Temporal Correlation Patterns

Intersecting Joins, Streams, Events and Reactive Programming

REBLS '15

Oliver Bračevac
TU Darmstadt, Germany

ABSTRACT

Reactive languages provide a number of abstractions such as joins, streams, events and signals with capabilities to filter and correlate data. However, these languages/frameworks are often limited by the use of low-level combinators and require boilerplate code for complex data correlations.

In this paper, we argue that declarative temporal pattern matching in the style of Complex Event Processing enriches current language designs with expressive and concise data correlations. Complex event patterns define high-level events from structural and temporal constraints on multiple event sources. As such they constitute a promising addition to the reactive languages toolbelt. Nonetheless, there are highly diverse variants of pattern matching semantics, where none of the established languages supports the full spectrum, currently. We outline our current research in designing a universal, general purpose semantics for complex event correlation in reactive languages.

1. INTRODUCTION

There has been a surge of languages, libraries, frameworks and analytics engines which provide *reactive abstractions* for processing implicit or explicit flows of data [16, 2], be it events, streams [5], channels or signals [9, 14, 17]. In these systems, developers manipulate the flow of data with transformations (e.g., `flatMap`), filters and combinators (e.g., `zip`, `join`). In a broader sense, programmers *correlate* or find *patterns of coincidence* in ever changing data sources.

This paper analyzes the data correlation capabilities of reactive systems and relates them to the existing body of research on *Complex Event Processing* (CEP) [8]. The latter offers feature-rich, declarative, pattern-based queries involving structural and temporal constraints on events in order to identify high-level situations of interest. As a result of taking the CEP standpoint, we view data correlation as a temporal pattern matching problem. For example, determining the route of the five most expensive commodities in the last 24 hours may involve steps like the following. Specifying a "top-five" aggregate, retaining event data in a 24-hour window and finding a Kleene star [1, 10, 7] among shipment events which must be consecutive by their time stamps, share the same commodity ID and detect where two consecutive shipments must agree on the coordinates, the end and the start.

We demonstrate that abstractions for data correlation in current reactive systems have limited expressivity and are special cases of CEP-like temporal patterns. We gain insights on how a sufficiently expressive pattern language may enable seamless integration of different reactive paradigms in applications. Finally, we take the perspective of identifying declarative synchronization and coordination in concurrent programming as special cases of data correlation.

This approach provides a significant gain in expressivity but it is challenging. Temporal pattern matching bears considerable complexity and variability due to a wide array of parameters controlling the matching semantics, such as selection, consumption, contiguity and reasoning about occurrence time. For example, given a sequence pattern "a followed by b" and an observation sequence *abab*, are there two or three occurrences? Both answers may be acceptable, depending on the use case. For example, *continuity* [1, 21] is a semantic parameter which controls what is considered a followup event. We argue that developers should have the full control over the semantic options for temporal pattern matching in order to quickly adapt to new and changing requirements. Instead, currently, no reactive, stream, CEP or related language supports the full spectrum of pattern matching semantics for correlations.

As a first step to address this issue, we propose CorrL, a formal semantic framework based on finite automata, which is suitable to reason about the design space for data correlations. CorrL is capable of expressing joins, partial order patterns, stream transformations, aggregations and timing predicates. This paper outlines our current work on developing CorrL and future research directions.

2. CORRELATION IN REACTIVE SYSTEMS

For brevity, we will focus our discussion on *join patterns* (joins) as conceived by Fournier and Gonthier [12]. Join patterns are declarative *synchronization patterns* for concurrent programming in asynchronous message passing systems. Programmers specify patterns to synchronize over a finite number of message occurrences on asynchronous channels. Associated to a pattern is its *reaction*, which is a computation that triggers each time the join pattern matches successfully.

A standard example is a message exchange

$$\texttt{alice}\langle a\rangle \ \texttt{\&}\ \texttt{bob}\langle b\rangle \ \triangleright\ \texttt{bob}\langle a\rangle \ \parallel\ \texttt{alice}\langle b\rangle$$

which atomically swaps message payloads on the two channels `alice` and `bob`. The part to the left of $\triangleright$ specifies a synchronization pattern on the channels (separated by `&`). Each message on both channels carries a payload value. The expression to the right specifies the reaction, which is to send bob's payload (bound by the pattern to b) to alice and alice's payload (bound to a) to bob, in parallel. Triggering a reaction means to atomically consume the messages matching the join pattern from the channels and invoke the reaction

```
1  Serializer(src_1,..., src_n, N, out) :=
2    //take the union of all source streams
3    let srcs = union(src_1,...,src_n) in
4
5      //initialize buffer with the first frame
6      srcs⟨frame⟩ if frame.id == 0 ▷
7        buffer⟨frame :: Nil⟩ ‖ last⟨0⟩
8
9      //append to buffer in the order of the id
10     srcs⟨frame⟩ & buffer⟨fs⟩ & last⟨n⟩
11       if frame.id == n + 1 ▷
12         buffer⟨fs·frame⟩ ‖ last⟨frame.id⟩
13
14     //output buffer in chunks
15     buffer⟨fs⟩ if fs.length ≥ N ▷
16       buffer⟨fs.drop(N)⟩ ‖ out⟨fs.take(N)⟩
```

Figure 1: Stream serializer pseudocode with join patterns, guards and pattern matching.

body with the parameters extracted from the pattern. Additionally, if a message matches multiple patterns, exactly one of them gets to consume it, nondeterministically.

Since their introduction, a number of language designs incorporating joins emerged, such as JoCaml [6], Polyphonic C♯ [3], EventJava [11], JErlang [15] or JEScala [20]. Not to mention library designs integrating joins into the pattern matching facilities of the host language [13]. These languages add useful features to the basic joins concept, such as guarded pattern matching, synchronous channels and pattern matching in a specific order. On top of that, recent work [19] describes lock-free algorithms for scalable and efficient implementations of join patterns.

Most of the aforementioned designs apply joins to solve general concurrency problems. However, they are also useful for event and stream processing, which is why found their way into stream processing libraries, such as Rx.[1]

To illustrate the utility of joins for streams, assume we want to implement part of a player for on-demand video streaming. Videos are chopped up into individual frames, which are streamed from multiple distributed caches in parallel. Frames are ordered by a consecutive `id` number. A player component thus has to serialize frames into the right order. It also should divide the serialized video stream into fixed-size chunks. We model a serializer via join patterns, assuming "channels are streams". Figure 1 shows a pseudocode for such a stream serializer over n source streams $\texttt{src}_i$ and a given chunk size N, which outputs the stream of chunks to the sink `out`. We use advanced language features found in current join languages, such as filtering and correlation with guards. Our example correlates data observed on the `srcs`, `buffer` and `last` streams via join patterns and filters.

Notably, the serializer manages a *state* in form of a message on channel `buffer`[2], carrying a list of ordered frames and `last`, which keeps track of the latest frame in the serialization. The last two joins specify transition rules on the state, i.e., they consume a matching state in the pattern matching process and emit a successor state in the reaction. After initialization, the implementation should maintain the invariant that there is exactly one such message on `buffer` in order to function correctly. Similarly, there should be at most one message on the `last` channel. Such as property is closely related to the notion of *signals* in reactive programming [17]. A proper type system which supports signals could help in checking the invariant statically. On the other hand, what happens to frames delivered out of order? E.g., the buffer is at frame 1000 but the next observed frame has number 2100. An implementation has to retain the frame until the buffer is at frame 2099, i.e., the `srcs` channel represents a collection of frames. In reactive programming terms, a channel is therefore a signal carrying a collection type, where singleton collection types, such as `last`, are important for state representation. Our example is evidence that streams, joins and signals can and should be expressible in a common language or at least be interoperable. To the best of our knowledge, no such integration has been attempted yet.

There are different variations for the join-pattern matching semantics. Joins in the standard Join Calculus compete for the messages in the channels and there is no prioritization of patterns. Some implementations chose this semantics and provide basic fairness guarantees in addition (e.g. [3]). For our serializer example in Figure 1, fairness means that the third pattern fires sufficiently often and is not dominated by the second one, i.e., the entire video is delivered over `out` eventually. However, for the application domain, this guarantee is too weak. Video playback should not be "choppy", which depends on timely and continuous delivery of chunks over `out`. We could improve the situation by assigning a priority to the third rule, in order to output available chunks as soon as possible. Other join implementations, such as JErlang [15], allow syntactic prioritization, but drop nondeterminism. Currently, no implementation allows mixed modes, i.e., prioritize some patterns and let other patterns compete.

A next logical step is to annotate percentages to patterns, controlling their statistical occurrence over time. This is potentially useful for probabilistic experiments, games and controlled resource utilization (e.g., load balancing, scheduling). Ultimately, the join language should provide abstractions that allow for a *dynamic adjustment* of these percentages. To fully support a smooth video stream, one has to dynamically adjust the buffer size and firing frequency of the third rule in Figure 1, accounting for the current input rates. It is difficult to achieve this in a concise and declarative way in current join implementations.

More importantly, programmers cannot choose from the full spectrum of matching semantics for joins in today's implementations and require intrinsic knowledge of their inner workings. Committing to a specific semantics is disadvantageous when facing different or changing software requirements.

Finally, joins are a special kind of temporal pattern, which requires no particular occurrence order or timing among the events, i.e., joins require a trivial partial order. Other sensible variants are desirable, but non-trivial to express, e.g., selecting only events which are at most 10ms apart or events whose payload is above average over the past hour. Such examples are instances of CEP-like temporal patterns, which are more general with respect to ordering, temporal constraints and aggregations. However, CEP-like patterns do not address the aforementioned issues of fairness and priority.

[1] `https://github.com/ReactiveX/RxJavaJoins`

[2] We could eliminate `buffer` altogether, specifying a N-way join pattern on `srcs`. For illustrative purposes, we chose this verbose variant.

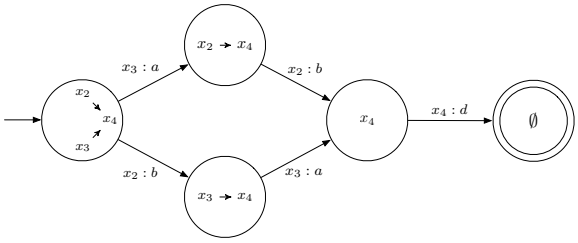

Figure 2: Example of a partial order pattern in CorrL.

3. THE CORRL APPROACH

The first goal of CorrL is to provide a formally grounded framework for studying different event correlation semantics. In the long term, we want to design a language for CEP-like temporal pattern matching which enhances existing reactive systems. We found that most works on CEP focus on implementation aspects and neglect full specifications of their query semantics, hence the need for a semantic framework. It turns out that finite automata are often the semantic basis of CEP systems (e.g., [10, 1, 7]), each coming with its own definitional variations. Our efforts so far consisted of designing an NFA-based intermediate representation (IR) which we believe to be sufficiently general to express the existing CEP semantic models.

Figure 2 shows how CorrL models partial order patterns with automata. We require that events with name x_2 and x_3 must occur before an event x_4. We take the partial order itself as the state description. Transitions have guards, consisting of typed name bindings $x_2 : b$ and a predicate (omitted for brevity), e.g., if the next observed event is of type b satisfying the predicate, then the transition binds it to the name x_2, so that predicates in subsequent transitions can refer to this event.

CorrL's IR has a formally specified operational semantics and is agnostic of query languages, in order to study and support different language designs in the future. We deviate from standard NFAs by a) assigning a local memory which stores partial match and aggregation data to running computations and b) generating output (i.e., complex events) on success, which is derived from the match data. Each state may transition on a particular type of event, even complex events generated by other automata. Hence, a temporal pattern is embodied by a collection of particular automaton definitions, which together form a propagation graph similar to reactive programming. Each automaton has a set of currently incomplete pattern match attempts. If an attempt is successful, the automaton generates a complex event and forwards the result to its dependent automata. The propagation semantics is similar to ELM [9]. It incorporates asynchronous message passing and enables concurrent and pipelined executions, while ensuring consistent outputs.

3.1 Many-Worlds Interpretation

One of the distinguishing features of the CorrL operational semantics is how it models variability in pattern matching. It embraces ambiguity in the sense that on every choice point in the pattern matching process, all choices are taken. Recall the contiguity semantic option from Sec. 1, that controls which events are uninteresting for sequence patterns. Thus, of the two outcomes in the example, CorrL will yield the one with the maximum amount of matches, where different matches may overlap on shared events. On an operational level, this means that whenever an ongoing automaton computation can perform a transition, it forks into two copies, where one stays in the current state and the other transitions into the next state. We observed similar techniques in CEP systems, such as [18, 7].

To emulate the semantics of a specific system, we supply a function which "cuts away" results from the set of all possible pattern matches, where the resulting set contains precisely the results which the system would produce. A formal model is thus obtained from the most general semantics of CorrL and an appropriate *cut function*.

Another semantic option is *selection order*, which controls the order in which pattern matching considers the events and may reduce the set of matches. For example, "the first a followed by the last b" is different from "any a followed by any b". CEP query languages such as TESLA [7] or Esper[3] have corresponding selection qualifiers to achieve this. In CorrL we can express the selection order by a cut function.

4. OUTLOOK

Our work on CorrL opens a number of interesting questions which we would like to answer in future work. The semantic framework separates the fundamental shape of a pattern (embodied by automata definitions) from semantic variations (embodied as cut functions). At the level of a high-level query language, this separation facilitates adapting pattern definitions to new requirements (i.e., change the cut function). This approach is also beneficial for studying new kinds of selection qualifiers or other restrictions on the result set, for which there may not be any language construct available, yet. However, we pay a hefty price if we keep this separation at the implementation level. The set of all possible pattern matches may yield exponentially many results, which we would have to retain and pass to the cut function.

Ideally, we should be able to discard automata computations as early as possible, i.e., *fuse* the cut computation into the pattern matching process, but in a way which is generic and extensible and not restricted to a few well-known contiguity and selection order strategies. The fusion process should yield efficient pattern matching code, e.g., if we modeled standard join patterns (Sec. 2), the performance should be comparable to [19]. We are currently working towards a JVM implementation of CorrL's semantic framework written in Scala.

CorrL so far models the variability of temporal pattern matching on the semantic level. It is also important to expose the full range of parameters at the language level to users and to abstract over variants. We are evaluating query DSLs based on object algebras, where recent work in the stream processing domain appears promising [4].

In Section 2 we pointed out interesting variations of join patterns, which control fairness and priorities. It is currently not clear if and how we can model all of them with automata and cut functions alone or if the semantic framework must be enriched.

Acknowledgements.

This work has been supported by the European Research Council, grant No. 321217.

[3] `http://www.espertech.com/products/esper.php`

5. REFERENCES

[1] J. Agrawal, Y. Diao, D. Gyllstrom, and N. Immerman. Efficient Pattern Matching over Event Streams. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, SIGMOD '08, pages 147–160, New York, NY, USA, 2008. ACM.

[2] T. Akidau, R. Bradshaw, C. Chambers, S. Chernyak, R. J. Fernández-Moctezuma, R. Lax, S. McVeety, D. Mills, F. Perry, E. Schmidt, and S. Whittle. The Dataflow Model: A Practical Approach to Balancing Correctness, Latency, and Cost in Massive-scale, Unbounded, Out-of-order Data Processing. *Proceedings of the VLDB Endowment*, 8(12):1792–1803, 2015.

[3] N. Benton, L. Cardelli, and C. Fournet. Modern Concurrency Abstractions for C#. *Transactions on Programming Languages and Systems (TOPLAS)*, 26(5):769–804, 2004.

[4] A. Biboudis, N. Palladinos, G. Fourtounis, and Y. Smaragdakis. Streams à la carte: Extensible Pipelines with Object Algebras. In J. T. Boyland, editor, *29th European Conference on Object-Oriented Programming (ECOOP 2015)*, volume 37 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 591–613, Dagstuhl, Germany, 2015. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[5] B. Chandramouli, J. Goldstein, M. Barnett, R. DeLine, J. C. Platt, J. F. Terwilliger, and J. Wernsing. Trill: A High-Performance Incremental Query Processor for Diverse Analytics. *Proceedings of the VLDB Endowment*, 8(4):401–412, 2014.

[6] S. Conchon and F. Le Fessant. Jocaml: Mobile Agents for Objective-Caml. In *Proceedings of the First International Symposium on Agent Systems and Applications Third International Symposium on Mobile Agents*, ASAMA '99, pages 22–29, Washington, DC, USA, 1999. IEEE Computer Society.

[7] G. Cugola and A. Margara. TESLA: A formally defined event specification language. In *Proceedings of the Fourth ACM International Conference on Distributed Event-Based Systems*, DEBS '10, pages 50–61, New York, NY, USA, 2010. ACM.

[8] G. Cugola and A. Margara. Processing Flows of Information: From Data Stream to Complex Event Processing. *ACM Computing Surveys*, 44(3):15:1–15:62, June 2012.

[9] E. Czaplicki and S. Chong. Asynchronous Functional Reactive Programming for GUIs. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '13, pages 411–422, New York, NY, USA, 2013. ACM.

[10] A. Demers, J. Gehrke, M. Hong, M. Riedewald, and W. White. Towards Expressive Publish/Subscribe Systems. In *Proceedings of the 10th International Conference on Advances in Database Technology*, EDBT'06, pages 627–644, Berlin, Heidelberg, 2006. Springer-Verlag.

[11] P. Eugster and K. Jayaram. EventJava: An Extension of Java for Event Correlation. In S. Drossopoulou, editor, *ECOOP 2009 – Object-Oriented Programming*, volume 5653 of *Lecture Notes in Computer Science*, pages 570–594, Berlin, Heidelberg, 2009. Springer-Verlag.

[12] C. Fournet and G. Gonthier. The reflexive CHAM and the Join-calculus. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '96, pages 372–385, New York, NY, USA, 1996. ACM.

[13] P. Haller and T. Van Cutsem. Implementing Joins Using Extensible Pattern Matching. In D. Lea and G. Zavattaro, editors, *Coordination Models and Languages*, volume 5052 of *Lecture Notes in Computer Science*, pages 135–152. Springer-Verlag, Berlin, Heidelberg, 2008.

[14] I. Maier. *Reactive Programming Abstractions for Complex Event Logic and Dynamic Data Dependencies*. PhD thesis, IC, Lausanne, 2013.

[15] H. Plociniczak and S. Eisenbach. JErlang: Erlang with Joins. In D. Clarke and G. Agha, editors, *Coordination Models and Languages*, volume 6116 of *Lecture Notes in Computer Science*, pages 61–75. Springer-Verlag, Berlin, Heidelberg, 2010.

[16] G. Salvaneschi, P. Eugster, and M. Mezini. Programming with Implicit Flows. *Software, IEEE*, 31(5):52–59, Sept 2014.

[17] G. Salvaneschi, G. Hintz, and M. Mezini. REScala: Bridging Between Object-oriented and Functional Style in Reactive Applications. In *Proceedings of the 13th International Conference on Modularity*, MODULARITY '14, pages 25–36, New York, NY, USA, 2014. ACM.

[18] N. P. Schultz-Møller, M. Migliavacca, and P. Pietzuch. Distributed Complex Event Processing with Query Rewriting. In *Proceedings of the Third ACM International Conference on Distributed Event-Based Systems*, DEBS '09, pages 4:1–4:12, New York, NY, USA, 2009. ACM.

[19] A. J. Turon and C. V. Russo. Scalable Join Patterns. In *Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications*, volume 46 of *OOPSLA '11*, pages 575–594, New York, NY, USA, 2011. ACM.

[20] J. M. Van Ham, G. Salvaneschi, M. Mezini, and J. Noyé. JEScala: Modular Coordination with Declarative Events and Joins. In *Proceedings of the 13th International Conference on Modularity*, MODULARITY '14, pages 205–216, New York, NY, USA, 2014. ACM.

[21] W. White, M. Riedewald, J. Gehrke, and A. Demers. What is "Next" in Event Processing? In *Proceedings of the 26th ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, PODS '07, pages 263–272, New York, NY, USA, 2007. ACM.