace depending on some monitor's verdict; both constructs can be freely composed. We show how all operators of Linear Temporal Logic, as well as the parametric slicing of an input trace, can be written as Stream Logic constructs.

1 Introduction

Trace validation is performed in various areas of computer science. For example, in programming, analyzing a trace of events can be used to determine the success of a test run [2] or for debugging purposes to ensure that methods of an object have been called in the correct order [7]; similarly, a trace can represent a recorded interaction between a client and a web service and one can verify that each client interacts properly with the server according to some predefined protocol [6]. Providing a verdict dynamically, as the events are produced by the system, is its realtime counterpart called *runtime monitoring*.

Several notations have been developed to describe the set of valid traces specific to each use case. Regular expressions are supported by tools like MOP [7]; Monpoly [4], BeepBeep [6] and ProM [8] employ Linear Temporal Logic (LTL) or first-order extensions thereof; Logscope [5] and RuleR [3] use a language based on μ-calculus.

Reasoning about properties on traces can sometimes prove difficult. It is well known, for example, that there exist properties for which neither a "true" nor a "false" verdict can be given for any finite prefix of a trace; however, depending on the notation used, identifying such properties may be complex. Hence in LTL, apart from trivial cases that are easy to spot (the formula $\mathbf{G}\,\mathbf{F}\,p$ is one such example), the general problem reduces to satisfiability solving and belongs to the PSPACE-complete class.

In this paper, we propose a reformulation of formal trace specifications using different base concepts, with runtime monitoring and partial evaluation of trace prefixes in mind. The result is a formal notation for the processing of event

The authors acknowledge the financial support of the Natural Sciences and Engineering Research Council of Canada (NSERC).

J. Garcia-Alfaro et al. (Eds.): FPS 2015, LNCS 9482, pp. 251–258, 2016.
DOI: 10.1007/978-3-319-30303-1_15

traces called Stream Logic (SL), detailed in Sect. 2. SL defines two basic objects, called *monitors* and *filters*. A monitor evaluates a Boolean condition over an input trace, while a filter outputs events from an input trace depending on some monitor's verdict, and both constructs can be freely composed. The formal semantics of SL is defined, and a few syntactical identities are also presented.

We then proceed to highlight some of the advantages of SL over other notations. In Sect. 3, we show how SL subsumes LTL by emulating each of its operators, and describe how conclusions about a monitor's possible outcome can be drawn through purely syntactical manipulations. Finally, Sect. 4 describes a proof-of-concept implementation of a Stream Logic processor showing the feasibility of the approach on a number of use cases. In particular, our SL interpreter provides order-of-magnitude speed gains compared to a tool applying the classical recursive semantics of LTL to monitor a trace.

2 A Calculus for Event Streams

In this section, we describe the formal notation and semantics of Stream Logic, a language for expressing both conditions and filters on finite traces of events. We shall first present each concept intuitively in Sect. 2.1, and then show in Sect. 2.2 the formal semantics of the complete system.

2.1 Basic Constructs

A trace of events is a finite sequence of atomic events, represented as $a, b, \ldots$, taken from some finite set of symbols. Traces will be designated as $\overline{m} = m_1, m_2, \ldots, m_n$. We assume there is a set of predicates $p, q, r, \ldots$, which each return either true ($\top$) or false ($\bot$) for each possible event. When the context is clear, an event symbol a will also stand for the predicate that returns true whenever its input is symbol a, and false otherwise. For a trace $\overline{m} = m_1, m_2, \ldots$, we let $\overline{m}^k$ be the trace obtained from $\overline{m}$ and starting at the k-th symbol: $m_k, m_{k+1}, \ldots$.

Monitors. The first fundamental concept of SL is that of a monitor. A monitor φ takes a trace $\overline{m} = m_1, m_2, \ldots m_n$ as an input, and returns as an output a sequence $\overline{v} = v_1, v_2, \ldots$, where $v_i \in \{\top, \bot\}$. This is noted $\overline{m} : \varphi$. Informally, a monitor is fed messages from $\overline{m}$ one by one, and at any moment, can be queried for its current state, which by definition is the last symbol appended to $\overline{v}$. Hence a monitor queried multiple times without being fed any new message in between will return the same symbol. By definition, a monitor that has not been fed any message returns a third, special value noted ?.

In the following, we will consider a few simple monitors. The first is the "true" monitor, which outputs the sequence $\top, \top, \ldots$. We will abuse notation and use the $\top$ symbol to designate this monitor. Similarly, we have the monitors $\bot$ and ?. Finally, we have the constant monitor, noted c for some constant, which, for any trace $\overline{m} = m_1, m_2, \ldots$, outputs the sequence $\overline{v} = v_1, v_2, \ldots$, where $v_i = \top$ if $m_i = c$, and $v_i = \bot$ if $m_i \neq c$.

These monitors can be combined to produce compound results using the classical Boolean connectives. Given a trace $\overline{m} = m_1, m_2, \ldots$ and two monitors φ and ψ producing sequences $\overline{v}_\varphi$ and $\overline{v}_\psi$, the monitor $\varphi \wedge \psi$ produces the sequence $\overline{v}_{\varphi\wedge\psi}$ that is the pairwise conjunction of symbols in each monitor's output sequence. The case of disjunction and negation monitors can be defined in the expected way.

Clearly, if φ has read k symbols, its output can contain at most k symbols. However, as we shall see, each monitor need not to produce their output symbols at the same time. The conjunction monitor can only output symbol i if monitors φ and ψ have both output symbol i; otherwise, the conjunction monitor must delay the output of symbol i until both values are available (or until a conclusion can be drawn anyway, such as when one of the monitors returns $\bot$). This entails that a monitor can be fed an input message, and not produce the corresponding output symbol immediately.

We also introduce additional binary connectives for monitors. The first is operator $\wedge_1$. Informally, the monitor $\varphi \wedge_1 \psi$ returns the value of the *first* of φ or ψ that is no longer undefined. In the case where φ and ψ both take a value at the same time, $\wedge_1$ behaves like $\wedge$. Connective $\vee_1$ is defined similarly with respect to $\vee$.

Filters. The second important construct in SL is the filter. The filter is an operator which, given some monitor φ and an input trace $\overline{m}$, returns a subtrace retaining only the symbols that match a specific condition. Formally, let $\overline{m} = m_1, m_2, \ldots$ be a message trace, and $\overline{v} = v_1, v_2, \ldots$ be the output sequence for monitor φ on that trace. The expression

$$\overline{m} : \frac{\infty}{\varphi}$$

constructs from $\overline{m}$ the subtrace made only of symbols m_i such that $v_i = \top$. Hence, the filter $\frac{\infty}{c\vee d}$ produces the subtrace made only of symbols c or d.

The ∞ symbol in the top part of the fraction indicates that one is to take all messages from $\overline{m}$ that satisfy φ. We can also indicate to return only one particular message by replacing ∞ by a number. Hence $\frac{k}{\varphi}$ returns only the k-th message that satisfies φ (i.e. $\overline{m}'_k$).

Filters and monitors can be chained; that is, the output of a filter can be given as the input for a monitor, or another filter. If f is a filter and φ is a monitor, then $\overline{m} : f : \varphi$ is the monitor that evaluates φ on trace $\overline{m}$, but is being fed only the input symbols that are returned by f.

Consider for example the following monitor, assuming an input sequence $\overline{m}$:

$$\frac{1}{b \wedge \left(\frac{2}{\top} : c\right)} : \top$$

The filter this time retains for the input trace only symbols m_i that satisfy two conditions. The first is that $m_i = b$. The second is itself a compound monitor, that is given as its input trace the sequence $\overline{m}' = m_i, m_{i+1}, \ldots$. This sequence

Table 1. The formal semantics of SL. In this recursive definition, f and f' are arbitrary filters and φ and ψ are arbitrary monitors.

$$[\![\overline{m}, \top]\!]_i \triangleq \begin{cases} \top & \text{if } 1 \leq i \leq |\overline{m}| \\ \epsilon & \text{otherwise} \end{cases}$$

$$[\![\overline{m}, p]\!]_i \triangleq \begin{cases} \epsilon & \text{if } i > |\overline{m}|, \text{ otherwise...} \\ \top & \text{if } \overline{m}_i \text{ satisfies } p \\ \bot & \text{if } \overline{m}_i \text{ does not satisfy } p \end{cases}$$

$$[\![\overline{m}, \neg\varphi]\!]_i \triangleq \begin{cases} \bot & \text{if } [\![\overline{m}, \varphi]\!]_i = \top \\ \top & \text{if } [\![\overline{m}, \varphi]\!]_i = \bot \\ \epsilon & \text{otherwise} \end{cases}$$

$$[\![\overline{m}, \varphi \wedge \psi]\!]_i \triangleq \begin{cases} \bot & \text{if } [\![\overline{m}, \varphi]\!]_i = \bot \text{ or } [\![\overline{m}, \varphi]\!]_i = \bot \\ \top & \text{if } [\![\overline{m}, \varphi]\!]_i = \top \text{ and } [\![\overline{m}, \varphi]\!]_i = \top \\ \epsilon & \text{otherwise} \end{cases}$$

$$\left[\!\left[\overline{m}, \frac{k}{\varphi}\right]\!\right] \triangleq \begin{cases} [\![\overline{m}^2, \frac{k-1}{\varphi}]\!] & \text{if } [\![\overline{m}, \varphi]\!]_1 = \top \\ [\![\overline{m}^2, \frac{k}{\varphi}]\!] & \text{otherwise} \end{cases}$$

$$[\![\overline{m}, f : \varphi]\!] \triangleq [\![[\![\overline{m}, f]\!], \varphi]\!]$$

first goes through a filter that retains only its second message (i.e. m_{i+1}), and passes it on to the constant monitor c; this monitor outputs the symbol $\top$ only if $m_{i+1} = c$. Hence the filter will return the first message m_i such that $m_i = b$ and $m_{i+1} = c$. The end monitor outputs the $\top$ symbol whenever it receives a message from that filter. The end result is that the monitor returns $\top$ as soon as the property "some b is immediately followed by a c" is observed at least once.

2.2 Semantics

Now that we have described intuitively the basic constructs of SL, we shall formally define the semantics of the language; this is done in Table 1. The notation $[\![\overline{m}, \varphi]\!]$ designates the output trace produced by feeding the input trace $\overline{m}$ to monitor φ. Similarly, the notation $[\![\overline{m}, f]\!]$ defines the output trace produced by feeding $\overline{m}$ to filter f. Since $[\![\cdot]\!]$ represents a trace, we use the subscript notation $[\![\cdot]\!]_i$ to denote the i-th event of that output.

An interesting side effect of this semantics is that it defines a form of buffering for events to be processed by monitors or filters, without the need for managing these buffers explicitly in the notation. As an example, let us consider the filter expression $\frac{\infty}{\frac{1}{c}:\top}$. Let us first apply the finite semantics to determine the output of this filter on the input trace made of the single symbol a. Working from the inside expression outwards, one realizes that $[\![a, \frac{1}{c}]\!] = \epsilon$; this propagates outwards and leads to the conclusion that the filter outputs nothing. However, while the top-level filter outputs nothing after receiving the first event, one can see that it

outputs a after receiving c. In other words, events are implicitly "buffered" by some monitors and some filters until some condition allows them to be released.

3 Applications

In this section, we show potential uses of SL in various applications. We concentrate on a reduction to Linear Temporal Logic, the possibility of simplifying monitor expressions, and the characterization of monitorable properties in a purely syntactical way.

3.1 Linear Temporal Logic

As a first application of SL, we show how temporal operators from Linear Temporal Logic can be rewritten using the concepts of filters and monitors.

The first case is the **F** ("eventually") operator, which we can write as:

$$\mathbf{F}\,\varphi \triangleq \frac{1}{\varphi} : \top$$

In this case, the filter $\frac{1}{\varphi}$ will create a subtrace retaining only the first message that satisfies φ and discarding all others. On that trace, we evaluate the expression $\top$, which will return true as soon as it reads one message from the trace (otherwise the whole expression evaluates to ?).

This translation also conveys the intuitive meaning of the operator: until a message satisfying φ is read, $\mathbf{F}\,\varphi$ is not *yet* true, but can become so in the future (and can never become false). Despite the existence of a single undefined value, the filter tells us whether this value can turn true in the future (when the expression at the right of the filter is $\top$), or turn false in the future (when the expression at the right of the filter is $\bot$), and hence "simulate" four-valued, finite-trace semantics for LTL.

Similarly, for **X**, the monitor simply returns the value of φ on the trace made of the second message, which is written as:

$$\mathbf{X}\,\varphi \triangleq \frac{2}{\top} : \varphi$$

Finally, operator **U** ("until") requires slightly more work:

$$\varphi\,\mathbf{U}\,\psi \triangleq \frac{1}{\neg\varphi} : \bot \;\wedge_1 \frac{1}{\psi} : \top$$

The translation of **U** builds two subtraces from an input trace $\overline{m}$. The first one retains the first message of $\overline{m}$ that does not satisfy φ. The left-hand side of $\wedge_1$ hence returns $\bot$ as soon as a message from $\overline{m}$ is read that does not satisfy φ. Similarly, the right-hand side of $\wedge_1$ returns $\top$ as soon as a message from $\overline{m}$ is read that satisfies ψ. Using these definitions, one can recursively translate any LTL expression into an equivalent filter expression.

3.2 Identities and Monitor Simplification

A number of identities on filters and monitors can be derived from the semantics defined in the previous section.

The first identities apply on filters. For instance, it is clear that the monitor $\perp$, when used as the condition for a filter, will result in nothing being output; this can be expressed as:

$$\overline{m} : \frac{k}{\perp} \equiv \epsilon \tag{1}$$

Dually, passing an input trace through a constant filter, is equivalent to this constant filter.

$$f : \perp \equiv \perp \tag{2}$$

Identities can also be defined for monitors. For example, it is straightforward to conclude that a monitor φ given the empty trace is equivalent to the constant monitor ?, which never outputs any event.

$$\epsilon : \varphi \equiv ? \tag{3}$$

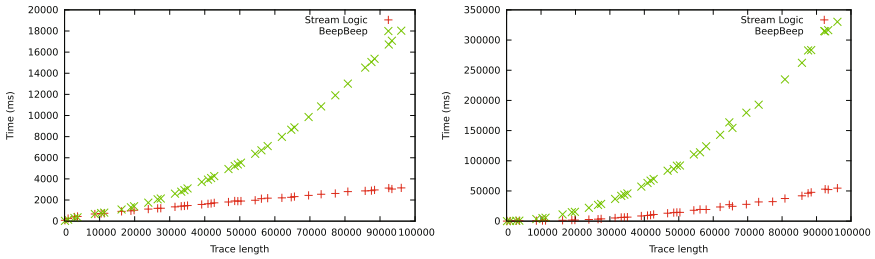
These are a few examples of syntactical identities that can be derived from monitors and filters. These identities can be then applied in a straightforward way to perform simplification of filter and monitor expressions to reason about their possible verdicts. As an example, consider the LTL formula $\mathbf{G}\,\mathbf{F}\,p$. In the context of monitoring finite prefixes of traces, a monitor for this formula must return ? on every message it reads. Realizing this fact using standard techniques involves, for example, the construction of the corresponding Büchi automaton and the discovery that neither state is labelled as accepting. However, writing this expression in SL yields:

$$\begin{aligned} \overline{m} : \frac{1}{\left(\frac{1}{p}\perp\right)}\perp &\equiv \overline{m} : \frac{1}{\perp}\perp \text{ by (2)} \\ &\equiv \epsilon : \perp \text{ by (1)} \\ &\equiv ? \text{ by (3)} \end{aligned}$$

Hence we have seen how, by purely syntactical means, it is possible to simplify the original monitor down to the *constant* ?. This shall be distinguished from the monitor that performs a computation that just happens to return ? all the time.

4 Implementation and Experiments

To assess the feasibility of the approach, we implemented a runtime monitor/filter based on the concepts described in this paper. The implementation



(a) P1 (b) P2

Fig. 1. Total running time with respect to messages received for various traces on sample properties.

is made of 1,600 lines of Java code independent of any external library, and is publicly available online.[1]

We ran a simple benchmark comparing our proof-of-concept implementation of Stream Logic with the latest version of BeepBeep[2], another Java-based runtime monitor written by one of the authors. BeepBeep was chosen among other existing monitors for a number of reasons: first, it is capable of accepting events made of parameter-value pairs in XML format; second, it is also implemented in Java (eliminating differences caused by the implementation language) and has roughly the same size in lines of code; finally, it uses a completely different evaluation algorithm, described in [6], which uses the recursive semantics of Linear Temporal Logic to monitor specifications.

We followed the same methodology as described in [1]. We built a dataset consisting of traces of randomly-generated events, with each event being made of up to ten random parameters, labelled $p_0, \ldots, p_9$ each carrying five possible values. Each trace has a length between 1 and 100,000 events, and 50 such traces were produced. Two properties were then verified on these traces. Property #1 is $\mathbf{G}\, p_0 \neq 0$, and simply asserts that in every event, parameter p_0, when present, is never equal to 0. Property #2 is $\mathbf{G}\,(p_0 = 0 \rightarrow \mathbf{X}\, p_1 = 0)$: it expresses the fact that whenever $p_0 = 0$ in some event, the next event is such that $p_1 = 0$.

Each property was written both as an SL expression and as an LTL formula, and these were sent respectively to our proof-of-concept implementation of an SL engine and to BeepBeep. The running time for evaluating these properties on each trace was computed and plotted in Fig. 1.

Two major observations can be made from these preliminary results. First, the SL engine, globally, performs faster than BeepBeep for a majority of properties and traces: in the case of P1, BeepBeep averages 5,000 events per second, and SL obtains roughly 25,000. However, the gap between both tools widens, both as the trace lengthens and as the property to process becomes more complex. In the case

[1] https://bitbucket.org/sylvainhalle/streamlogic.
[2] http://sourceforge.net/projects/beepbeep, version 1.7.6.

of P2, SL still handles about 2,000 events per second, while BeepBeep is down at approximately 280.

5 Conclusion

In this paper, we have shown a formal notation for the processing of event traces called Stream Logic (SL). The distinguishing point of SL is that it is based upon two simple concepts (filter and monitor), which, when freely combined, suffice to support all operators of Linear Temporal Logic. We have shown experimentally on a sample dataset how the application of the formal semantics of SL yields an evaluation algorithm with better running time compared to the traditional LTL evaluation algorithm presented in past literature. The promising results obtained on the proof-of-concept implementation discussed in this paper lead to a number of extensions and improvements over the current method. In particular,we shall investigate whether there exists a characterization of monitorable or enforceable properties based on syntactical properties of SL expressions.

References

1. Barre, B., Klein, M., Soucy-Boivin, M., Ollivier, P.-A., Hallé, S.: MapReduce for parallel trace validation of LTL properties. In: Qadeer, S., Tasiran, S. (eds.) RV 2012. LNCS, vol. 7687, pp. 184–198. Springer, Heidelberg (2012). doi:10.1007/978-3-642-35632-2_20
2. Barringer, H., Havelund, K.: TraceContract: a scala DSL for trace analysis. In: Butler, M., Schulte, W. (eds.) FM 2011. LNCS, vol. 6664, pp. 57–72. Springer, Heidelberg (2011)
3. Barringer, H., Rydeheard, D., Havelund, K.: Rule systems for run-time monitoring: from eagle to RuleR. J. Logic Comput. **20**(3), 675–706 (2010)
4. Basin, D., Klaedtke, F., Müller, S.: Policy monitoring in first-order temporal logic. In: Touili, T., Cook, B., Jackson, P. (eds.) CAV 2010. LNCS, vol. 6174, pp. 1–18. Springer, Heidelberg (2010)
5. Groce, A., Havelund, K., Smith, M.H.: From scripts to specifications: the evolution of a flight software testing effort. In: Kramer, J., Bishop, J., Devanbu, P.T., Uchitel, S. (eds.) ICSE (2), pp. 129–138. ACM (2010)
6. Hallé, S., Villemaire, R.: Runtime enforcement of web service message contracts with data. IEEE Trans. Serv. Comput. **5**(2), 192–206 (2012)
7. Meredith, P.O., Jin, D., Griffith, D., Chen, F., Rosu, G.: An overview of the MOP runtime verification framework. STTT **14**(3), 249–289 (2012). doi:10.1007/s10009-011-0198-6
8. Verbeek, H.M.W., Buijs, J.C.A.M., van Dongen, B.F., van der Aalst, W.M.P.: XES, XESame, and ProM 6. In: Soffer, P., Proper, E. (eds.) CAiSE Forum 2010. LNBIP, vol. 72, pp. 60–75. Springer, Heidelberg (2010)