A Rule-Based Constraint Language for Event Streams

Isaac Mackey, Jianwen Su

Dept. of Computer Science, University of California, Santa Barbara

Abstract
Software systems and the availability of data collecting devices lead to large amounts of data in the form of event streams. The ability to query, analyze, reason about, and monitor event streams is in high demand. In this work, we formalize a model of events and event streams in workflow systems, and a constraint language by extending Datalog with timestamp variables and gap constraints (inequalities with constant offsets) over a time domain. We illustrate an application of our language by expressing rules with time constraints and data dependencies for workflow systems, where event streams from workflow enactments are monitored in real-time to ensure compliance with policies, regulations, and business rules.

Keywords
temporal constraints, monitoring, events

1. Introduction

Event streams are increasingly available, either from data collecting devices for IoT systems or generated by software systems for applications including workflow systems and distributed systems [1]. Querying data streams have been studied in research communities with SQL as a primary language [2], though other works address the challenge of maintaining query answers incrementally for logic programs [3, 4, 5]. In addition to query answering, event stream processing can identify exceptional situations, i.e., violations of system policies and regulations. Arising from operational and business rules, security policies, and compliances, OMG introduced a rule-based language SBVR [6]. Consequently, efficient and effective reasoning methods for constraint languages becomes an important element of event stream processing. In our earlier work [7, 8], we formulated a Datalog-like language to specify desirable event behavior during workflow execution, and developed several monitoring techniques. In this paper, we present the language and overview the monitoring problem and key results.

Syntactically, our language generalizes Datalog by allowing (i) timestamp variables and (ii) conjunctions in a rule head. Our previous work primarily uses this language for expressing constraints. To this end, it generalizes tuple- and equality-generating dependencies (TGDs and EGDs) in relational databases [9] by allowing timestamp variables and gap constraints (inequalities with constant offsets) over a time domain. Our language is similar in essence to, for example, DEDALUS [10] (which only allows single time variables in the rule body), *LARS* [11]

Datalog 2.0 2022: 4th International Workshop on the Resurgence of Datalog in Academia and Industry, September 05, 2022, Genova - Nervi, Italy
imackey1415@gmail.com (I. Mackey); su@ucsb.edu (J. Su)
0000-0001-6462-1912 (I. Mackey); 0000-0002-4637-1339 (J. Su)

(which uses window operators for expressing intervals, instead of time variables that can be used in multiple (in)equalities), and "temporal" Datalog [12] (which does not allow inequalities over time variables). Streamlog [13] is another Datalog extension for event stream processing, but its purpose is to reduce blocking queries, so it does not allow rules whose head variables match timestamps before the timestamps for body variables. In comparison a key omission in our language is the ability to generate new tuples (as our current focus is event monitoring).

In Sec. 2 we formalize a model of events and event streams, and present the constraint language. In Sec. 3, we describe the *monitoring* and *early detection* problems and present relevant results. Finally, we discuss generalizations of these results and further applications in Sec. 4.

2. A Rule-Based Constraint Language

In this section, we present a language for writing constraints on events in workflow enactments. The language is a variant of Datalog with two notable distinctions: the use of (i) timestamp variables, and (ii) multiple atoms in a rule head. Furthermore, we adopt rules as constraints rather than for deriving facts, which means the satisfaction of constraints by enactments is the primary concern, rather than the deduction of new facts. This makes our language a natural extension of tuple- and equality-generating dependencies [9]. We define below the key notions of the language, including "events," "enactments," and "rules."

We assume the existence of six countably infinite, pairwise disjoint sets of: *activity names* **A**, *attributes* **C**, discrete and ordered *timestamps* **T**, *(data) values* **D**, *variables* **V**, and (workflow) enactment *identifiers*, or simply IDS, **I**. Without loss of generality, we use the natural numbers $\mathbb{N}$ as timestamps.

Activities are atomic units of work in a workflow, each with a name and data attributes. The execution of an activity produces an "event" at a timestamp at which the activity's data attributes obtain values.

Definition: An *activity* $A(\mathrm{c}_1, ..., \mathrm{c}_n)$ has a name $A \in \mathbf{A}$ and an enumeration of n ($\geqslant 0$) distinct attributes $\mathrm{c}_1, ..., \mathrm{c}_n$ from **C**. An *event* of $A(\mathrm{c}_1, ..., \mathrm{c}_n)$ is a tuple $e{=}(\tau, A, \pi, \mu)$ where $\tau \in \mathbf{T}$ is a timestamp, $\pi \in \mathbf{I}$ an ID, and $\mu : \{\mathrm{c}_1, ..., \mathrm{c}_n\} \to \mathbf{D}$ a mapping to data values.

A *partial enactment* of a workflow system is a set of events such that (i) all events share the same enactment ID, (ii) there is exactly one special *START* event (that marks its beginning) and at most one *END* event (that marks its completion), (iii) the timestamp of *START* is less than that of all other events, and (iv) the timestamp of *END*, if it is present, is greater than that of all other events. An *enactment* is a partial enactment that contains an *END* event.

A (partial) enactment can be stored as a relational database, e.g., the database in Fig. 1 for an enactment with ID π_1. For example, the first row of the Request table indicates a Request event for π_1 from user Alice with account $a3$ at time 2.

To express constraints on enactments, we use a language introduced in [7, 14] and extended with data in [8], presented below. An *event atom* is an expression "$A(v_1, ..., v_n)@x$" where $A(\mathrm{c}_1, ..., \mathrm{c}_n)$ is an activity and $v_1, ..., v_n, x$ are variables in **V**; x is called the *timestamp* variable. A *gap atom* is an expression "$x{\pm}\epsilon\,\theta\,y$" where x, y are timestamp variables, ϵ (the gap) is a timestamp in **T**, and $\theta \in \{<, \leqslant, \geqslant, >, =\}$ is an (in)equality predicate. Without loss of generality, we use natural numbers as timestamps, with the associated $+/-$ operations and ordering.

Start

ID	ts
π_1	1

Request

ID	*user*	*account*	ts
π_1	Alice	a3	2
π_1	Alice	a4	6

Approval

ID	*user*	ts
π_1	Alice	3

Reserve

ID	*user*	*account*	ts
π_1	Alice	a4	8
π_1	Alice	a3	9

Payment

ID	*user*	*account*	ts
π_1	Alice	a3	8
π_1	Alice	a4	9

End

ID	ts
π_1	15

Figure 1: An enactment π_1 with events, as a relational database

Definition: A *rule* is an expression "$\varphi \to \psi$" where φ (the *body*) and ψ (the *head*) are sets of event and gap atoms such that each variable in a gap atom in φ occurs in an event atom in φ and each variable in a gap atom in ψ occurs in an event atom in $\varphi \cup \psi$.

Example 1. Consider an IaaS provider that offers high-performance cloud computing rentals. The service is managed by a workflow with activities like Request and Payment, which carry attributes like *user* and *account*. Fig. 1 shows a partial enactment of the workflow. Consider that the IaaS provider checks enactments against business rules; these rules may measure service availability, quality, etc. The provider has the following rule:

$$r_1 : \text{Request}(u, a)@x \to \text{Payment}(u, a)@y, y \leqslant x+7$$

This rule r_1 indicates that a payment for a rental must be completed within 7 days of the request by the same user and account. Observe that the timestamp x of the Request event and the timestamp y of the Payment event are constrained to give an upper bound (or "deadline") for the payment.

A second rule r_2 requires the presence of Reserve and Payment events (with a matching user and account) given certain Request and Approval events:

$$\begin{aligned} r_2 :\ & \text{Request}(u, a)@x, \text{Approval}(u)@y, x \leqslant y \leqslant x+7 \\ & \to \text{Reserve}(u, a)@w, \text{Payment}(u, a)@v, x \leqslant w \leqslant x+3, y \leqslant v \leqslant y+7, v \leqslant w+4 \end{aligned}$$

In r_2, the timestamp variable x in the body constrains another timestamp variable y in the body, restricting which pairs of events trigger r_2. Also, x and y in the body both constrain v, w in the head. ❚

Rule satisfaction is defined for enactments using variable assignments: an *assignment* is a mapping from $\mathbf{V}$ to $\mathbf{D} \cup \mathbf{T}$: timestamp variables are mapped to $\mathbf{T}$ and all other variables to $\mathbf{D}$. An assignment is *complete* if it is a total mapping for the variables in an atom (or set of atoms). An enactment η satisfies an event atom $A(v_1, ..., v_n)@x$ under a complete assignment μ if $(\mu(x), A, \xi, \mu(v_1), ..., \mu(v_n))$ is an event in η and ξ is the ID of η. Let $r\colon \varphi(\bar{x}, \bar{y}) \to \psi(\bar{y}, \bar{z})$ be a rule with distinct variables $\bar{x}, \bar{y}, \bar{z}$. An enactment η *satisfies* r if η satisfies the formula $\forall \bar{x}\bar{y}(\varphi(\bar{x}, \bar{y}) \to \exists \bar{z}\psi(\bar{y}, \bar{z}))$ in first-order logic; a *violation* of r is an assignment α such that η satisfies φ with α and there is no assignment β that extends α such that η satisfies ψ with β.

3. Monitoring and Early Detection

In this section, we describe an application of rules for monitoring enactments studied in our recent work [15, 14, 8]. We define the monitoring problem and then present two results: one concerning the subclass of dataless, simple rules and another concerning early violation detection for the general class of rules.

The rule satisfaction problem is to test if specified rules are satisfied by a (complete) enactment. Often, however, it is possible to know a rule violation is inevitable before the enactment is complete if the violation is present in all of its possible futures. Consider r_1 in Ex. 1, a Request event at time x is a violation if no Payment event is observed by time $x + 7$. Then, a violation can be reported before the enactment completes once time $x + 8$ is observed. It is advantageous to monitor the inevitability of violations as the partial enactment is updated. To formulate the monitoring problem, each partial enactment is treated as a relational database, and a "batch" holds the set of incoming events.

Definition: A *batch* for a partial enactment η is a finite set Δ of events such that (i) all events in Δ have the same timestamp, greater than all timestamps in η, (ii) the ID of each event in Δ is the ID of η, (iii) if Δ has *START*, η is the empty set, and (vi) if η has *END*, Δ is the empty set.

Given a partial enactment η and a batch Δ, the (online) *monitoring problem* is to decide if every enactment containing the union $\eta \cup \Delta$ violates a (set of) specified rule(s).

In [15, 14] we study this problem for events carrying no data. A rule is *dataless* if all of its event atoms have no data attributes, i.e., each event atom has the form $A@x$. We develop a translation into linear temporal logic (LTL) for subclasses of dataless, "simple" rules. Recall that the body or head is a conjunctive formula. These can be represented as an undirected graph. Let ϕ be a rule body or rule head. The *graph of* ϕ is an undirected graph $G_\phi = (V, E)$ such that V is the set of timestamp variables in ϕ and E is the set of pairs (x, y) such that ϕ contains a gap atom using both x and y. We say ϕ is *acyclic* if G_ϕ is acyclic. A rule is *simple* if the body and head are both acyclic and they share at most one variable.

In [14], we show that the monitoring problem can be solved for dataless, simple rules by translating rules into LTL formulas, then constructing finite state machines from LTL formulas on finite traces using existing techniques [16]. Then, detecting violations is reduced to applying a state transition function at each timestamp and checking the reachability of an accepting state. We use the temporal operators *next* and *future* in future-time LTL [17], as well as *past* and *yesterday* [18]. It was stated:

Theorem 2. [14] Let r be a simple, dataless rule. An LTL formula γ_r can be effectively constructed such that for all enactments η,

η satisfies r *iff* η satisfies the LTL formula γ_r.

We also studied the monitoring problem for the general class of rules, including enactments with data. Consider r_2 in Example 1, events Request and Approval at times x and y, resp., with $x \leqslant y \leqslant x{+}7$. There may be a violation at the earlier of $x{+}3$ and $y{+}7$ if the corresponding head event doesn't arrive. Calculating the earliest time a violation becomes inevitable is the *early detection* problem. Notably, detecting violations early is much desired as it may, for example, allow the workflow system to reclaim resources from erring enactments. In [8], we present an algorithm for early detection. The core technical development is to calculate for each rule body assignment (a potential violation), the latest time it can be extended (the "deadline") w.r.t. partially evaluated head gap atoms. The deadline informs the monitoring algorithm when to report violations and can result in violations detected significantly earlier than the enactment's *END* event. In the following theorem, we assert that our algorithm reports violations at the earliest possible time, i.e., when processing a batch update that makes a violation inevitable.

Theorem 3. [8] Let r be a rule, η a partial enactment, and Δ a batch for η. It can be effectively determined if some enactment containing η satisfies r but no enactments containing $\eta \cup \Delta$ satisfy r.

The algorithm in [8] processes multiple enactments and reports every variable assignment corresponding to a violation. While the current implementation uses relational algebra to handle event data and imperative subroutines to calculate deadlines, it appears that the algorithm can be mostly expressed in Datalog, which could provide an alternative means of evaluation.

4. Conclusions

Complex temporal constraints occur in many application domains (IoT, cyber-physical systems, etc.). For IoT devices, memory and network limitations demand space-efficient algorithms for stream processing and tolerance of out-of-order events. In cyber-physical systems, violations must be anticipated as early as possible so that the system can maintain a safe state; this requires studying early detection for constraints that mix time, space, and other resources. Finally, it is desirable to have rules that check for the absence of events, generate facts, and allow summaries and aggregates, alongside rules that express constraints.

References

[1] A. Margara, E. D. Valle, A. Artikis, N. Tatbul, H. Parzyjegla (Eds.), International Conference on Distributed and Event-Based Systems, ACM, ACM, 2021.

[2] J. Xie, J. Yang, A survey of join processing in data streams, in: Data Streams, Springer, 2007, pp. 209–236.

[3] A. Gupta, I. S. Mumick, V. S. Subrahmanian, Maintaining views incrementally, in: Proc. ACM Conference on Management of Data (SIGMOD), 1993, pp. 157–166.

[4] D. Anicic, P. Fodor, S. Rudolph, R. Stühmer, N. Stojanovic, R. Studer, A rule-based language for complex event processing and reasoning, in: International Conference on Web Reasoning and Rule Systems, Springer, 2010, pp. 42–57.

[5] M. Gebser, T. Grote, R. Kaminski, P. Obermeier, O. Sabuncu, T. Schaub, Stream reasoning with answer set programming: Preliminary report, in: Thirteenth International Conference on the Principles of Knowledge Representation and Reasoning, 2012.

[6] I. S. Bajwa, M. G. Lee, B. Bordbar, Sbvr business rules generation from natural language specification, in: 2011 AAAI Spring Symposium Series, Citeseer, 2011.

[7] I. Mackey, J. Su, Mapping business rules to ltl formulas, in: ICSOC 2019, 2019, pp. 563–565.

[8] I. Mackey, R. Chimni, J. Su, Early detection of temporal constraint violations, in: 29th International Symposium on Temporal Representation and Reasoning (TIME), to appear, 2022.

[9] S. Abiteboul, R. Hull, V. Vianu, Foundations of Databases, Addison-Wesley, 1995.

[10] P. Alvaro, W. R. Marczak, N. Conway, J. M. Hellerstein, D. Maier, R. Sears, Dedalus: Datalog in time and space, in: International Datalog 2.0 Workshop, Springer, 2010, pp. 262–281.

[11] H. Beck, M. Dao-Tran, T. Eiter, Lars: A logic-based framework for analytic reasoning over streams, Artificial Intelligence 261 (2018) 16–70.

[12] A. Ronca, M. Kaminski, B. C. Grau, B. Motik, I. Horrocks, Stream reasoning in temporal datalog, in: Proceedings of the AAAI Conference on Artificial Intelligence, volume 32, 2018.

[13] C. Zaniolo, Logical foundations of continuous query languages for data streams, in: International Datalog 2.0 Workshop, Springer, 2012, pp. 177–189.

[14] I. Mackey, J. Su, Mapping singly-linked, acyclic rules to linear temporal logic formulas, in submission, 2022.

[15] I. Mackey, J. Su, Mapping business rules to ltl formulas, in: ICSOC 2019, 2019, pp. 563–565.

[16] G. De Giacomo, M. Y. Vardi, Linear temporal logic and linear dynamic logic on finite traces, in: IJCAI'13 Proceedings of the Twenty-Third international joint conference on Artificial Intelligence, Association for Computing Machinery, 2013, pp. 854–860.

[17] A. Pnueli, The temporal logic of programs, in: FoCS, 1977.

[18] O. Lichtenstein, A. Pnueli, L. Zuck, The glory of the past, in: Workshop on Logic of Programs, Springer, 1985, pp. 196–218.