Sequenced Event Set Pattern Matching

TESI PER IL DOTTORATO DI RICERCA IN INFORMATICA

DOKTORARBEIT IN INFORMATIK

PH.D. THESIS IN COMPUTER SCIENCE

Bruno Cadonna

Faculty Supervisor: Johann Gamper, Free University of Bozen-Bolzano

Approved by: Peter M. Fischer, University of Freiburg
Alessandro Artale, Free University of Bozen-Bolzano

Keywords: Event Pattern Matching, Complex Event Processing, Stream, Automaton

ACM categories: H.2.4 [Database Management Systems]: Systems – Query Processing

Contents

List of Figures

List of Tables

List of Algorithms

Acknowledgments

First of all I would like to express my sincere thanks to my supervisor Johann Gamper who convinced me to apply for the Ph.D. programme at the Free University of Bozen-Bolzano and supervised me on the research journey that followed. Johann gave me great freedom in my research and encouraged me to pursue my ideas. His office door was always open for me to discuss with him any type of problem. Johann's contributions of ideas, paper writing and administrative support were fundamental to the completion of this thesis.

I am very grateful to Michael Böhlen for showing me the importance of thorough research and the beauty of a precise and concise presentation of research results. He gave me the opportunity to spend five months in his research group, the Database Technology Group of the University of Zurich, which led to a fruitful collaboration.

My research was carried out in the context of the MEDAN project funded by the Health District South Tyrol and with the participation of the Hospital Meran-Merano. I wish to thank Manfred Mitterer and Gilbert Spizzo from the Department of Hematology and Oncology as well as Peter Huber and his collaborators from the hospital's ICT department for their continuous support in providing us real-world data and useful domain knowledge.

I am obliged to my colleagues in Bozen-Bolzano and Zurich with whom I had the luck to share this experience. Especially, I would like to thank Nikolaus Augsten with whom I spent many hours on the train commuting. I would not like to miss any conversation we had, regardless whether computer science related or not.

I am truly thankful to my parents Elisabeth Balzarek and Alfredo Cadonna who made my education possible and gave me the opportunity to broaden my horizon. I am also very grateful to my sister Cristiana Cadonna and my brother Alessandro Cadonna who always supported me with both words and deeds.

Finally, I owe the largest debt of gratitude to my fiancé Birgit Lacheiner who boosted me morally when I most needed it. Birgit gave me the best of advice I could have ever asked for and she never stopped believing in me.

Abstract

The goal of this thesis is to design, develop and evaluate new methods for event pattern matching. Event pattern matching is a query technique where an input stream of events is matched against a pattern. The output consists of matches of the pattern in the input events. Event pattern matching is widely applicable in different domains and is regarded as one of the most important building blocks for the construction of event processing applications. Current solutions for event pattern matching allow to formulate patterns that match a sequence of single events imposing one specific order. The support for matching all permutations of events is limited.

In this thesis, we introduce and formally define the sequenced event set (SES) pattern matching problem, which is the problem of matching a stream of input events against a complex pattern that specifies a sequence of sets of events rather than a sequence of single events. Events that match a set specified in the pattern can occur in any permutation, whereas events that match different sets have to follow the order of the sets in the pattern.

We present SES automata for the evaluation of SES pattern matching. A SES automaton is a nondeterministic finite state automaton enriched with a match buffer that collects matching events during execution. A pattern query is first translated to a SES automaton, and the automaton is then executed on an event stream. We conduct an analysis of the runtime complexity of the SES automaton algorithm where we consider different types of patterns. The runtime complexity is linear in the size of the event stream. Furthermore, the runtime depends on the length of the pattern and on the maximal time span of a match specified in the pattern together with the density of the events in the event stream. An experimental evaluation with real-world data shows that the SES automaton algorithm clearly outperforms a baseline approach that enumerates all permutations of single events.

To improve the performance of event pattern matching, we propose a two-phase evaluation strategy that consists of a preprocessing step followed by a pattern match-

ing step. The cheap preprocessing reduces the events that need to be processed by the expensive pattern matching step. The two-phase evaluation strategy is general enough to be applicable with SES automata as well as with other event pattern matching algorithms. Experiments with real-world data and two different event pattern matching algorithms show that the two-phase evaluation strategy significantly improves performance.

Event selection strategies allow to restrict the set of all possible matches of a pattern in an event stream to a set of matches that meets application-specific needs. The widely used skip-till-next-match event selection strategy has been used in settings where some events in the input stream are noise and should be ignored. Due to its greedy behavior, skip-till-next-match fails to identify some events as noise and consequently can miss matches that satisfy the pattern query. We propose a new event selection strategy, called robust skip-till-next-match, that improves the skipping of noise in event pattern matching and finds matches that are missed by skip-till-next-match due to its greedy behavior. We present a backtracking mechanism that extends automaton-based event pattern matching algorithms to find all matches according to robust skip-till-next-match. An extensive experimental evaluation with real-world data shows that the matches missed with skip-till-next-match can be substantial and that our backtracking solution clearly outperforms an alternative solution that finds all possible matches followed by a post processing step.

CHAPTER 1

Introduction

1.1 Event Pattern Matching

The goal of this thesis is to design, develop and evaluate new methods for event pattern matching. Event pattern matching is a query technique where an input stream of events is matched against a pattern. The output consists of matches of the pattern in the input events (see Figure 1.1).

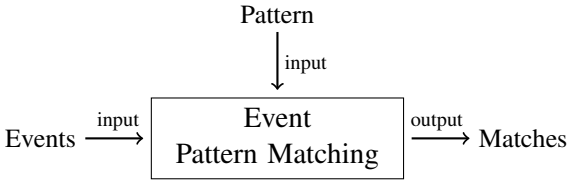


Figure 1.1: Event Pattern Matching.

An event is a data item with a timestamp that represents its occurrence time. Examples of events are the administration of a medication to a patient, the trade of shares in stock markets, or the measurement of a sensor. The events in an input stream are chronologically ordered by occurrence time. A pattern specifies constraints on chronological order, attribute values and quantification of matching events. Furthermore, a pattern limits the maximal duration of the time interval spanned by matching events. Matches contain events from the input stream that satisfy the constraints specified in the pattern.

Event pattern matching is widely applicable in different domains such as financial services [3, 4, 28, 29, 30, 31, 46, 52, 53], fraud detection [55, 61], business activity

monitoring [32], click stream analysis [32, 46, 53], RFID-based tracking and monitoring [4, 31, 43], sensor networks [6, 26], RSS feed monitoring [28], and health services [42, 44]. It is regarded as one of the most important building blocks for the construction of event processing applications [27, 35]. The increasing importance of event pattern matching in practical applications is underpinned by the availability of commercial and open source products [34, 45, 50, 54, 58] and by a recent SQL change proposal to extend SQL for pattern matching over sequences of tuples [61].

Current solutions for event pattern matching allow to formulate patterns that match a sequence of single events imposing one specific order. The support for matching all permutations of events is limited. However, some application scenarios need to partially ignore the order of events. This requirement arises from variations in the order of the input events that exist naturally in the application domain but should be ignored by the data analysis task. The need to ignore the order of events is also recognized by the aforementioned SQL change proposal that specifies a PERMUTE operator to retrieve sequences of input tuples that match any *permutation* of a set of variables specified in the pattern. Multiple PERMUTE operators in series permit to match *sequences of sets of tuples*. Only a subset of the pattern matching operators in the SQL change proposal has been implemented [30, 31, 53], and no implementation of the PERMUTE operator is known.

Event pattern matching solutions do not necessarily find the same set of matches for a pattern in an event stream. The matches may differ according to the requirements of the application the solution is made for. To satisfy the diverse needs of event processing applications various *event selection strategies* have been proposed [4] to select the desired matches from the set of all possible matches.

In this thesis, we introduce the *sequenced event set (SES) pattern matching* problem which is the problem of matching sequences of sets of events instead of sequences of single events. We present an algorithm to evaluate SES pattern queries and an evaluation strategy that increases the efficiency of the algorithm. We focus on the widely used *skip-till-next-match* event selection strategy that is applied when some events are noise and should be ignored. We show drawbacks of skip-till-next-match and propose a new event selection strategy that improves the skipping of noise in the input stream.

In the rest of this chapter, we describe SES pattern matching (Section 1.2), then we discuss different event selection strategies, next we give an overview of our contributions and publications (Section 1.4), and finally, we present the organisation of the thesis (Section 1.5).

1.2 Sequenced Event Set Pattern Matching

In this section, we describe sequenced event set (SES) pattern matching that copes with the requirement of ignoring the order of some events in a pattern. SES pattern matching is an event pattern matching problem where an input stream of events is matched against

Chemo

	PID	*L*	*V*	*U*	*T*
e_1	1	C	1672.5	mg	3 Jul
e_2	1	B	7100	1/μl	4 Jul
e_3	1	P	111.5	mg	5 Jul
e_4	2	B	10100	1/μl	6 Jul
e_5	2	P	88	mg	8 Jul
e_6	2	C	1320	mg	10 Jul
e_7	2	P	98	mg	11 Jul
e_8	1	B	3400	1/μl	17 Jul
e_9	2	B	4000	1/μl	18 Jul
e_{10}	2	B	4900	1/μl	19 Jul

(a) Events.

e_1	1	C	1672.5	mg	3 Jul
e_3	1	P	111.5	mg	5 Jul
e_8	1	B	3400	1/μl	17 Jul

e_5	2	P	88	mg	8 Jul
e_6	2	C	1320	mg	10 Jul
e_7	2	P	98	mg	11 Jul
e_9	2	B	4000	1/μl	18 Jul

(b) Matches.

Chemo with $PID = 1$

	PID	*L*	*V*	*U*	*T*
e_1	1	C	1672.5	mg	3 Jul
e_2	1	B	7100	1/μl	4 Jul
e_3	1	P	111.5	mg	5 Jul
e_8	1	B	3400	1/μl	17 Jul

Chemo with $PID = 2$

PID	*L*	*V*	*U*	*T*	
2	B	10100	1/μl	6 Jul	e_4
2	P	88	mg	8 Jul	e_5
2	C	1320	mg	10 Jul	e_6
2	P	98	mg	11 Jul	e_7
2	B	4000	1/μl	18 Jul	e_9
2	B	4900	1/μl	19 Jul	e_{10}

(c) Events Grouped by Patient.

Figure 1.2: Events of Chemotherapy Treatments.

a pattern that specifies a sequence of sets of events (SES pattern) rather than a sequence of single events. While the order of the input events that match the same set is irrelevant, i.e., any permutation of the input events is matched, the order of the input events that match distinct sets must correspond to the order of the sets in the pattern. SES patterns range from a sequence of singleton sets that forces one specific order to a single set that allows all permutations of the desired events.

Example 1.1. Analysis of Chemotherapy Data
A chemotherapy is a treatment for cancer patients that consists of a sequence of events, such as the administration of medications and laboratory examinations. Physicians at the Department of Hematology of the Hospital of Meran-Merano retrospectively analyse data recorded during chemotherapies to gain a better understanding of the effects of the treatment on patients.

Figure 1.2(a) shows a sample stream of chemotherapy events, `Chemo`. The attributes represent patient ID (PID), event type (L), value (V) with measurement unit (U), and occurrence time (T) of an event, respectively. For instance, event e_1 represents the administration of 1672.5 mg of Cyclophosphamide to patient 1 on July 3. An example

of a typical query to analyse the effect of medications on the blood is the following:

> *Within fifteen days and for each patient, find the events that match one administration of Cyclophosphamide (C), one or more administrations of Prednisone (P), followed by a single blood count measurement (B).*

Figure 1.2(b) shows answers to the query in `Chemo`.

To find the matches that answer the example query, two characteristics of the event stream and the query must be considered. First, the order of the administrations of Cyclophosphamide ($\mathtt{e}_1$, $\mathtt{e}_6$) and Prednisone ($\mathtt{e}_3$, $\mathtt{e}_5$, $\mathtt{e}_7$) differs among patient 1 and patient 2 as shown in Figure 1.2(c). This happens when treatments are modified to meet individual needs of patients. Such modifications should be ignored by the query. Second, to analyse the effect of the medications, the blood counts that occur after the medication administrations must be found. Thus, a pattern to answer the query above has to take into account variations in the chronological order of some events in the stream, and, at the same time, guarantee the succession of other events in the matches.

With an event pattern matching solution that supports only the specification of a sequence of single events, multiple patterns are needed to find the desired matches. The patterns needed to answer the example query are the following, where c, p^+, and b match one Cyclophosphamide administration (C), one or more Prednisone administration (P), and one blood count measurement (B), respectively:

$$\langle c, p^+, b\rangle$$
$$\langle p^+, c, b\rangle$$
$$\langle p^+, c, p^+, b\rangle$$

The first and the third pattern detect the matches in Figure 1.2(b). The amount of patterns for such a solution grows with the factorial of the number of events whose order should be ignored, since each permutation of the events needs to be considered.

Now, assume the query in Example 1.1 is changed to match one or more Cyclophosphamide administrations instead of only one. The patterns needed to answer the modified query are the following:

$$\langle c^+, p^+, b\rangle$$
$$\langle p^+, c^+, b\rangle$$
$$\langle p^+, c^+, p^+, b\rangle$$
$$\langle p^+, c^+, p^+, c^+, b\rangle$$
$$\vdots$$

For each pattern that ends with a Cyclophosphamide administration, another pattern is needed that ends with a Prednisone administration. Similarly, for each pattern that

Stocks

	S	P	V	T
e_1	GOOG	612	100	9:32:344
e_2	IBM	502	200	9:32:357
e_3	IBM	505	100	9:32:368
e_4	GOOG	615	400	9:32:380
e_5	IBM	511	300	9:32:396
e_6	GOOG	621	200	9:32:401

Figure 1.3: Events of Stock Trades.

ends with a Prednisone administration, another pattern is needed that ends with a Cyclophosphamide administration. Hence, the number of the patterns is only bounded by the maximal amount of events allowed in the time span specified by the query, i.e., the maximal number of treatment event that can occur in fifteen days. In summary, the creation and evaluation of multiple patterns becomes quickly infeasible.

With SES pattern matching the following pattern can be formulated to answer the query in Example 1.1:

$$\langle \{c, p^+\}, \{b\} \rangle.$$

The first set $\{c, p^+\}$ matches the events e_1, e_3 and e_5, e_6, e_7 for patient 1 and patient 2, respectively. The second set $\{b\}$ matches events e_8 and e_9.

As a second application area that requires pattern queries which cannot be easily formulated with other event pattern matching solutions we consider the analysis of stock trade data.

Example 1.2. Analysis of Stock Trade Data
Figure 1.3 shows an event stream of stock trades, `Stocks`. The attributes represent stock symbol (S), price per stock (P), volume of the trade (V), and occurrence time (T). For instance, event e_1 represents the trade of 100 Google stocks at a price of \$612 at time 9:32:344. To analyse the correlation of Google and IBM trades, the following query is issued:

> *Within a period of 100 ms, find a sequence of three Google (*GOOG*) stock trades interleaved with a sequence of three IBM (*IBM*) stock trades; both sequences of trades have strictly monotonic increasing stock prices.*

A pattern that specifies a sequence of single events is limited to express the query above as a sequence of three successions of a Google stock trade followed by an IBM stock trade. That is:

$$\langle g_1, i_1, g_2, i_2, g_3, i_3 \rangle$$

where g_j and i_j match Google and IBM stock trades, respectively, and the prices of the stocks increase with j. With such a pattern, two trades of the same stock that occur one after another are never matched. For example in event stream `Stocks` in Figure 1.3, such a pattern detects only events $\texttt{e}_1$, $\texttt{e}_2$, $\texttt{e}_4$, and $\texttt{e}_5$ and ignores $\texttt{e}_3$ and $\texttt{e}_6$. Thus, the pattern does not find any match in `Stocks` since it detects only two and not three trades for Google and IBM stocks. Alternatively, a pattern for each possible combination of Google and IBM stock trades could be used:

$$\langle g_1, i_1, g_2, i_2, g_3, i_3 \rangle$$
$$\langle g_1, i_1, g_2, i_2, i_3, g_3 \rangle$$
$$\langle g_1, i_1, i_2, g_2, g_3, i_3 \rangle$$
$$\langle g_1, i_1, i_2, g_2, i_3, g_3 \rangle$$
$$\langle i_1, g_1, g_2, i_2, g_3, i_3 \rangle$$
$$\langle i_1, g_1, g_2, i_2, i_3, g_3 \rangle$$
$$\langle i_1, g_1, i_2, g_2, g_3, i_3 \rangle$$
$$\langle i_1, g_1, i_2, g_2, i_3, g_3 \rangle$$

However, such a solution suffers from the same drawbacks as described for the analysis of chemotherapy data, i.e., the amount of patterns grows with the factorial of the number of events whose order should be ignored, thus, the evaluation of the query becomes quickly infeasible.

The SES pattern to answer the query in Example 1.2 is

$$\langle \{g_1, i_1\}, \{g_2, i_2\}, \{g_3, i_3\} \rangle.$$

The first set matches events $\texttt{e}_1, \texttt{e}_2$ in the event stream `Stocks`, the second set matches events $\texttt{e}_4$, $\texttt{e}_3$, and the third set matches events $\texttt{e}_5$, $\texttt{e}_6$.

1.3 Event Selection Strategies

An event selection strategy addresses what matches out of all possible matches of a pattern in an event stream should be returned by an event pattern matching solution. The set of all possible matches of a pattern in an event stream can be large and the differences between the matches can be minimal (see Figure 1.6). Applications might only be interested in a subset of all possible matches. To restrict the result set to matches that fulfill application-specific needs, different event selection strategies have been proposed [4] that are presented in the following.

(Partitioned) Contiguity: Events in a match must be contiguous in the input stream. All events in the input stream that occur between two events of the same match are also contained in that match. If the input stream can be conceptually partitioned according to an equality condition, contiguity can be relaxed to partitioned contiguity

Chemo with $PID = 2$

	PID	*L*	*V*	*U*	*T*
e_4	2	B	10100	1/μl	6 Jul
✓ e_5	2	P	88	mg	8 Jul
✓ e_6	2	C	1320	mg	10 Jul
✓ e_7	2	P	98	mg	11 Jul
✓ e_9	2	B	4000	1/μl	18 Jul
e_{10}	2	B	4900	1/μl	19 Jul

(a) Example of a Contiguous Match in a Partition.

Chemo

	PID	*L*	*V*	*U*	*T*
e_1	1	C	1672.5	mg	3 Jul
e_2	1	B	7100	1/μl	4 Jul
e_3	1	P	111.5	mg	5 Jul
e_4	2	B	10100	1/μl	6 Jul
✓ e_5	2	P	88	mg	8 Jul
✓ e_6	2	C	1320	mg	10 Jul
✓ e_7	2	P	98	mg	11 Jul
✗ e_8	1	B	3400	1/μl	17 Jul
✓ e_9	2	B	4000	1/μl	18 Jul
e_{10}	2	B	4900	1/μl	19 Jul

(b) Example of a Non-Contiguous Match.

e_5	2	P	88	mg	8 Jul
e_6	2	C	1320	mg	10 Jul
e_7	2	P	98	mg	11 Jul
e_9	2	B	4000	1/μl	18 Jul

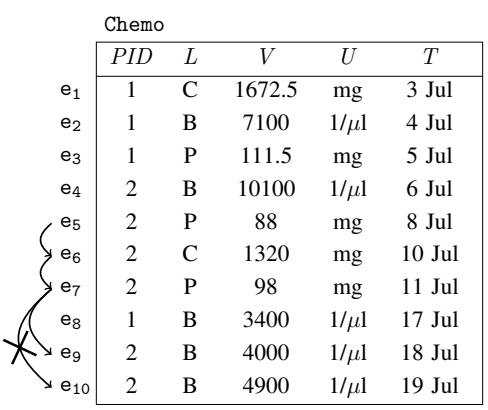
e_6	2	C	1320	mg	10 Jul
e_7	2	P	98	mg	11 Jul
e_9	2	B	4000	1/μl	18 Jul

(c) Matches in Chemo according to Partitioned Contiguity.

Figure 1.4: Contiguity and Partitioned Contiguity.

where events in a match only need to be contiguous within the same partition. Matches found with partitioned contiguity subsume matches found with contiguity. For example, Figure 1.4(a) shows a match in Chemo that conforms to partitioned contiguity. The checked events represent a match. Between events e_5 and e_9 in the partition of Chemo for patient 2, there exist only events that are part of the same match. In contrast, Chemo does not contain any match that conforms to contiguity since no match can be found with contiguous events in the unpartitioned event stream Chemo. For example, in Figure 1.4(b), between e_5 and e_9 there exist e_8 that does not satisfy the pattern together with e_5, e_6, e_7 and e_9 and makes e_7 and e_9 not contiguous events. Figure 1.4(c) shows all matches in Chemo according to partitioned contiguity.

Skip-till-next-match: Events that do not match the pattern are skipped until the next matching event is read. Skip-till-next-match is used to ignore events in the input stream that are noise to the pattern. Matches found with skip-till-next-match subsume matches found with contiguity and partitioned contiguity. Figure 1.5(a) shows the events selected for a match in Chemo according to skip-till-next-match. The arrows show the progression of the match. Events e_5, e_6, and e_7 are matched. Then, e_8 is skipped because it does not belong to patient 2. Finally, the blood count e_9 is preferred to blood count e_{10} because it is immediately next to the medication administration e_7 that was matched last. All matches in Chemo according to skip-till-next-match are shown in Figure 1.5(b).



Chemo

	PID	*L*	*V*	*U*	*T*
e_1	1	C	1672.5	mg	3 Jul
e_2	1	B	7100	1/μl	4 Jul
e_3	1	P	111.5	mg	5 Jul
e_4	2	B	10100	1/μl	6 Jul
e_5	2	P	88	mg	8 Jul
e_6	2	C	1320	mg	10 Jul
e_7	2	P	98	mg	11 Jul
e_8	1	B	3400	1/μl	17 Jul
e_9	2	B	4000	1/μl	18 Jul
e_{10}	2	B	4900	1/μl	19 Jul

(a) Selection of the Next Event.

e_1	1	C	1672.5	mg	3 Jul
e_3	1	P	111.5	mg	5 Jul
e_8	1	B	3400	1/μl	17 Jul

e_5	2	P	88	mg	8 Jul
e_6	2	C	1320	mg	10 Jul
e_7	2	P	98	mg	11 Jul
e_9	2	B	4000	1/μl	18 Jul

e_6	2	C	1320	mg	10 Jul
e_7	2	P	98	mg	11 Jul
e_9	2	B	4000	1/μl	18 Jul

(b) Matches in Chemo according to Skip-till-next-match.

Figure 1.5: Skip-till-next-match.

e_1	1	C	1672.5	mg	3 Jul
e_3	1	P	111.5	mg	5 Jul
e_8	1	B	3400	1/μl	17 Jul

e_5	2	P	88	mg	8 Jul
e_6	2	C	1320	mg	10 Jul
e_7	2	P	98	mg	11 Jul
e_9	2	B	4000	1/μl	18 Jul

e_5	2	P	88	mg	8 Jul
e_6	2	C	1320	mg	10 Jul
e_7	2	P	98	mg	11 Jul
e_{10}	2	B	4900	1/μl	19 Jul

e_5	2	P	88	mg	8 Jul
e_6	2	C	1320	mg	10 Jul
e_9	2	B	4000	1/μl	18 Jul

e_5	2	P	88	mg	8 Jul
e_6	2	C	1320	mg	10 Jul
e_{10}	2	B	4900	1/μl	19 Jul

e_6	2	C	1320	mg	10 Jul
e_7	2	P	98	mg	11 Jul
e_9	2	B	4000	1/μl	18 Jul

e_6	2	C	1320	mg	10 Jul
e_7	2	P	98	mg	11 Jul
e_{10}	2	B	4900	1/μl	19 Jul

Figure 1.6: All Possible Matches in Chemo (Skip-till-any-match).

Skip-till-any-match: All possible matches of the pattern in the event stream are found. No restrictions are applied on the result set. The result set found with skip-till-any-match subsumes matches found with (partitioned) contiguity and skip-till-next-match. Figure 1.6 shows matches in Chemo that conform to skip-till-any-match.

1.4 Contributions

Sequenced Event Set Pattern Matching. We introduce and formally define the sequenced event set pattern matching problem, which is the problem of matching a stream of input events against a complex pattern that specifies a sequence of sets of events rather than a sequence of single events. Events that match a set specified in the pattern can occur in any permutation, whereas events that match different sets have to be strictly consecutive, following the order of the sets specified in the pattern.

Automaton-based Evaluation. We present an automaton-based algorithm to evaluate SES pattern matching. The algorithm translates a pattern into a so-called SES automaton. A SES automaton is a nondeterministic finite automaton enriched with a match buffer. During execution, a SES automaton collects matching events in its match buffer. If the automaton reaches the accepting state, it outputs the match buffer as a match. We conduct a detailed complexity analysis of the automaton-based algorithm, which considers different kinds of patterns and provides upper bounds for the runtime complexity. We experimentally show that our solution clearly outperforms a brute force approach that enumerates all permutations of single events expressed by a SES pattern.

Two-phase Evaluation Strategy. We propose a general two-phase pattern matching strategy that can be combined with different event pattern matching algorithms. The two-phase strategy consists of a preprocessing step and a pattern matching step. Instead of eagerly matching incoming events, the preprocessing step buffers events in a match window to apply different pruning techniques (filtering, partitioning, and testing for necessary match conditions). In the second step, an event pattern matching algorithm is called only for match windows that satisfy the necessary match conditions. Preprocessing is relatively cheap and significantly reduces the number of events that need to be processed by a more expensive event pattern matching algorithm as well as the number of calls to the algorithm. We report the results of an empirical evaluation with an automaton-based and a join-tree-based pattern matching algorithm. For both algorithms, our two-phase evaluation strategy shows clear performance improvements.

Improve the Skipping of Noise. The widely used skip-till-next-match event selection strategy greedily matches incoming events if they satisfy the constraints in the pattern along with the events matched so far; events that violate the constraints are ignored as noise. Due to this greedy behavior, skip-till-next-match can miss matches that satisfy the pattern query. We propose a robust skip-till-next-match event selection strategy that finds matches that are missed by skip-till-next-match due to its greedy behavior. Robust skip-till-next-match considers all constraints in the query pattern together with a complete match to determine whether an input event is relevant or irrelevant. To implement the new strategy in automaton-based pattern matching algorithms, we pro-

pose a backtracking mechanism that allows to reclassify relevant events as irrelevant. Extensive experiments using real-world data show that the number of missed matches with skip-till-next-match can be quite substantial, and that our proposed backtracking mechanism outperforms an alternative solution that first produces all possible matches followed by a post processing step to filter out non-compliant matches.

Publications. The contributions presented in this thesis are published in the following articles:

- B. Cadonna, J. Gamper, and M. H. Böhlen. Sequenced event set pattern matching. In *Proceedings of the* 14^{th} *International Conference on Extending Database Technology (EDBT '11)*, pages 33–44, 2011.
- B. Cadonna, J. Gamper, and M. H. Böhlen. Efficient event pattern matching with match windows. In *Proceedings of the* 18^{th} *ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD '12)*, pages 471–479, 2012.
- B. Cadonna, J. Gamper, and M. H. Böhlen. A robust skip-till-next-match event selection strategy for event pattern matching. Submitted to 7^{th} *ACM International Conference on Distributed Event-Based Systems (DEBS '13)*.

1.5 Organization of the Thesis

The thesis is organized as follows. Chapter 2 discusses related work. Chapter 3 introduces and formally defines SES pattern matching. Chapter 4 presents the automaton-based evaluation of SES pattern matching. Chapter 5 proposes the two-phase evaluation strategy. Chapter 6 describes a robust skip-till-next-match event selection strategy that improves the skipping of noise in the input stream. Finally, Chapter 7 concludes the thesis and points to future work.

CHAPTER 2

Related Work

In this section we discuss related work. We classified related work into four groups: complex event processing (Section 2.1), publish/subscribe (Section 2.2), data stream management (Section 2.3) and miscellaneous (Section 2.4).

2.1 Complex Event Processing

Complex event processing (CEP) is a computing paradigm in which events that happened in the outside world are collected and combined to composite events according to user-defined patterns. Hence, event pattern matching is one of the main functionalities of CEP systems, and a lot of research has been done in the past. An overview of CEP can be found in [27, 35].

The SQL change proposal [61] describes an extension to SQL for pattern matching in sequences of tuples. The SQL extension is motivated by various use cases for event processing, such as security, financial, fraud detection, and RFID applications, where a tuple represents an event. The SQL extension addresses event pattern matching with the (partitioned) contiguity event selection strategy, i.e., events in a match must be contiguous in the input stream, and defines various quantifiers for events including Kleene plus. The extension defines the PERMUTE operator and its EXPAND FACTORS variant, which allow to express a set of events. Multiple PERMUTE EXPAND FACTORS operators in succession permit to express SES patterns that do not contain any Kleene plus quantifier. The SQL extension focuses on the specification of the language and provides no implementation of the various operators.

The DejaVu system [30, 31] implements in MySQL a subset of the SQL change proposal [61] for event pattern matching in live and archived data streams. Besides

event pattern matching, DejaVu addresses pattern correlation queries. Pattern correlation queries allow to join matches of a pattern in a live stream with matches of another pattern in an archived stream based on conditions over event attributes and the temporal distance of the matches. As specified in the SQL change proposal, DejaVu uses the (partitioned) contiguity event selection strategy. The PERMUTE operator is not implemented. Hence, SES patterns cannot be expressed in a simple way. Multiple patterns are required, one for each possible match. The number of possible matches grows exponentially with the cardinality of the sets of events specified in the SES pattern and becomes worse if Kleene plus quantifiers are used. Clearly, such a solution becomes quickly cumbersome and inefficient for all but very small sets of events (cf. brute force algorithm in Section 4.7.2). The evaluation of pattern matching queries is based on finite state automata. The query processor of MySQL is accordingly extended and the automata run as integral part of the query plan.

SQL-TS [52, 53] extends SQL to process complex sequential patterns in database systems. The SQL change proposal [61] mentioned above is partly based on SQL-TS. SQL-TS uses the (partitioned) contiguity event selection strategy. SQL-TS does not implement the PERMUTE operator, and SES patterns can only be expressed with multiple patterns similar to DejaVu, thus suffering from the same drawbacks. For the evaluation of sequential pattern queries, the Knuth-Morris-Pratt string matching algorithm is adapted for event pattern matching.

ZStream [46] is a cost-based query processor using join trees for matching patterns with the operators sequence, conjunction, disjunction, negation, and Kleene closure. It uses the skip-till-any-match event selection strategy, i.e., it finds all possible matches of a pattern in an input stream. Similar to a SES pattern, the conjunction operator allows to express that two events occur within a specified time window, and their order does not matter. However, unlike a SES pattern, when a conjunction is combined with a Kleene plus operator, all occurrences of the quantified event must be consecutive without any other matching event in between. For the evaluation of the pattern queries, operators are arranged in a query tree that processes events from the leaves to the root. Hashing is used to evaluate equality predicates, and a cost-based reordering algorithm is used to find an efficient order of the operators. The algorithm finds an optimal order for patterns that consist of a sequence of single events; this is not guaranteed for queries that consist of sets of more than two events since the reordering does not consider different orders of binary conjunction operators.

The pattern specification language SASE+ [41, 60] uses the nondeterministic finite state automaton NFA^{b} [4] to detect matches in a stream of events. SASE+ introduces and supports all event selection strategies described in Section 1.3. Furthermore, it allows the use of the Kleene plus quantifier. The specification of SES patterns is similar to DejaVu and suffers from the same drawbacks. Various optimization techniques are adopted by NFA^{b}. Similar automaton instances are merged to reduce space and runtime requirements. Events are partitioned to reduce the number of events processed by one automaton instance. Automata are only started if an input event matches the beginning

of the pattern. The two-phase evaluation strategy presented in Chapter 5 elaborates further on the partitioning and the conditional starting of automata.

AFA [20] is a nondeterministic finite automaton enriched with a register that holds runtime information which is accessible to the automaton transitions during the execution. AFA uses contiguity and skip-till-next-match event selection strategies. The work provides solutions for dynamic patterns, i.e., patterns that change over time, stream disorder, and revisions. AFA represents a generalization of the SES automaton presented in Chapter 4. It would be interesting future work to apply the solutions for dynamic patterns, stream disorder and revisions provided by AFA to the SES automaton.

The constraint-aware complex event processing (C-CEP) system [32] focuses on the detection of optimal points for terminating the evaluation of partial pattern query matches that will never be satisfied. For that purpose, a query unsatisfiability checking technique is applied that uses pre-existing constraints that are evaluated on the input stream at runtime. These constraints originate from the business logic. The implemented C-CEP system employs an automaton that allows to specify only SES patterns with equality relationships between events. SES patterns with more general relationships between events are not addressed. Different from this work, our pruning techniques described in the two-phase evaluation strategy in Chapter 5 need no extra constraints, but exploit the pattern query to filter out irrelevant events and to delay the instantiation of automata until some necessary conditions are satisfied.

NEEL [44] is a CEP query language for expressing nested CEP pattern queries composed of sequence, negation, conjunction and disjunction operators. Nesting conjunction operators into a sequence operator specifies a sequence of event sets. A Kleene plus quantifier is not supported. Hence, NEEL cannot express all SES patterns. The event selection strategy used in NEEL is skip-till-any-match. The evaluation of a nested pattern query consists of rewriting the pattern query, discover sharing opportunities among subexpressions in the pattern query, and finding a good execution plan.

NEXT CEP system [55] focuses on distributed event pattern matching with query rewriting. The query rewriting is applied to find equivalent patterns with lower CPU costs during evaluation. It uses a SQL-like language with six operators. An operator to specify sets of events is not included. Thus, the formulation of SES patterns is limited. NEXT CEP applies the skip-till-next-match event selection strategy. For the evaluation, the operators are translated to nondeterministic finite automata.

Akdere et al. [6] present plan-based complex event processing where input events are generated by distributed sources and a base node computes the event pattern matching. The main goal is to minimize the transmission of useless events from the sources to the base node. To achieve this goal, event acquisition and processing plans are calculated based on temporal relationships among events and event occurrence statistics. They use a pattern language that supports the specification of sequences and sets of events but does not allow sequences of sets of events. The language does not include a Kleene plus quantifier. All possible matches of a pattern (skip-till-any-match) are detected. Pattern queries are evaluated based on event detection graphs [51].

Amit [3] is an application development and runtime control tool intended to enable fast and reliable development of reactive and proactive applications. The situation manager is a monitor component of Amit that reads input events and detects matches of patterns (called situations). The pattern language provides various operators including a sequence and a conjunction operator. By nesting patterns, sequences of event sets can be specified. The Kleene plus quantifier is supported with the operator atleast. However, different from SES patterns, a Kleene plus quantifier nested in a conjunction operator only allows to match consecutive occurrences of the quantified event without any other matching event in between. The situation manager defines its own event selection strategies that partly overlap the ones used in our work.

T-REX [26] is a CEP middleware that provides the language TESLA for event pattern matching. TESLA supports a sequence operator, sets of events are expressible but a Kleene plus quantifier is missing. The event selection strategies presented partly overlap with the ones used in this thesis. The evaluation of pattern queries is based on finite automata.

RACED [25] is a distributed complex event processing middleware that implements a protocol enabling efficient and distributed detection of complex events inside a network of service brokers and dynamically adapts to network traffic. The pattern language proposed can express sequences of sets of events but the formulation is intricate because no explicit sequence operator is provided. Kleene plus quantifiers are not supported. The evaluation techniques to detect matches of pattern queries and the event selection strategies used are not addressed.

PEEX [43] is a system that detects probabilistic high-level events from RFID data. In particular it addresses data errors and ambiguity observed in event pattern matching for the RFID application domain. Its pattern language PEEXL supports a sequence operator but not a conjunction operator to express event sets. PEEX is based on a relational database management system. Pattern queries are translated to SQL.

A theoretical study about the CEDR pattern specification language [14] focuses mainly on the management of stream imperfections. CEDR does not include any operator to match sets of events.

An analysis of temporal models for CEP systems [59] presents a formal framework that is capable of describing the semantics of the sequence operator, i.e., the next event to detect, in various CEP systems. Algebraic properties of the sequence operator of temporal models of different CEP systems are discussed and desirable properties to improve performance are described.

2.2 Publish/Subscribe

With the publish/subscribe paradigm subscribers express their interests in events or patterns of events, and are notified when events are generated by publishers that match their interests. Publish/subscribe systems can be classified into topic-based and

content-based systems [36]. In topic-based systems publishers generate events under a topic and subscriber can subscribe to topics. In content-based systems a subscriber uses a subscription language to express its interests. An overview of publish/subscribe systems is available in [27, 36].

Cayuga [15, 28, 29] reads an incoming stream of events (publications) and selects event sequences according to queries that express user's interests (subscriptions). It uses an algebra that is inspired by regular expressions to formulate queries over event streams. Cayuga is unable to express all SES patterns because it can only specify relationships between consecutive events and not between arbitrary events in a set of events. Left-associated algebra expressions produce matches that conform to the skip-till-next-match event selection strategy. For the evaluation, a left-associated algebra expression is translated into a corresponding nondeterministic finite state automaton. Non-left-associated algebra expressions can be broken up into a set of left-associated ones, and each of them is then translated to a nondeterministic finite state automaton. Optimization techniques focus on the simultaneous evaluation of multiple queries using indices. By indexing automaton instances, an input event is only processed by those automaton instances that may change state. This is similar to the filter mechanism in the two-phase evaluation strategy presented in Chapter 5. To reduce the number of transitions that are evaluated for each incoming event, transitions are indexed on their condition. Cayuga instantiates an automaton for each input event that matches the beginning of the pattern query. Methods for distributing the evaluation of Cayuga's pattern queries across multiple machines in a cluster are described in [16].

Other publish/subscribe systems have limited expressiveness, since they are unable to specify a temporal order [5, 18, 37] and/or other relationships among (attributes of) events [5, 7, 18, 37].

2.3 Data Stream Management

Data stream management systems (DSMSs) provide functionalities to process data streams. Requirements for DSMSs consist of continuous queries over streams, adaptive resource management and reasonable approximations to queries due to bounded memory and real-time deadlines. Event pattern matching is outside of the scope of DSMSs, yet query languages of DSMSs allow – up to a certain extent – to express the constraints specified in patterns. An overview of challenges and models in DSMSs is available in [11, 12]. A survey of DSMSs can be found in [27].

STREAM [8, 9, 48] provides the language CQL to formulate continuous queries over streams and relations. It implements a dynamic resource management to allow high data rates and large numbers of continuous queries. SES patterns can be formulated in CQL queries by using a combination of selection, join, and aggregation operations. The formulation of a pattern becomes more and more intricate with growing length of the pattern due to the increasing number of join and selection operations that

are required to specify such sequences. CQL is unable to express SES patterns with a Kleene plus quantifier since the number of join operations must be known at query time.

Aurora [2, 17] and its further development Borealis [1] use a query algebra that contains seven primitive operations for expressing continuous and ad-hoc queries. Its dynamic resource management is driven by quality-of-service specifications supplied by applications. SES patterns can be formulated by a combination of join, selection and aggregation operations. Thus, Aurora is subject to the same limitations as STREAM regarding the formulation of SES patterns.

TelegraphCQ [21] uses an adaptive query engine based on the Eddy concept [10] to process queries. Within the query engine tuples are routed through query modules. Modules are pipelined, non-blocking versions of join, selection, projection, grouping, and aggregation operations and duplicate elimination. The routing of the tuples is continuously adapted on a tuple-by-tuple basis. Similar to STREAM, Aurora and Borealis, with TelegraphCQ, SES patterns can be formulated with join, selection and aggregation operations suffering from the same limitations.

NiagaraCQ [22] queries distributed XML data sets using a query language like XML-QL. To increase scalability, it groups continuous queries based on the observation that many queries share similar structures. Event pattern matching can be simulated by transforming the event stream into a XML document that has an element for each event. Formulating SES patterns in XML-QL for NiagaraCQ is intricate and Kleene plus quantification to match multiple occurrences of similar events is not supported.

Gigascope [23, 24] is a DSMS that focuses on network applications, including traffic analysis, intrusion detection, and performance monitoring. Its SQL-like language called GSQL provides selection, join, grouping and aggregate operators. SES patterns with Kleene plus cannot be expressed with GSQL. During evaluation, queries are translated into execution plans that can be rearranged based on operator costs.

Stream Mill [13] uses the SQL-like language ESL. ESL permits to define continuous queries that contain selection, join, union and grouping operations. Additionally, it supports the creation of user-defined aggregates (UDA) in an imperative way. The authors claim that with non-blocking UDAs ESL allows to express every non-blocking query expressible in any other possible language. Hence, in theory SES pattern can be expressed. Complexity of pattern query formulation and efficiency comparison to our work need to be investigated and we leave it for future work. For evaluation, ESL queries are compiled into query plans, the plans are optimized and executed as known from relational database management systems.

Herald-driven optimization [33] is a semantic query optimization technique applied to queries on data streams with selection and join operations. Heralds are metadata about future stream data sent with the stream. Based on such heralds a query can be continuously optimized. Heralds could be employed with the two-phase evaluation strategy presented in Chapter 5 to filter multiple input events at once and for more efficient updates of the statistics used to test the necessary match conditions.

Stream Schema [38] is a model for the specification of static metadata for data streams. Stream Schema allows to specify constraints about the structure of the items in a data stream, logical partitioning by item structure and attribute values, value relationships (e.g., order), and repetitions of subsequences in the stream. The constraints can be combined and used for query optimization during the static analysis of the queries. We leave the investigation of possible benefits of Stream Schema for SES pattern matching and the two-phase evaluation strategy described in Chapter 5 as future work.

2.4 Miscellaneous

Active databases react to events that originate from operations within the system such as data manipulations. Event languages of active databases allow to express compositions of events to which the system should react. Various event languages have been proposed [19, 39, 40, 47, 62]. They are unable to express general SES patterns because they are either very limited or unable to express relationships between attributes of events.

Event Analyzer [42] is a data warehouse component to analyze event sequences that uses skip-till-any-match event selection strategy. Its pattern specification language is able to express patterns that match simultaneous events. Simultaneous events can be interpreted as a special case of SES patterns because no total order can be imposed on the events, and hence, their order should be ignored in a query. However, the pattern specification language is not able to express more general SES patterns.

SEQ [56, 57] is a sequence database with an SQL-like query language, called SEQUIN. SEQUIN allows to select event sequences with a combination of selection, join, and aggregation operations. With the growing length of the retrieved event sequences, the queries become more and more intricate due to the increasing number of join and select operations that are required to specify such sequences. SEQUIN is unable to express SES patterns with a Kleene plus quantifier since the number of join operations must be known at query time.

CHAPTER 3

Sequenced Event Set Pattern Matching

3.1 Introduction

In this chapter, we introduce and formally define *sequenced event set* (SES) pattern matching. In SES pattern matching an input stream of events is matched against a pattern that specifies a sequence of sets of events rather than a sequence of single events. While the order of the input events that match the same set is irrelevant, i.e., any permutation of the input events is matched, the order of the input events that match distinct sets must correspond to the order of the sets in the pattern.

Example 3.1. As a running example, we consider the analysis of chemotherapy data. Figure 3.1 shows a sample event stream `Chemo`. The attributes represent patient ID (PID), event type (L), value (V) with measurement unit (U), and occurrence time (T) of an event, respectively. For instance, event $\mathtt{e}_1$ represents the administration of 1672.5 mg of Cyclophosphamide to patient 1 on July 3. To investigate the effect of the medications on the blood count the following query is issued:

> Q1: *For each patient, find the events that match (in any order) one administration of Cyclophosphamide (C), one or more administrations of Prednisone (P) with monotonically increasing values, and one administration of Doxorubicin (D), followed by a single blood count measurement (B); all events occur within fifteen days.*

The rest of this chapter is organized as follows. In Section 3.2, we formally define the concepts of event stream, pattern and match used in SES pattern matching. Section 3.3 sums up the chapter.

Chemo

	PID	*L*	*V*	*U*	*T*
e_1	1	C	1672.5	mg	3 Jul
e_2	1	B	7100	1/μl	4 Jul
e_3	1	P	111.5	mg	5 Jul
e_4	2	B	10100	1/μl	6 Jul
e_5	1	D	84	mgl	7 Jul
e_6	2	P	88	mg	8 Jul
e_7	2	D	84	mgl	9 Jul
e_8	2	C	1320	mg	10 Jul
e_9	2	P	98	mg	11 Jul
e_{10}	1	P	116.5	mg	12 Jul
e_{11}	2	P	88	mg	15 Jul
e_{12}	1	B	3400	1/μl	17 Jul
e_{13}	2	B	4000	1/μl	18 Jul
e_{14}	2	B	4900	1/μl	19 Jul
e_{15}	1	B	3000	1/μl	22 Jul

Figure 3.1: Events of Chemotherapy Treatments.

3.2 Definition of SES Pattern Matching

In this section we formalize SES pattern matching. The notation is summarized in Table 3.1.

Symbol	Description	Example
E	event stream	Chemo in Fig. 3.1
e	event	e_1 = (1, 'C', 1672.5, mg, 3 Jul)
A	attribute of e	PID with $e_1.PID = 1$
T	occurrence time of e	$e_1.T =$ 3 Jul
$\vec{e}$	chronologically ordered event sequence	$\langle e_1, e_2, e_3 \rangle$
P	pattern	$(\langle \{c, p^+, d\}, \{b\} \rangle, \Theta, 15\text{ d})$
B_i	set of variables	$\{c, p^+, d\}$
v	variable in P	c
Θ	set of constraints	$\{c.L =$ 'C', $c.PID = p.PID\}$
θ	constraint	$c.L =$ 'C'
τ	maximal time span	15 d (15 days)
γ	substitution	$\{c/\langle e_1 \rangle, p/\langle e_3 \rangle, d/\langle e_5 \rangle, b/\langle e_{12} \rangle\}$

Table 3.1: Notation.

An *event* is represented as a tuple with schema $\mathbf{E} = (A_1, \ldots, A_l, T)$, where T is a temporal attribute that stores the occurrence time of an event. For T we assume a totally ordered time domain. An *event stream*, E, is a set of events with a total order given by attribute T (see event stream Chemo in Fig. 3.1). A *chronologically ordered*

sequence of events is represented as $\vec{e} = \langle e_1, \ldots, e_n \rangle$ with e_1 and e_n referring to the first and last event in $\vec{e}$, respectively.

Definition 3.1. (*Pattern*) A *pattern*, P, is a triple

$$P = (\langle B_1, \ldots, B_k \rangle, \Theta, \tau),$$

where $\langle B_1, \ldots, B_k \rangle$, $k \geq 1$, is a sequence of pairwise disjoint sets of variables of the form v or v^+, $\Theta = \{\theta_1, \ldots, \theta_l\}$ is a set of constraints over variables in $B_1, \ldots, B_k$, and τ is a duration.

A variable, $v \in B_i$, binds a sequence of a single event, $\langle e_1 \rangle$. A quantified variable, $v^+ \in B_i$, binds a sequence of one or more events, $\langle e_1, \ldots, e_n \rangle$, $n \geq 1$. To simplify notation, we will use v to refer to both, v and v^+, when it is clear from the context. Set Θ contains constraints over variables that must be satisfied by matching events. We distinguish between properties and relationships, i.e., $\Theta = \Theta^{\mathrm{p}} \cup \Theta^{\mathrm{r}}$. *Properties*, $\theta \in \Theta^{\mathrm{p}}$, have the form $(v.A\ \phi\ C)$, where $v.A$ refers to an attribute of a matching event, C is a constant, and $\phi \in \{=, \neq, <, \leq, >, \geq, \}$ is a comparison operator. Properties can be further partitioned by variable, $\Theta^{\mathrm{p}} = \Theta^{\mathrm{p}}_{v_1} \cup \cdots \cup \Theta^{\mathrm{p}}_{v_m}$. Each set $\Theta^{\mathrm{p}}_{v_i}$ (also referred to as properties of variable v_i) contains all constraints of the form $(v_i.A\ \phi\ C)$. *Relationships*, $\theta \in \Theta^{\mathrm{r}}$, have the form $(v_i.A_k\ \phi\ v_j.A_l)$ or $(prev(v.A)\ \phi\ v.A)$, where $prev(v.A)$ refers to the previous event that has been bound to a quantified variable v^+. Finally, τ is the maximal time span within which all matching events must occur.

Example 3.2. Query Q1 is formulated as pattern $P_1 = (\langle \{c, p^+, d\}, \{b\} \rangle, \Theta, 15\ \mathrm{d})$. The first set, $B_1 = \{c, p^+, d\}$, contains three variables with the quantifier $^+$ applied to p. The second set, $B_2 = \{b\}$, contains one variable. The maximal time span is fifteen days. The constraints, separated in properties and relationships, are $\Theta = \Theta^{\mathrm{p}}_c \cup \Theta^{\mathrm{p}}_p \cup \Theta^{\mathrm{p}}_d \cup \Theta^{\mathrm{p}}_b \cup \Theta^{\mathrm{r}}$ with properties:

$$\begin{aligned}
\Theta^{\mathrm{p}}_c &= \{\ c.L = \text{'C'}\ \},\\
\Theta^{\mathrm{p}}_p &= \{\ p.L = \text{'P'}\ \},\\
\Theta^{\mathrm{p}}_d &= \{\ d.L = \text{'D'}\ \},\\
\Theta^{\mathrm{p}}_b &= \{\ b.L = \text{'B'}\ \},
\end{aligned}$$

and relationships

$$\begin{aligned}
\Theta^{\mathrm{r}} = \{\ & c.PID = p.PID,\\
& p.PID = d.PID,\\
& d.PID = b.PID,\\
& prev(p.V) < p.V\ \}.
\end{aligned}$$

Variable c binds a single administration of Cyclophosphamide ($c.L =$ 'C'), p^+ binds one or more administrations of Prednisone ($p.L =$ 'P'), d binds a single administration

of Doxorubicin ($d.L$ = 'D'), and b binds a single blood count measurement ($b.L$ = 'B'). The first three relationships force the matched events to refer to the same patient. The fourth relationship requires the amount of Prednisone to be increasing.

To define the matching of a pattern P and an event stream E, we use a *substitution* $\gamma = \{v_1/\vec{e}_1, \ldots, v_m/\vec{e}_m\}$. Each pair $v/\vec{e}$ represents a *binding* of variable v to a sequence, $\vec{e} = \langle e_1, \ldots, e_n \rangle, n \geq 1$, of events in E. A substitution contains exactly one binding for each variable in P, and an event can appear in at most one of the bindings. For a constraint $\theta \in \Theta$, $\theta\gamma$ denotes the *instantiation* of θ by γ and is obtained from θ by simultaneously replacing all variables v_i by the corresponding event sequence $\vec{e}_i$. The instantiation of a set of constraints Θ is $\Theta\gamma = \{\theta_1\gamma, \ldots, \theta_l\gamma\}$. The truth value of an instantiation is defined through an *interpretation*, $I(\Theta\gamma)$, as follows:

- $I(\Theta\gamma) \equiv I(\{\theta_1\gamma, \ldots, \theta_l\gamma\}) \equiv I(\theta_1\gamma) \wedge \cdots \wedge I(\theta_l\gamma)$,
- $I(\vec{e}.A \,\phi\, C) \equiv \forall x \in \vec{e}\,(\; x.A \,\phi\, C\;)$,
- $I(\vec{e}_i.A_i \,\phi\, \vec{e}_j.A_j) \equiv \forall x \in \vec{e}_i, y \in \vec{e}_j\,(\; x.A_i \,\phi\, y.A_j\;)$,
- $I(prev(\vec{e}.A) \,\phi\, \vec{e}.A) \equiv \begin{cases} \forall x_{i-1}, x_i \in \vec{e}\,(\; x_{i-1}.A \,\phi\, x_i.A\;) & \text{if } |\vec{e}| \geq 2, \\ \text{true} & \text{otherwise.} \end{cases}$

The interpretation of $\Theta\gamma$ is the conjunction of the interpretation of the individual constraints $\theta_i\gamma$. If θ is a property, i.e., has the form $(v.A\,\phi\,C)$, attribute A of each event that is bound to v is compared to the constant value C. If θ is a relationship of the form $(v_i.A_i \,\phi\, v_j.A_j)$, all combinations of attributes A_i and A_j of all events bound to v_i and v_j, respectively, are compared. If θ is a relationship of the form $(prev(v.A) \,\phi\, v.A)$, the attribute A of each pair of consecutive events that are bound to v are compared.

Example 3.3. Let Θ be the set of constraints of pattern $P_1 = (\langle\{c, p^+, d\}, \{b\}\rangle, \Theta, 15\text{ d})$ and $\gamma = \{c/\langle \mathsf{e}_8\rangle,\ p/\langle \mathsf{e}_6, \mathsf{e}_9\rangle,\ d/\langle \mathsf{e}_7\rangle,\ b/\langle \mathsf{e}_{13}\rangle\}$ be a substitution. The instantiation of Θ (cf. Example 3.2) with γ is

$$\begin{aligned}\Theta\gamma = \{&\langle \mathsf{e}_8\rangle.L = \text{'C'}, \langle \mathsf{e}_6, \mathsf{e}_9\rangle.L = \text{'P'}, \langle \mathsf{e}_7\rangle.L = \text{'D'},\\ &\langle \mathsf{e}_{13}\rangle.L = \text{'B'},\\ &\langle \mathsf{e}_8\rangle.PID = \langle \mathsf{e}_6, \mathsf{e}_9\rangle.PID, \langle \mathsf{e}_6, \mathsf{e}_9\rangle.PID = \langle \mathsf{e}_7\rangle.PID,\\ &\langle \mathsf{e}_7\rangle.PID = \langle \mathsf{e}_{13}\rangle.PID,\\ &prev(\langle \mathsf{e}_6, \mathsf{e}_9\rangle.V) < \langle \mathsf{e}_6, \mathsf{e}_9\rangle.V\}.\end{aligned}$$

The interpretation of $\Theta\gamma$ is

$$
\begin{aligned}
I(\Theta\gamma) \equiv\ & I(\langle \mathtt{e}_8\rangle.L = \text{'C'}) \wedge I(\langle \mathtt{e}_6, \mathtt{e}_9\rangle.L = \text{'P'}) \wedge I(\langle \mathtt{e}_7\rangle.L = \text{'D'}) \wedge \\
& I(\langle \mathtt{e}_{13}\rangle.L = \text{'B'}) \wedge \\
& I(\langle \mathtt{e}_8\rangle.PID = \langle \mathtt{e}_6, \mathtt{e}_9\rangle.PID) \wedge I(\langle \mathtt{e}_6, \mathtt{e}_9\rangle.PID = \langle \mathtt{e}_7\rangle.PID) \wedge \\
& I(\langle \mathtt{e}_7\rangle.PID = \langle \mathtt{e}_{13}\rangle.PID) \wedge \\
& I(prev(\langle \mathtt{e}_6, \mathtt{e}_9\rangle.V) < \langle \mathtt{e}_6, \mathtt{e}_9\rangle.V) \\
\equiv\ & \mathtt{e}_8.L = \text{'C'} \wedge \mathtt{e}_6.L = \text{'P'} \ \wedge \mathtt{e}_9.L = \text{'P'} \wedge \mathtt{e}_7.L = \text{'D'} \wedge \\
& \mathtt{e}_{13}.L = \text{'B'} \wedge \\
& \mathtt{e}_8.PID = \mathtt{e}_6.PID \wedge \mathtt{e}_8.PID = \mathtt{e}_9.PID \wedge \mathtt{e}_6.PID = \mathtt{e}_7.PID \wedge \\
& \mathtt{e}_9.PID = \mathtt{e}_7.PID \wedge \mathtt{e}_7.PID = \mathtt{e}_{13}.PID \wedge \\
& \mathtt{e}_6.V < \mathtt{e}_9.V \\
\equiv\ & \text{'C'} = \text{'C'} \wedge \text{'P'} = \text{'P'} \wedge \text{'P'} = \text{'P'} \wedge \text{'D'} = \text{'D'} \wedge \text{'B} = \text{'B} \wedge \\
& 2 = 2 \wedge 2 = 2 \wedge 2 = 2 \wedge 2 = 2 \wedge 2 = 2 \wedge \\
& 88 < 98 \\
\equiv\ & \text{true}
\end{aligned}
$$

Definition 3.2. (*Match*) Let $P = (\langle B_1, \ldots, B_k\rangle, \Theta, \tau)$ be a pattern and E be an event stream. A substitution $\gamma = \{v_1/\vec{e}_1, \ldots, v_m/\vec{e}_m\}$ is a *match* of P in E if and only if the following holds:

$$I(\Theta\gamma) \text{ is true,} \tag{3.1}$$

$$\forall v_i/\vec{e}_i, v_j/\vec{e}_j \in \gamma\ (\ v_i \in B_i \wedge v_j \in B_{i+1} \rightarrow e_{i_n}.T < e_{j_1}.T\ ), \tag{3.2}$$

$$\forall v_i/\vec{e}_i, v_j/\vec{e}_j \in \gamma\ (\ |e_{i_1}.T - e_{j_n}.T| \leq \tau\ ). \tag{3.3}$$

Condition 3.1 requires that a match satisfies all constraints in Θ. Condition 3.2 ensures that all events in a match that are bound to a variable in B_i must occur before all events that are bound to any variable in B_{i+1}. No order is imposed on events that are bound to variables in the same B_i, hence any permutation is matched. Condition 3.3 constrains all events in a match to occur within a time span of τ.

Example 3.4. Figure 3.2 illustrates a match of P_1 in relation `Chemo`. Each of the variables c, d, and b binds a single event, whereas p binds a sequence of two events. The instantiation $\Theta\gamma$ is satisfied (Condition 3.1): $\mathtt{e}_8$ is a Cyclophosphamide administration ('C'), $\mathtt{e}_6$ and $\mathtt{e}_9$ are Prednisone administrations ('P'), $\mathtt{e}_7$ is a Doxorubicin administration ('D') and $\mathtt{e}_{13}$ is a blood count measurement ('B'); all events refer to the same patient; and the value of $\mathtt{e}_6$ is less than the value of $\mathtt{e}_9$. Events $\mathtt{e}_6$, $\mathtt{e}_7$, $\mathtt{e}_8$, and $\mathtt{e}_9$ that match the first set in P_1 occur before $\mathtt{e}_{13}$ that matches the second set in P_1 (Condition 3.2). The time span between the first ($\mathtt{e}_6$) and last event ($\mathtt{e}_{13}$) is less than fifteen

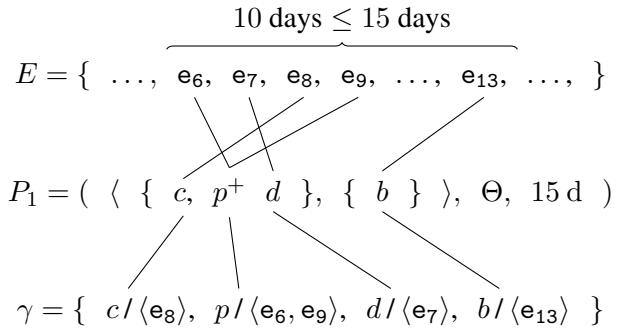


Figure 3.2: Match for Pattern P_1.

days (Condition 3.3). The complete list of matches is:

$$
\begin{array}{llll}
\{\, c/\langle \mathsf{e}_1\rangle, & p/\langle \mathsf{e}_3\rangle, & d/\langle \mathsf{e}_5\rangle, & b/\langle \mathsf{e}_{12}\rangle \,\},\\
\{\, c/\langle \mathsf{e}_1\rangle, & p/\langle \mathsf{e}_{10}\rangle, & d/\langle \mathsf{e}_5\rangle, & b/\langle \mathsf{e}_{12}\rangle \,\},\\
\{\, c/\langle \mathsf{e}_1\rangle, & p/\langle \mathsf{e}_3, \mathsf{e}_{10}\rangle, & d/\langle \mathsf{e}_5\rangle, & b/\langle \mathsf{e}_{12}\rangle \,\},\\
\{\, c/\langle \mathsf{e}_8\rangle, & p/\langle \mathsf{e}_6\rangle, & d/\langle \mathsf{e}_7\rangle, & b/\langle \mathsf{e}_{13}\rangle \,\},\\
\{\, c/\langle \mathsf{e}_8\rangle, & p/\langle \mathsf{e}_6\rangle, & d/\langle \mathsf{e}_7\rangle, & b/\langle \mathsf{e}_{14}\rangle \,\},\\
\{\, c/\langle \mathsf{e}_8\rangle, & p/\langle \mathsf{e}_9\rangle, & d/\langle \mathsf{e}_7\rangle, & b/\langle \mathsf{e}_{13}\rangle \,\},\\
\{\, c/\langle \mathsf{e}_8\rangle, & p/\langle \mathsf{e}_9\rangle, & d/\langle \mathsf{e}_7\rangle, & b/\langle \mathsf{e}_{14}\rangle \,\},\\
\{\, c/\langle \mathsf{e}_8\rangle, & p/\langle \mathsf{e}_{11}\rangle, & d/\langle \mathsf{e}_7\rangle, & b/\langle \mathsf{e}_{13}\rangle \,\},\\
\{\, c/\langle \mathsf{e}_8\rangle, & p/\langle \mathsf{e}_{11}\rangle, & d/\langle \mathsf{e}_7\rangle, & b/\langle \mathsf{e}_{14}\rangle \,\},\\
\{\, c/\langle \mathsf{e}_8\rangle, & p/\langle \mathsf{e}_6, \mathsf{e}_9\rangle, & d/\langle \mathsf{e}_7\rangle, & b/\langle \mathsf{e}_{13}\rangle \,\},\\
\{\, c/\langle \mathsf{e}_8\rangle, & p/\langle \mathsf{e}_6, \mathsf{e}_9\rangle, & d/\langle \mathsf{e}_7\rangle, & b/\langle \mathsf{e}_{14}\rangle \,\}.
\end{array}
$$

Definition 3.2 specifies the set of all possible matches of P in E, but applications might be interested in a subset only. This can be controlled by different *event selection strategies* [4]. In this thesis, we use the skip-till-next-match event selection strategy and leave the adaptation of presented techniques to other event selection strategies for future work. Skip-till-next-match greedily matches incoming events if they satisfy the constraints in the pattern along with the events matched so far; events that violate the constraints are skipped as noise.

Example 3.5. Consider $P_1 = (\langle\{c, p^+, d\}, \{b\}\rangle, \Theta, 15\text{ d})$ and match γ in Figure 3.2. Match γ conforms to skip-till-next-match because all events in γ are greedily matched and satisfy the constraints in the pattern along with the events matched earlier. Event e_6 starts the match because it matches variable p in the first set, $\{c, p^+, d\}$, of P_1. Events e_7, e_8 and e_9 are the first after e_6 that match c, d, and again p in $\{c, p^+, d\}$. Then, e_{10}, e_{11}, and e_{12} are skipped since they do not satisfy the constraints specified by P_1. Finally, e_{13} is the first event after e_9 that satisfies b in $\{b\}$. In contrast, match $\{c/\langle \mathsf{e}_8\rangle, p/\langle \mathsf{e}_6, \mathsf{e}_9\rangle, d/\langle \mathsf{e}_7\rangle, b/\langle \mathsf{e}_{14}\rangle\}$ does not conform to skip-till-next-match, since

$\mathtt{e}_{14}$ bound to b occurs after $\mathtt{e}_{13}$ that also satisfies the constraints in P_1 along with the events matched earlier. The complete list of matches of P_1 in E that conform to skip-till-next-match is:

$$\begin{array}{llll} \{\ c/\langle \mathtt{e}_1 \rangle, & p/\langle \mathtt{e}_3, \mathtt{e}_{10} \rangle, & d/\langle \mathtt{e}_5 \rangle, & b/\langle \mathtt{e}_{12} \rangle\ \}, \\ \{\ c/\langle \mathtt{e}_8 \rangle, & p/\langle \mathtt{e}_9 \rangle, & d/\langle \mathtt{e}_7 \rangle, & b/\langle \mathtt{e}_{13} \rangle\ \}, \\ \{\ c/\langle \mathtt{e}_8 \rangle, & p/\langle \mathtt{e}_6, \mathtt{e}_9 \rangle, & d/\langle \mathtt{e}_7 \rangle, & b/\langle \mathtt{e}_{13} \rangle\ \}. \end{array}$$

3.3 Summary

In this chapter, we introduced and formally defined SES pattern matching. A pattern consists of a sequence of sets of variables, a set of constraints and a duration. A variable matches a single event and a variable that is quantified with a plus matches one or more events. Events that match variables in the same set can occur in any permutation, whereas events that match variables from different sets must occur in the order of the sets in the pattern. The set of constraints contains conditions over the variables that must be satisfied by matching events. The duration determines the maximal time span within which all matching event must occur. A match is a set of bindings that binds variables in the pattern to chronologically ordered sequences of events from the input stream. A match must satisfy all constraints specified in the pattern. In this thesis, we focus on the skip-till-next-match event selection strategy.

CHAPTER 4

Automaton-based Evaluation

4.1 Introduction

In this chapter we present an automaton-based algorithm for the evaluation of pattern queries in SES pattern matching. The algorithm first translates a query pattern into a so-called SES automaton, which is then executed on the input event stream. We analyse the runtime complexity of the automaton-based algorithm considering different kinds of nondeterminism. Finally, we experimentally evaluate the algorithm.

Example 4.1. As a running example for this chapter, we reuse the event stream `Chemo` and the query Q1 from the previous chapter which we replicate in Figure 4.1 and below for the sake of convenience.

> Q1: *For each patient, find the events that match (in any order) one administration of Cyclophosphamide (C), one or more administrations of Prednisone (P) with monotonically increasing values, and one administration of Doxorubicin (D), followed by a single blood count measurement (B); all events occur within fifteen days.*

The corresponding pattern for Query Q1 is

$$P_1 = (\langle \{c, p^+, d\}, \{b\} \rangle, \Theta, 15\ \mathrm{d}).$$

The rest of this chapter is organized as follows. Section 4.2 defines the SES automaton. In Section 4.3, we describe the construction of a SES automaton from a pattern. The execution of a SES automaton is presented in Section 4.4. Section 4.5 provides the algorithm for the evaluation of event pattern matching with SES automata.

Chemo

	PID	*L*	*V*	*U*	*T*
e_1	1	C	1672.5	mg	3 Jul
e_2	1	B	7100	l/μl	4 Jul
e_3	1	P	111.5	mg	5 Jul
e_4	2	B	10100	l/μl	6 Jul
e_5	1	D	84	mgl	7 Jul
e_6	2	P	88	mg	8 Jul
e_7	2	D	84	mgl	9 Jul
e_8	2	C	1320	mg	10 Jul
e_9	2	P	98	mg	11 Jul
e_{10}	1	P	116.5	mg	12 Jul
e_{11}	2	P	88	mg	15 Jul
e_{12}	1	B	3400	l/μl	17 Jul
e_{13}	2	B	4000	l/μl	18 Jul
e_{14}	2	B	4900	l/μl	19 Jul
e_{15}	1	B	3000	l/μl	22 Jul

Figure 4.1: Events of Chemotherapy Treatments.

In Section 4.6, we analyse the runtime complexity of the algorithm. Section 4.7 shows the result of an experimental evaluation with real-world data. The chapter ends with a summary in Section 4.8.

4.2 Definition of SES Automaton

A SES automaton is a nondeterministic finite state automaton enriched with a match buffer, β, which collects bindings during the execution of the automaton.

Definition 4.1. (*SES Automaton*) Let $P = (\langle B_1, \ldots, B_k \rangle, \Theta, \tau)$ be a pattern with variables $V = \{v_1, \ldots, v_m\}$ and $\mathbf{E}$ be a schema of events. A *SES automaton*, N, is a six tuple

$$N = (V, Q, \Sigma, \delta, q_s, q_f),$$

where

- V is the set of variables in P,
- $Q = \{q_1, \ldots, q_l\}$, $q_i \subseteq V$, is a finite set of states,
- Σ is the set of all possible events with schema $\mathbf{E}$,
- δ is a transition function $\delta : Q \times \Sigma \rightarrow V^*$,
- $q_s \in Q$ is the start state, and

- $q_f \in Q$ is the accepting state.

Each state, $q \in Q$, is defined as the subset of the variables in P that have been matched so far. The transition function δ takes as arguments a state q and an event e, and returns a finite set of variables $\{v_1, \ldots, v_n\}$. If the output contains exactly one variable v, the automaton changes to state $q \cup \{v\}$ and adds the binding of v and input event e to match buffer β. If the output contains multiple variables $\{v_1, \ldots, v_n\}$, nondeterminism occurs. The automaton branches into n automata and for each variable v_i a distinct automaton changes to state $q \cup \{v_i\}$ and adds the binding of v_i and input event e to its match buffer β. Transition function δ uses a transition condition Θ_δ to decide if a variable is contained in the output. Transition condition Θ_δ contains constraints of the form $(v.A\ \phi\ C)$, $(v_i.A_k\ \phi\ v_j.A_l)$ and $(prev(v.A)\ \phi\ v.A)$ over variables in P. The execution of an automaton begins in the start state, $q_\mathrm{s} = \emptyset$. The accepting state, q_f, marks the acceptance of the bindings in β as a match.

To simplify notation, we redefine a SES automaton to be a four tuple $N = (Q, \Delta, q_\mathrm{s}, q_f)$, where $\Delta = \{\delta_1, \ldots, \delta_n\}$ is a finite set of transitions of the form $\delta = (q, v, \Theta_\delta)$. A transition, $\delta = (q, v, \Theta_\delta) \in \Delta$, leads from a source state, $q \in Q$, to a target state, $q \cup \{v\} \in Q$, if the transition condition, Θ_δ, is satisfied by the current input event; the input event is bound to variable v and added to match buffer β.

A SES automaton can be graphically represented as a graph. Nodes represent states, and edges represent transitions. An edge is labeled with the variable that is bound by the transition and the corresponding transition condition. The start state is marked with an incoming arrow, the accepting state is doubly circled.

Example 4.2. Figure 4.2 shows the SES automaton for the pattern $(\langle\{b\}\rangle, \Theta, 15\ \mathrm{d})$ with $\Theta = \{b.L = \text{'B'}\}$, which is the second set B_2 (in isolation) of pattern P_1 in Example 4.1. The corresponding automaton is

$$(\{\emptyset, \{b\}\}, \Delta, \emptyset, \{b\}).$$

It has two states, the start state $\emptyset$ and the accepting state $\{b\}$. To facilitate reading, in the graphical illustration we denote states by the concatenation of the corresponding variables, e.g., the node labeled with b represents state $\{b\}$. There is a single transition,

$$\Delta = \{(\emptyset, b, \{b.L = \text{'B'}\})\},$$

with a condition that constrains variable b to match a blood count measurement. Notice that no other conditions are involved, since B_2 is considered in isolation.

4.3 Construction of a SES Automaton

The construction of a SES automaton for a pattern, P, is a two-step process:

1. each individual set in P is translated into a SES automaton and

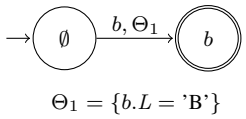


Figure 4.2: SES Automaton for Pattern $(\langle\{b\}\rangle, \Theta, 15\ \mathrm{d})$.

2. the individual automata from step 1 are concatenated according to the order of the sets in P.

4.3.1 Translation of a Single Set in the Pattern

Let $P = (\langle B_1, \ldots, B_k\rangle, \Theta, \tau)$ be a pattern, and consider a single set B_i with variables $V_i = \{v_1, \ldots, v_m\}$ in P. For each subset of V_i, the corresponding SES automaton contains a state, yielding $Q = \{q : q \in \mathcal{P}(V_i)\}$ as the set of states. For each state, $q \in Q$, and single variable, $v \in V_i \setminus q$, a transition $\delta = (q, v, \Theta_\delta)$ is built with the following constraints:

$$\Theta_\delta = \{\theta \in \Theta : (\theta \equiv v.A\ \phi\ C) \vee (\theta \equiv v.A_k\ \phi\ w.A_l \wedge w \in B_1 \cup \cdots \cup B_{i-1} \cup (q \cup \{v\}))\}.$$

That is, Θ_δ is defined as the set of all conditions in Θ that constrain events bound to variable v (of transition δ) either with respect to a constant value C or with respect to events that are bound to other variables from the current state or from preceding sets in P. For each state, q, and quantified variable, $v^+ \in B_i$ and $v \in q$, a transition $\delta = (q, v, \Theta_\delta)$ is created, which loops at state q since $q \cup \{v\} = q$ for $v \in q$. The transition condition Θ_δ is constructed in a similar way as above, but additionally it includes conditions of the form $prev(v.A)\ \phi\ v.A$.

Example 4.3. Figure 4.3 shows the SES automata N_1 and N_2 for the sets $\{c, p^+, d\}$ and $\{b\}$ in the example pattern P_1, respectively. The automaton for $\{c, p^+, d\}$ is

$$N_1 = (Q_1, \Delta_1, \emptyset, \{c, p, d\}),$$

where

$$Q_1 = \{\ \emptyset,\ \{c\},\ \{d\},\ \{p\},\ \{c, d\},\ \{c, p\},\ \{d, p\},\ \{c, p, d\}\ \},$$

and

$$\begin{aligned}\Delta_1 = \{\ &(\emptyset, c, \Theta_1),\ (\emptyset, d, \Theta_2),\ (\emptyset, p, \Theta_3),\\ &(\{c\}, d, \Theta_4),\ (\{c\}, p, \Theta_5),\\ &(\{d\}, c, \Theta_6),\ (\{d\}, p, \Theta_7),\\ &(\{p\}, c, \Theta_8),\ (\{p\}, d, \Theta_9),\ (\{p\}, p, \Theta_{10}),\\ &(\{c, d\}, p, \Theta_{11}),\\ &(\{c, p\}, d, \Theta_{12}),\ (\{c, p\}, p, \Theta_{13}),\\ &(\{d, p\}, c, \Theta_{14}),\ (\{d, p\}, p, \Theta_{15}),\\ &(\{c, p, d\}, p, \Theta_{16})\ \}.\end{aligned}$$

The automaton for $\{b\}$ is

$$N_2 = (Q_2, \Delta_2, \emptyset, \{b\})$$

where

$$Q_2 = \{\ \emptyset, \{b\}\ \}$$

and

$$\Delta_2 = \{\ (\emptyset, b, \Theta_{17})\ \}.$$

The set of states corresponds to all possible subsets of the variables in the sets of P_1. Notice the looping transitions in all states that include the quantified variable p^+. The different paths through the individual automata correspond to all possible permutations of input events that are matched by the corresponding set.

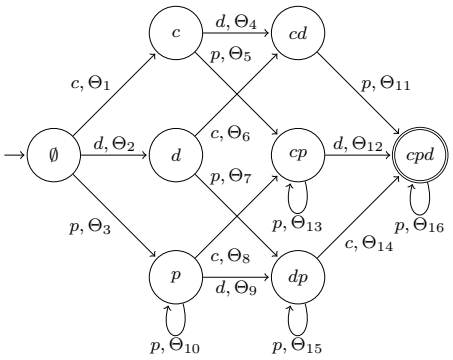
4.3.2 Concatenation of SES Automata

The second step is to concatenate the individual automata $N_1, \ldots, N_k$ from step 1. For two consecutive automata N_i, N_{i+1}, this means essentially to merge the accepting state of N_i with the start state of N_{i+1} plus doing some renaming.

More formally, let $N_1 = (Q_1, \Delta_1, q_{s_1}, q_{f_1})$ and $N_2 = (Q_2, \Delta_2, q_{s_2}, q_{f_2})$ be the automata for the sets B_1 and B_2, respectively. The concatenation of N_1 and N_2 yields a SES automaton $N = (Q_1 \cup Q_2^*, \Delta_1 \cup \Delta_2^*, q_{s_1}, q_{f_2} \cup V_1)$, which is constructed as follows. The states in Q_2 are renamed to include the set of variables V_1 of the set B_1, i.e., $Q_2^* = \{q \cup V_1 : q \in Q_2\}$. The states in the transitions are renamed in an analogous way, i.e., for each transition $(q, v, \Theta_\delta) \in \Delta_2$ we have a transition $(q \cup V_1, v, \Theta_\delta)$ in Δ_2^*.

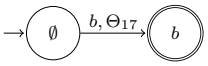
The concatenation of a sequence of automata, $N_1, \ldots, N_n$, is done in the same order as the corresponding sets, $B_1, \ldots, B_n$, specified in the pattern. N_1 and N_2 are concatenated into an intermediate automaton $N' = N_1 N_2$, which is then concatenated with N_3 to give $N'' = (N_1 N_2) N_3$, and so on.

Example 4.4. The SES automaton in Figure 4.4 is the result of concatenating N_1 and N_2 from Figure 4.3, and it corresponds to pattern P_1 in the running example. The



$\Theta_1 = \{\ c.L = \text{'C'}\ \}$
$\Theta_2 = \{\ d.L = \text{'D'}\ \}$
$\Theta_3 = \{\ p.L = \text{'P'}\ \}$
$\Theta_4 = \{\ d.L = \text{'D'},\ c.PID = d.PID\ \}$
$\Theta_5 = \{\ p.L = \text{'P'},\ c.PID = p.PID\ \}$
$\Theta_6 = \{\ c.L = \text{'C'},\ c.PID = d.PID\ \}$
$\Theta_7 = \{\ p.L = \text{'P'}\ \}$
$\Theta_8 = \{\ c.L = \text{'C'},\ c.PID = p.PID\ \}$
$\Theta_9 = \{\ d.L = \text{'D'},\ c.PID = d.PID\ \}$
$\Theta_{10} = \{\ p.L = \text{'P'},\ prev(p.V) < p.V\ \}$
$\Theta_{11} = \{\ p.L = \text{'P'},\ c.PID = p.PID\ \}$
$\Theta_{12} = \{\ d.L = \text{'D'},\ c.PID = d.PID\ \}$
$\Theta_{13} = \{\ p.L = \text{'P'},\ c.PID = p.PID,\ prev(p.V) < p.V\ \}$
$\Theta_{14} = \{\ c.L = \text{'C'},\ c.PID = d.PID,\ c.PID = p.PID\ \}$
$\Theta_{15} = \{\ p.L = \text{'P'},\ prev(p.V) < p.V\ \}$
$\Theta_{16} = \{\ p.L = \text{'P'},\ c.PID = p.PID,\ prev(p.V) < p.V\ \}$

(a) N_1 for $\{c, p^+, d\}$



$\Theta_{17} = \{\ b.L = \text{'B'},\ d.PID = b.PID\ \}$

(b) N_2 for $\{b\}$

Figure 4.3: SES Automata for Single Sets in $P_1 = (\langle\{c, p^+, d\}, \{b\}\rangle, \Theta, 15\ \mathrm{d})$.

complete automaton is specified as

$$N = (Q, \Delta, \emptyset, \{c, p, d, b\})$$

where

$$Q = \{\ \emptyset,\ \{c\},\ \{d\},\ \{p\},\ \{c, d\},\ \{c, p\},\ \{d, p\},\ \{c, p, d\},\ \{c, p, d, b\}\ \},$$

and

$$\begin{aligned}\Delta = \{\ & (\emptyset, c, \Theta_1),\ (\emptyset, d, \Theta_2),\ (\emptyset, p, \Theta_3),\\ & (\{c\}, d, \Theta_4),\ (\{c\}, p, \Theta_5),\\ & (\{d\}, c, \Theta_6),\ (\{d\}, p, \Theta_7),\\ & (\{p\}, c, \Theta_8),\ (\{p\}, d, \Theta_9),\ (\{p\}, p, \Theta_{10}),\\ & (\{c, d\}, p, \Theta_{11}),\\ & (\{c, p\}, d, \Theta_{12}),\ (\{c, p\}, p, \Theta_{13}),\\ & (\{d, p\}, c, \Theta_{14}),\ (\{d, p\}, p, \Theta_{15}),\\ & (\{c, p, d\}, p, \Theta_{16}),\ (\{c, p, d\}, b, \Theta_{17})\ \}.\end{aligned}$$

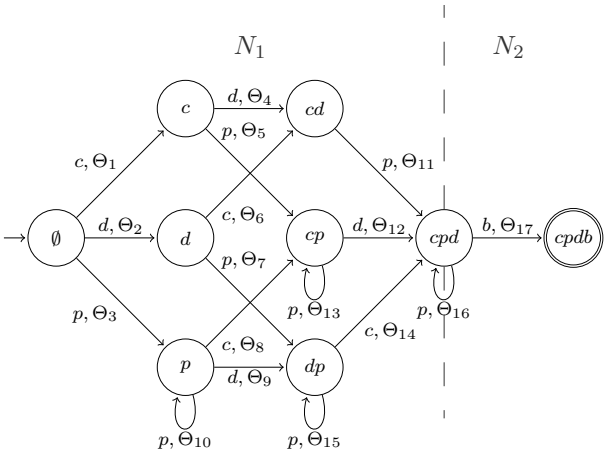


Figure 4.4: SES Automaton for $P_1 = (\langle\{c, p^+, d\}, \{b\}\rangle, \Theta, 15\text{ d})$.

The accepting state $\{c, p, d, b\}$ originates from state $\{b\}$ in N_2 extended by $\{c, p, d\}$, and the state $\{c, p, d\}$ originates from the "merging" of the accepting state of N_1 and the start state of N_2. The conditions, Θ_i, are identical to the conditions in Figure 4.3.

4.4 Execution of a SES Automaton

The execution of a SES automaton consists of reading events from the input, taking the transitions whose conditions are satisfied by an input event, and adding the input events that trigger a transition to the match buffer. To describe the execution of a SES automaton in more detail, we first define the concept of an automaton instance.

Definition 4.2. (*Automaton Instance*) An *automaton instance*, $\tilde{n}$, describes a SES automaton N during execution and is defined as a pair

$$\tilde{n} = (q_\text{c}, \beta),$$

where $q_\text{c} \in Q$ is the state $\tilde{n}$ is currently in and β is the corresponding match buffer.

An automaton instance, $\tilde{n}$, of a SES automaton $N = (Q, \Delta, q_\text{s}, q_f)$ begins in the start state q_s. It reads events from the input and tests the conditions, Θ_δ, of the transitions that leave the current state q_c. If an input event satisfies the condition of a transition, $\tilde{n}$ moves to the target state of the transition and adds a binding consisting of the variable of the transition and the input event to match buffer β. If an input event satisfies the condition of multiple transitions, nondeterminism occurs, and $\tilde{n}$ branches into multiple automaton instances. If no transition conditions are satisfied and the automaton instance is not in the start state, the input event is ignored and the automaton remains in its current state. When all input events are read, the automaton instances

that reached the accepting state, q_f, contain a match in their match buffer β. To ensure that the time interval spanned by events in a match does not exceed duration τ, only a subsequence of input stream E, called a *match window*, is passed to an automaton instance as input.

Definition 4.3. (*Match Window*) Let E be an event stream and τ be a duration specified in a pattern. The *match window* starting at event $e_i \in E$ is defined as

$$W = \langle e \in E : 0 \leq e.T - e_i.T \leq \tau \rangle.$$

A match window, W, is a maximal subsequence of E that starts at an event e_i and includes all subsequent events that are within distance τ.

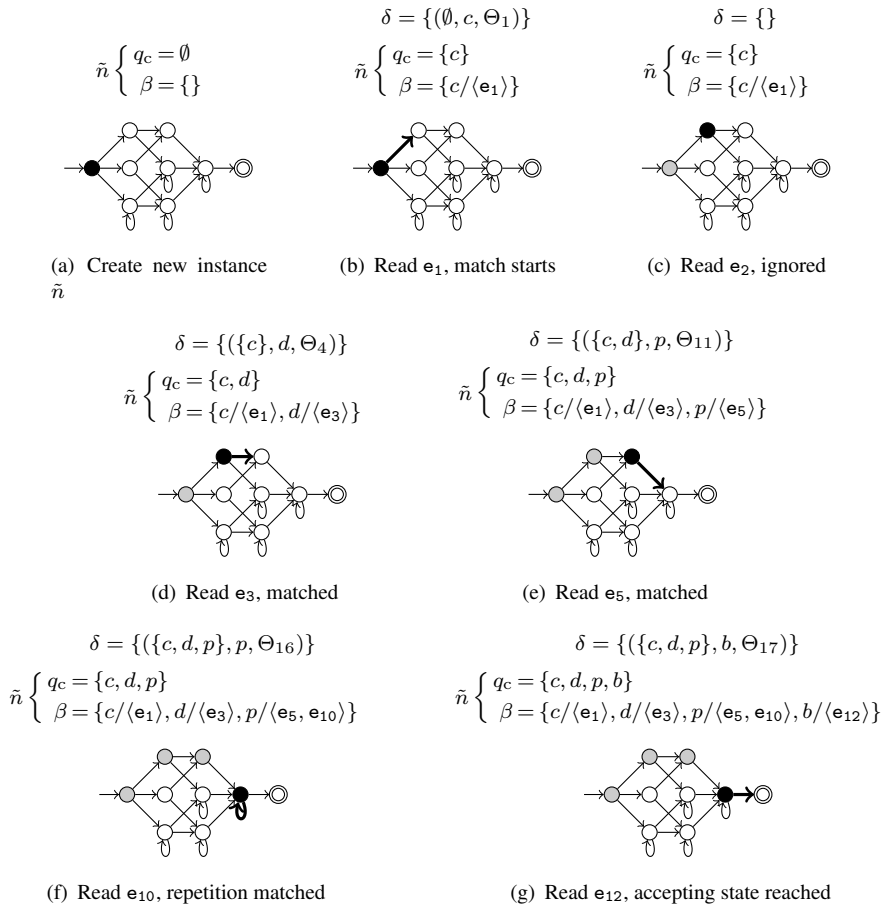
Example 4.5. Figure 4.5 illustrates two match windows, W_1 and W_2, for event stream `Chemo` and duration τ that is equal to fifteen days. The match windows start at event $\mathtt{e}_1$ and $\mathtt{e}_2$, respectively. Both match windows contain 13 events, each with a maximal time span of fifteen days between the first and the last event.

$$E = \underbrace{\mathtt{e}_1, \overbrace{\mathtt{e}_2, \mathtt{e}_3, \mathtt{e}_4, \mathtt{e}_5, \mathtt{e}_6, \mathtt{e}_7, \mathtt{e}_8, \mathtt{e}_9, \mathtt{e}_{10}, \mathtt{e}_{11}, \mathtt{e}_{12}, \mathtt{e}_{13}}^{W_2}}_{W_1}, \mathtt{e}_{14}, \cdots$$

Figure 4.5: Match Windows for `Chemo` and τ = 15 d.

Conceptually, a match window slides over the input stream event-by-event and covers at each step all events that are within duration τ from the earliest event in the window. Depending on the density of the input events along the time axis, the number of events in the match window might vary.

Example 4.6. Figure 4.6 shows a few steps of the execution of the SES automaton N from our running example with match window $W = \langle \mathtt{e}_1, \ldots, \mathtt{e}_{13} \rangle$ from the event stream `Chemo`. The example shows the automaton instance, $\tilde{n}$, that produces a match for patient 1. For each step we show the transition, δ, of the automaton instance ($\delta = \{\}$ if $\tilde{n}$ does not take any transition, i.e., the event is ignored), the current state and match buffer of $\tilde{n}$, and the transition graph. The black node represents the current state of $\tilde{n}$ before taking the transition, thick edges represent the transitions that $\tilde{n}$ takes, and gray nodes represent states which $\tilde{n}$ traversed before. For example, in Figure 4.6(e) the automaton instance is in state $\{c, d\}$, and input event $\mathtt{e}_5$ triggers transition $\delta = (\{c, d\}, p, \Theta_{11})$. The transition moves the automaton instance to state $\{c, p, d\}$ and adds the binding $p/\langle \mathtt{e}_5 \rangle$ to the match buffer β.



(a) Create new instance $\tilde{n}$ (b) Read e_1, match starts (c) Read e_2, ignored

(d) Read e_3, matched (e) Read e_5, matched

(f) Read e_{10}, repetition matched (g) Read e_{12}, accepting state reached

Figure 4.6: Execution of the SES Automaton for $P_1 = (\langle\{c, p^+, d\}, \{b\}\rangle, \Theta, 15\,\mathrm{d})$.

4.5 Algorithm

In this section, we describe two algorithms that together execute a SES automaton. The first algorithm fills the match window with the events that occur within duration τ. The second algorithm applies the SES automaton on the match window. To simplify the notation in the algorithms, we define the union of two sets of bindings. The *union of two sets of bindings* is denoted with the symbol $\uplus$ and has the following semantics: $\{v_i/\vec{e}_i, v_j/\vec{e}_j\} \uplus \{v_i/\vec{e}_l, v_k/\vec{e}_k\} = \{v_i/\langle\vec{e}_i \cup \vec{e}_l\rangle, v_j/\vec{e}_j, v_k/\vec{e}_k\}$.

Algorithm 1 shows function `Match`, which has two input parameters: a pattern P and an event stream E. The function returns all matches of P in E according to the skip-till-next-match event selection strategy. First, P is translated into a SES automaton N. Then, the algorithm iterates over all input events, $e \in E$, and feeds them into the match window W, which — once it is filled up — is passed to algorithm `SESExec` to

compute matches. Hence, before the current input event e can be inserted in the match window W, the algorithm needs to ensure that W does not exceed τ after the event is added. If e's distance from the first event in W exceeds τ, function `SESExec` is called that executes the SES automaton N on the match window W. Finally, after all events in stream E are read, the remaining events in W are processed by the SES automaton.

Algorithm 1: `Match`(P, E)

Input: pattern $P = (\langle B_1, \ldots, B_k \rangle, \Theta, \tau)$, event stream E
Output: set of matches

```
1  Translate P into N = (Q, Δ, q_s, q_f);
2  R ← ∅;
3  W ← empty match window;
4  foreach e ∈ E ordered by T do
5      while |W| > 0 and e.T − W[1].T > τ do
6          R ← R ∪ SESExec(N, W);
7          dequeue(W);
8      enqueue(W, e);
9  while |W| > 0 do
10     R ← R ∪ SESExec(N, W);
11     dequeue(W);
12 return R;
```

Algorithm 2 shows function `SESExec`, which takes as input parameters an automaton N, and a match window W, and returns the set of matches that start with the first event in W. First, the algorithm starts an automaton instance in the start state, q_s, with an empty match buffer β, and adds it to the set of automaton instances Ω. Then, the algorithm iterates over all events $e \in W$ and all automaton instances. For each automaton instance, the algorithm iterates over all outgoing transitions, $\delta \in \Delta$, from the current state, q_c. In each iteration, e is bound to variables v in the transition and stored in β' together with the current buffer. Next, Θ_δ is tested with β'. If β' satisfies Θ_δ, an automaton instance $(q \cup \{v\}, \beta')$ is created. If β' satisfies the transition conditions of several transitions, nondeterminism arises, and several new automaton instances branch from $\tilde{n}$. If none of the transitions fires ($\Omega_\delta = \emptyset$) and the automaton instance $\tilde{n}$ is not in the start state, the event e is ignored. Finally, after all events in W are read, the match buffers of the automaton instances in the accepting state are returned as matches.

4.6 Complexity Analysis

In this section, we analyze the runtime complexity of the algorithm `Match`, i.e., the execution of a SES automaton N for a pattern P with an event stream E as input. The runtime complexity of `Match` predominantly depends on the number of input events in

Algorithm 2: `SESExec`(N, W)

Input: SES automaton $N = (Q, \Delta, q_s, q_f)$, match window W
Output: set of matches

```
1  R ← ∅;
2  Ω ← {(q_s, ∅)};
3  foreach e ∈ W ordered by T do
4      foreach (q_c, β) ∈ Ω do
5          Ω_δ ← ∅;
6          foreach δ = (q, v, Θ_δ) ∈ Δ such that q = q_c do
7              β' ← β ⊎ {v/⟨e⟩};
8              if β' satisfies Θ_δ then
9                  Ω_δ ← Ω_δ ∪ {(q ∪ {v}, β')};
10         if Ω_δ = ∅ ∧ q_c ≠ q_s then
11             Ω_δ ← {(q_c, β)};
12         Ω ← Ω ∪ Ω_δ;
13 foreach (q_c, β) ∈ Ω do
14     if q_c = q_f then
15         R ← R ∪ {β};
16 return R;
```

stream E (Algorithm 1, line 4), the number of events in match window W (Algorithm 2, line 3), and the number of automaton instances in Ω that process each event in W (Algorithm 2, line 4). Automaton instances in Ω are created in the start state of the SES automaton (Algorithm 2, line 2) and if nondeterminism occurs. This gives a runtime complexity for `Match` of

$$\mathcal{O}(|E| \cdot |\widehat{W}| \cdot |\widehat{\Omega}|),$$

where $|\widehat{W}|$ is the maximal number of events in a match window with duration τ sliding over E event-by-event, and $|\widehat{\Omega}|$ is the maximal number of automaton instances during the processing of a match window. Determining $|E|$ and $|\widehat{W}|$ can be done by analyzing the input stream E, whereas for $|\widehat{\Omega}|$ we can provide an upper bound. In the remaining of this section, we analyse the upper bound of $|\widehat{\Omega}|$. We begin with the definition of *mutually exclusive variables*.

Definition 4.4. (*Mutually Exclusive Variables*) Let $(\langle B_1, \ldots, B_k \rangle, \Theta, \tau)$ be a pattern and e be an event. Two variables v, w in P are *mutually exclusive* if and only if

$$\exists v.A\ \phi\ C_1, w.A\ \phi\ C_2 \in \Theta\ (v \neq w \wedge \nexists e\ ((e.A\ \phi\ C_1) \text{ and } (e.A\ \phi\ C_2) \text{ are satisfied })).$$

In other words, two variables in a pattern are mutually exclusive, if there does not exist an event that satisfies all properties of both variables.

Example 4.7. In our running example, all variables are pairwise mutually exclusive, since Θ contains the conditions $c.L =$ 'C', $d.L =$ 'D', $p.L =$ 'P', and $b.L =$ 'B', which are all of the form $v.A\ \phi\ C$, and there does not exist an event that satisfies two of these conditions.

Lemma 4.1. Let $(\langle B_1, \ldots, B_k \rangle, \Theta, \tau)$ be a pattern and N be the corresponding SES automaton. If all variables in P are pairwise mutually exclusive, nondeterminism cannot occur in any state during the execution of N.

Proof. If all variables in a pattern P are pairwise mutually exclusive, Θ contains a condition of the form $v.A\ \phi\ C$ for each variable v that cannot be satisfied by the same event. In the corresponding SES automaton N, each two transitions, $\delta = (q, v, \Theta_\delta)$ and $\delta' = (q, w, \Theta'_\delta)$, that leave a state q have in their transition condition, Θ_δ and Θ'_δ, conditions of the form $v.A\ \phi\ C_1$ and $w.A\ \phi\ C_2$, that cannot be satisfied by the same event. Therefore, the transition condition Θ_δ and Θ'_δ cannot be satisfied contemporarily for the same input event, which excludes nondeterminism. □

The following analysis consists of two parts. In the first part, we consider a SES automaton N_i corresponding to one set B_i from pattern $P = (\langle B_1, \ldots, B_k \rangle, \Theta, \tau)$ in isolation. We distinguish four cases and provide for each case an upper bound for the cardinality of the automaton instances, $|\widehat{\Omega}|$, that process the events of a match window W. In the second part, we describe how to compute the upper bound of $|\widehat{\Omega}|$ of the SES automaton N for the complete pattern P based on the upper bounds from the first part.

4.6.1 Analysis of Single Set

For the first part, we assume a SES automaton, N_i, translated from a single set, B_i in P. To simulate nondeterminism that may occur due to the concatenation of the automata N_i and N_{i+1}, we assume a special transition with an empty variable ϵ and an empty transition condition $(q_{f_i}, \epsilon, \emptyset)$, leaving the accepting state q_{f_i} of N_i and leading to a special state $q_{f_i} \cup \{\epsilon\}$. The special transition is always taken by an automaton instance. In other words, the special transition and the special node simulate variables from B_{i+1} that are not pairwise mutually exclusive with variables in B_i, i.e., an event might contemporarily match variables from B_i and B_{i+1}. Transition $(q_{f_i}, \epsilon, \emptyset)$ and node $\{\epsilon\}$ do not belong to N_i and are only used for the complexity analysis.

Case 1

The variables in set B_i are pairwise mutually exclusive, and B_i does not contain any quantified variable, v^+.

Theorem 4.1. If all variables in B_i are pairwise mutually exclusive, and B_i does not contain any quantified variable, v^+, the upper bound of $|\widehat{\Omega}|$ is $\mathcal{O}(1)$.

Proof. According to Lemma 4.1, if the variables in B_i are pairwise mutually exclusive, nondeterminism cannot occur in N_i. Also the concatenation of automaton N_i to automaton N_{i+1} simulated by the special transition $(q_{f_i}, \epsilon, \emptyset)$ does not introduce any nondeterminism, since $(q_{f_i}, \epsilon, \emptyset)$ is the only transition that leaves the accepting state q_{f_i}. Consequently, the number of automaton instances in $\widehat{\Omega}$ stays constant which leads to a constant upper bound $\mathcal{O}(1)$. □

Figure 4.7 illustrates the SES automaton N_i that is translated from our assumed set B_i with $|B_i| = 3$. Special transition and special node are drawn with dashed lines. The figure shows that for Case 1 only one automaton instance traverses the automaton. The path from the start state to the accepting state was chosen arbitrarily; any other path would be valid as well.

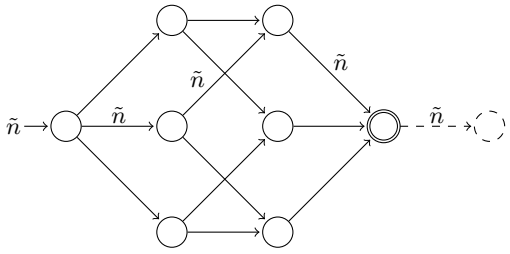


Figure 4.7: Case 1.

Case 2

The variables in set B_i are pairwise mutually exclusive, and B_i contains m quantified variables, v^+.

Theorem 4.2. If all variables in B_i are pairwise mutually exclusive, and B_i contains m quantified variable, v^+, the upper bound of $|\widehat{\Omega}|$ is $\mathcal{O}(|\widehat{W}|)$.

Proof. According to Lemma 4.1, if the variables in B_i are pairwise mutually exclusive, nondeterminism cannot occur in N_i. However, with quantified variables in B_i nondeterminism can occur due to the concatenation of automaton N_i with automaton N_{i+1}. The accepting state q_{f_i} has m transitions that loop. In the worst case an automaton instance that reads an event e in q_{f_i} takes one looping transition (e matches a quantified variable in B_i) and the special transition $(q_{f_i}, \epsilon, \emptyset)$ (e matches a variable in B_{i+1}). Hence, at most one automaton instance branches from the original automaton instance per event read. Thus, assuming that $|\widehat{W}|$ is the maximal number of events in a match window within an event stream E, the number of automaton instances in $\widehat{\Omega}$ has an upper bound $\mathcal{O}(|\widehat{W}|)$. □

Figure 4.8 illustrates the SES automaton N_i that is translated from our assumed set B_i with $|B_i| = 3$ and one quantified variable. It shows that for Case 2 only one

automaton instance traverses the automaton, but then at the accepting state $|W|$ automaton instances branch from the original instance due to nondeterminism introduced by the concatenation of automata simulated by transition $(q_{f_i}, \epsilon, \emptyset)$. As in the previous case, the path from the start state to the accepting state was chosen arbitrarily; any other path would be valid as well.

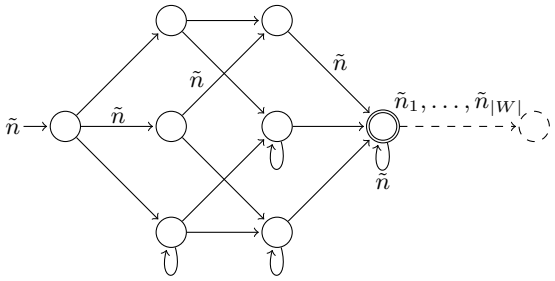


Figure 4.8: Case 2.

Case 3

The variables in set B_i are not pairwise mutually exclusive, and B_i does not contain any quantified variable, v^+.

Theorem 4.3. If the variables in B_i are not pairwise mutually exclusive and B_i does not contain any quantified variable, the upper bound of $|\widehat{\Omega}|$ is $\mathcal{O}(|B_i|!)$.

Proof. Since variables v in B_i are not pairwise mutually exclusive, Lemma 4.1 does not apply and nondeterminism might occur during the execution of N_i. In the worst case, each automaton instance that reaches a state $q \in Q$ branches to a number of automaton instances equal to the number of transitions that leave q. If t is the number of transitions that leave q, $t - 1$ new automaton instances are created, whereas one transition is taken by the original automaton instance. Thus, there exists an automaton instance for each path from the start state to the accepting state. The number of paths in a SES automaton translated from a pattern with one set B_i is $|B_i|!$, which gives an upper bound for $|\widehat{\Omega}|$ of $\mathcal{O}(|B_i|!)$. □

Figure 4.9 shows the SES automaton translated from our assumed set B_i with $|B_i| = 3$. It illustrates how automaton instances branch and which path each one takes to reach the accepting state. Each path in N is used by one automaton instance $\tilde{n}_i$.

Case 4

The variables in B_i are not pairwise mutually exclusive, and B_i contains quantified variables.

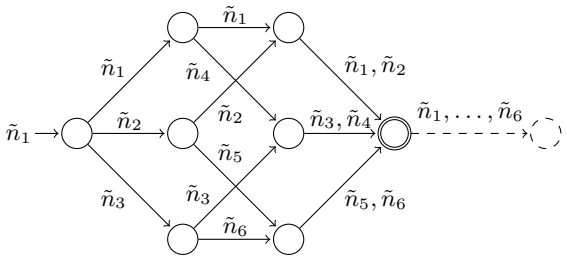


Figure 4.9: Case 3.

Theorem 4.4. If the variables in B_i are not pairwise mutually exclusive and B_i contains m quantified variables, v^+, the upper bound of $|\widehat{\Omega}|$ is

$$\begin{cases} \mathcal{O}((|B_i|-1)! \cdot |\widehat{W}|^{|B_i|}) & m = 1, \\ \mathcal{O}(m \cdot (|B_i|-1)! \cdot m^{|\widehat{W}| \cdot |B_i|}) & m > 1. \end{cases}$$

Proof. Since the variables in B_i are not pairwise mutually exclusive, Lemma 4.1 does not apply and nondeterminism might occur during the execution of N_i. Further, with quantified variables in B_i, N_i has transitions that loop at a state, i.e., source state q and target state $q \cup \{v\}$ of the transition are the same.

As in Case 3, each automaton instance that reaches a state q might branch to a number of automaton instances that is equal to the number of transitions that leave q. However, if there is a transition that loops at q, the automaton instance that takes this transition returns immediately to q and might cause a branching of automaton instances at q when the next input event is consumed.

Let $r_{\text{out}}(q, x_q)$ be a function that returns the number of automaton instances that originate from one automaton instance and that leave q through a transition that does not loop at q. Parameter x_q is the number of events that are consumed by an automaton instance at q. Function $r_{\text{out}}(q, x_q)$ differs depending on the number of transitions that loop at q:

- $r_{\text{out}}(q, x_q) = 1$ for a state without any transition that loops;
- $r_{\text{out}}(q, x_q) = x_q$ for a state with $m = 1$ transition that loops; and
- $r_{\text{out}}(q, x_q) = m^{x_q}$ for a state with $m > 1$ transitions that loop.

On a path from the start state to the accepting state, the maximal number of automaton instances that are created from one automaton instance and that leave a state is

$$|\Omega|_{\text{path}_j} = \prod_{\text{q} \in path_j} r_{\text{out}}(q, x_q).$$

The maximal number of automaton instances created on all paths is then

$$|\Omega|_{\text{out}} = \sum_{j=1}^{|B_i|!} |\Omega|_{\text{path}_j}.$$

The start state, $\emptyset$, is the first state of each path through N_i. Function $r_{\text{out}}(\emptyset, x_\emptyset) = 1$ because $\emptyset$ does not contain any quantified variable, and hence no transition loops at state $\emptyset$. The paths that match a quantified variable in the first transition contain the largest number of states with transitions that loop. The number of such paths is $m \cdot (|B_i| - 1)!$ because m paths start with a quantified variable, and after a quantified variable is matched in the first transition, there are $(|B_i| - 1)!$ paths to the accepting state. The upper bound of $|\widehat{\Omega}|$ with $x_q \leq |\widehat{W}|$ and m quantified variables is therefore

$$\begin{cases} \mathcal{O}((|B_i| - 1)! \cdot |\widehat{W}|^{|B_i|}) & m = 1, \\ \mathcal{O}(m \cdot (|B_i| - 1)! \cdot m^{|\widehat{W}| \cdot |B_i|}) & m > 1. \end{cases}$$

□

Figure 4.10 shows the SES automaton translated from our assumed set B_i with $|B_i| = 3$ and one quantified variable. The labels on the transitions are the result of the functions r_{out}, i.e., the number of automaton instances that leave state q. Thick edges emphasize paths which contain the largest number of states with looping transitions.

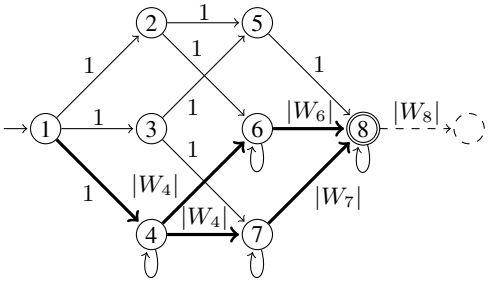


Figure 4.10: Case 4.

4.6.2 Analysis of Complete Pattern

Until now, we considered SES automata for a set B_i from a pattern P in isolation. Below, we describe how to compute the upper bound of $|\widehat{W}|$ for a pattern $P = (\langle B_1, \ldots, B_k \rangle, \Theta, \tau)$ with k sets, $k > 1$.

The SES automaton that corresponds to set B_i starts its execution with the automaton instances created by the SES automaton that corresponds to the previous set

B_{i-1}. Thus, for a pattern P with k sets, the upper bound of the number of automaton instances, $|\widehat{\Omega}|$, created by the corresponding SES automaton N is

$$\mathcal{O}(|\widehat{\Omega}|_1 \cdots |\widehat{\Omega}|_k),$$

where $|\widehat{\Omega}|_i$ is the upper bound for the set B_i in P and is determined according to Cases 1–4. If all variables in all sets B_i in P are pairwise mutually exclusive the upper bound for $|\widehat{\Omega}|$ can be reduced to

$$\mathcal{O}(1),$$

because in the complete SES automaton N nondeterminism cannot occur as stated in Lemma 4.1.

4.7 Experiments

In this section, we report the results of an empirical evaluation using real-world data. The experiments have two purposes. The first purpose is to compare the SES automaton-based algorithm to a brute force approach that matches sequences of sets of events by using a set of automata, each of which matches a sequence of single events. The second purpose is to evaluate the results of the complexity analysis in Section 4.6.

The two hypotheses corresponding to our purposes are the following. First, the SES automaton algorithm is more efficient in terms of runtime than the brute force approach. Second, the runtime is upper-bounded according to the theorems in Section 4.6.

4.7.1 Setup and Data

For the experiments, we implemented the automaton-based evaluation algorithm and the brute force approach in C. The relation with the input events is stored in an Oracle database, Enterprise Edition 11.1, which is accessed over the OCI API. The experiments were performed on a PC with four AMD Opteron 285 processors with 1.8 GHz and 16 GB memory, on which a 64-bit Linux 2.6.32 is installed.

We use two different real-world data sets. The Onco data set contains 341055 chemotherapy events from the Department of Hematology at the Hospital Meran-Merano. The NYSE data set contains 1M share trades in stock markets [49] over 34 hours.

4.7.2 Brute Force Algorithm

We compare our SES automaton that uses one automaton to match sequences of sets of events with a brute force algorithm that uses a set of automata, each of which matches a sequence of single events. The brute force algorithm generates all possible sequences

(orderings) of variables of a set in the pattern, creates a SES automaton for each of these sequences, and executes all automata over the event stream.

Let P be a pattern with sets $\langle B_1, \ldots, B_n \rangle$, where all sets contain only variables without a plus quantifier. A sequence of all variables in P is a concatenation of one permutation of each set B_i. The number of all possible sequences of variables is $|B_1|! \cdot |B_2|! \cdots |B_n|!$. The brute force algorithm creates for each of these sequences an automaton and executes all automata in parallel, i.e., iterates for each match window over these automata. By specifying each variable as a set with exactly one variable, we can use SES automata. The brute force algorithm essentially corresponds to straightforward extensions of the automata in [4, 28, 31]. We do not consider any optimizations of the automata described in these papers, rather our aim is to compare the efficiency of the plain automata to solve SES pattern matching.

Example 4.8. Consider a slight modification of the pattern in our running example, where all variables are variables without a plus quantifier, i.e., $(\langle \{c,\ p,\ d\},\ \{b\} \rangle,\ \Theta,\ 15\ \mathrm{d})$. The possible sequences of variables are given as follows:

$$P_1 = \langle c, d, p, b \rangle$$
$$P_2 = \langle c, p, c, b \rangle$$
$$P_3 = \langle d, c, p, b \rangle$$
$$P_4 = \langle d, p, c, b \rangle$$
$$P_5 = \langle p, c, d, b \rangle$$
$$P_6 = \langle p, d, c, b \rangle$$

Figure 4.11 shows the corresponding SES automaton that is created by our evaluation algorithm and the set of SES automata that are created by the brute force algorithm.

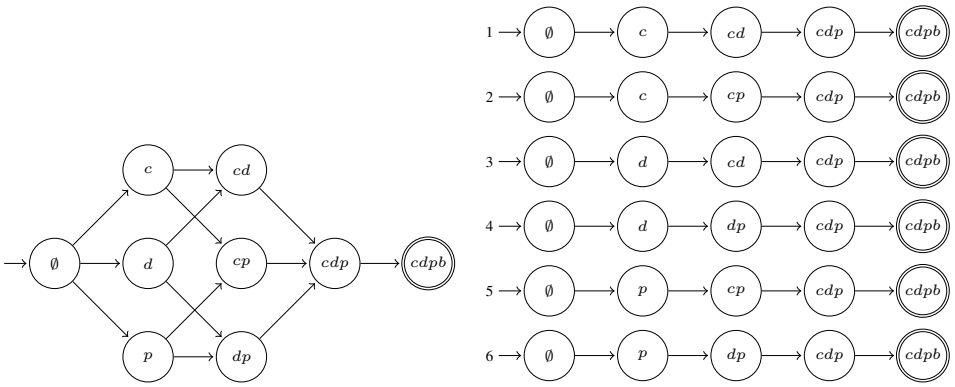
4.7.3 SES Automaton vs. Brute Force

The purpose of this experiment is to compare the SES automaton algorithm (ses) to the brute force (bf) algorithm. Our hypothesis is that the runtime of the SES automaton algorithm is less than the runtime of the brute force algorithm, and that the difference in runtime between the two algorithms is increasing with the number of the variables in a set in the pattern.

We use the following two patterns:

- $P_1 = (\langle \{v_1, v_2, v_3, v_4, v_5, v_6\} \rangle, \Theta_1, \tau)$
- $P_2 = (\langle \{v_1, v_2, v_3, v_4, v_5, v_6\} \rangle, \Theta_2, \tau)$

The conditions in Θ_1 specify that each variable v_i matches distinct types of treatment event of the same patient for the Onco data and trades of distinct shares for the NYSE



(a) SES Automaton (b) Brute Force Algorithm

Figure 4.11: SES Automaton and Set of Automata Created by the Brute Force Algorithm.

data. Thus, all variables are pairwise mutually exclusive. The conditions in Θ_2 specify that all variables match the same type of treatment events for the Onco data and trades of the same share for the NYSE data. Thus, all variables are not pairwise mutually exclusive. The maximal duration τ is 11 days for Onco and 1 s for NYSE.

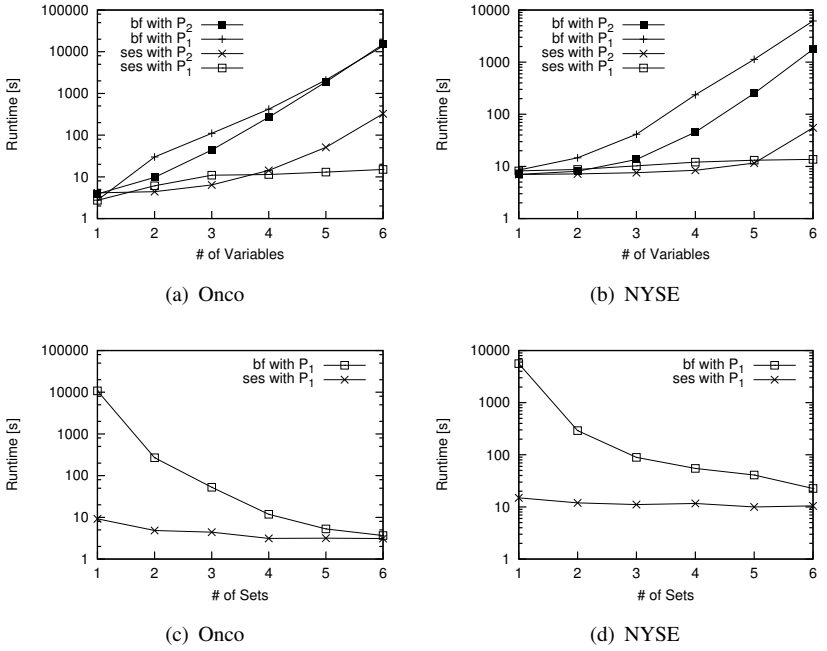
We experiment with two parameters. First, we vary the number of variables in one set from one to six in steps of one, i.e., $\{v_1\}$, $\{v_1, v_2\}, \ldots,$ $\{v_1,\ v_2,\ v_3,\ v_4,\ v_5,\ v_6\}$. Second, while using all six variables, we vary the number of sets from one to six in steps of one, i.e.,

- $\langle\{v_1, v_2, v_3, v_4, v_5, v_6\}\rangle$
- $\langle\{v_1, v_2, v_3\}, \{v_4, v_5, v_6\}\rangle$

  $\vdots$
- $\langle\{v_1\}, \{v_2\}, \{v_3\}, \{v_4\}, \{v_5\}, \{v_6\}\rangle$

The measured parameter is the runtime of the algorithm.

Figure 4.12 shows the results of the experiment. With pattern P_1, the SES automaton algorithm is up to three orders of magnitude faster than the brute force algorithm. The reason is that when the SES automaton algorithm creates one automaton instance for an event that matches a variable in the first set B_1 in P_1, the brute force algorithm creates $(|B_1| - 1)!$ automaton instances because $(|B_1| - 1)!$ automata start with the same variable (see Figure 4.11(b)). Since all variables are pairwise mutually exclusive, nondeterminism does not occur in both algorithms, hence automaton instances never branch during the execution. The differences in the number of automaton instances started can also be seen in the trends of the graphs for the SES automaton algorithm



(a) Onco (b) NYSE

(c) Onco (d) NYSE

Figure 4.12: SES vs. Brute Force

and the brute force algorithm with pattern P_1. The graphs for the SES automaton algorithm have a roughly constant trend for Onco and NYSE whereas the graphs for the brute force algorithm have an (hyper)exponential trend. Notice the logarithmic scale on the ordinate.

With P_2, the SES automaton algorithm needs up to one order of magnitude less runtime than the brute force algorithm. The graphs for both algorithms show a roughly (hyper)exponential trend. The SES automaton algorithm creates $|B_1|$ automaton instances when it consumes an event that matches all variables in B_1. Afterwards, if other events that match all variables in B_1 are encountered the number increases to maximal $|B_1|!$ due to nondeterminism (cf. Theorem 4.3). The brute force algorithm, instead, creates $|B_1|!$ automaton instances at once and keeps this number of instances until all events in the match window are processed, independently of other events that match all variables in B_1.

To summarize, the SES automaton algorithm creates automaton instances when they are needed and the brute force algorithm creates possibly needed automaton instances in advance duplicating execution steps.

4.7.4 Varying the Size of the Match Window

The purpose of this experiment is to validate the upper bound of the runtime

$$\mathcal{O}(|E| \cdot |\widehat{W}| \cdot |\widehat{\Omega}|)$$

and the upper bounds for $|\widehat{\Omega}|$ depending on the size of the match window W. The upper bounds for $|\widehat{\Omega}|$ are

- $\mathcal{O}(1)$ for pairwise mutually exclusive variables without plus (Theorem 4.1),
- $\mathcal{O}(|\widehat{W}|)$ for pairwise mutually exclusive variables with plus (Theorem 4.2),
- $\mathcal{O}(|B_1|!)$ for not pairwise mutually exclusive variables without plus (Theorem 4.3), and
- $\begin{cases} \mathcal{O}((|B_1|-1)! \cdot |\widehat{W}|^{|B_1|}) & m = 1, \\ \mathcal{O}(m \cdot (|B_1|-1)! \cdot m^{|\widehat{W}| \cdot |B_1|}) & m > 1 \end{cases}$ for not pairwise mutually exclusive variables with plus (Theorem 4.4).

Our hypothesis is that the runtime is upper-bounded as stated in the theorems.
We use the following five patterns:

- $P_1 = (\langle B_1 = \{v_1, v_2, v_3\}, B_2 = \{v_4, v_5\}\rangle, \Theta_1, \tau)$
- $P_2 = (\langle B_1 = \{v_1^+, v_2, v_3\}, B_2 = \{v_4, v_5\}\rangle, \Theta_1, \tau)$
- $P_3 = (\langle B_1 = \{v_1^+, v_2^+, v_3\}, B_2 = \{v_4, v_5\}\rangle, \Theta_1, \tau)$
- $P_4 = (\langle B_1 = \{v_1, v_2, v_3\}, B_2 = \{v_4, v_5\}\rangle, \Theta_2, \tau)$
- $P_5 = (\langle B_1 = \{v_1^+, v_2, v_3\}, B_2 = \{v_4, v_5\}\rangle, \Theta_2, \tau)$

With the Onco data, variables v_4 and v_5 in all patterns match distinct blood laboratory examinations, duration τ specifies a maximal duration of 5 days and all events matched refer to the same patient. In patterns P_1, P_2 and P_3, variables v_1, v_2, v_3 match distinct types of medication administrations, hence the variables are not pairwise mutually exclusive. Variables v_1, v_2, v_3 in patterns P_4 and P_5 match the same type of medication administration, so that the variables are not pairwise mutually exclusive.

With the NYSE data, all patterns specify a maximal duration τ of 1 s. In patterns P_1, P_2 and P_3, all variables match trades of distinct shares, and hence all variables are pairwise mutually exclusive. In patterns P_4 and P_5, variables v_1, v_2, v_3 match trades of the same share, so that v_1, v_2, v_3 are not pairwise mutually exclusive.

With both data sets, patterns P_1, P_2 and P_3 validate Theorem 4.1 and 4.2, and P_4 and P_5 validate Theorems 4.3 and 4.4.

To obtain event streams with increasing amounts of events in match windows, we generated ten data sets from the original Onco and NYSE data sets by copying the events that match the variables of the patterns above multiple times. For example, starting with O_1 that represents the original Onco data set, to obtain O_2 we copied the events once, for O_3 we copied them twice, etc. To keep the sizes of the event streams equal, we trimmed the event streams to the size of the original stream by removing the superfluous events at the tail. Table 4.1 shows the generated data sets with the maximal size of the match window $|\widehat{W}|$ for each of the Onco and NYSE event streams.

Onco	$\lvert\widehat{W}\rvert$	NYSE	$\lvert\widehat{W}\rvert$
O_1	2602	N_1	984
O_2	2946	N_2	1225
O_3	3290	N_3	1627
O_4	3634	N_4	1945
O_5	3978	N_5	2431
O_6	4322	N_6	2917
O_7	4666	N_7	3403
O_8	5010	N_8	3889
O_9	5354	N_9	4375
O_{10}	5698	N_{10}	4861

Table 4.1: Data Sets with Different Match Window Sizes.

We vary the maximal window size, $|\widehat{W}|$, by using the data sets O_1 to O_{10} and N_1 to N_{10}. The measured parameter is the runtime of the SES automaton algorithm.

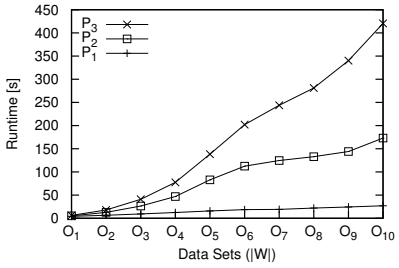
The graphs in Figure 4.13 show the runtime of the SES automaton algorithm depending on the maximal window size $|\widehat{W}|$. The results for patterns P_1, P_2, and P_3 in Figure 4.13(a) and 4.13(c) show a linear trend of runtime with an increasing $\widehat{W}$, which validates the runtime complexity and Theorem 4.1 and 4.2. For patterns P_4 and P_5, the graphs in Figure 4.13(b) and 4.13(d) show a polynomial trend (logarithmic scale of ordinate), which validates Theorem 4.3 and Theorem 4.4.

4.7.5 Varying the Size of the Event Stream

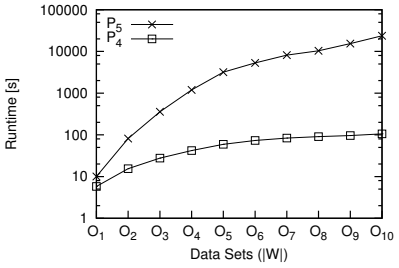
The purpose of this experiment is to show how the runtime of the SES automaton depends on the size of the event stream. Our hypothesis is that the runtime depends linearly from the size of the event stream.

We use the same patterns as in the previous experiment. We vary the size of event streams in ten steps: from 34105 to 341050 events for Onco and from 100K to 1M events for NYSE. For each step we measure the runtime of the SES automaton algorithm.

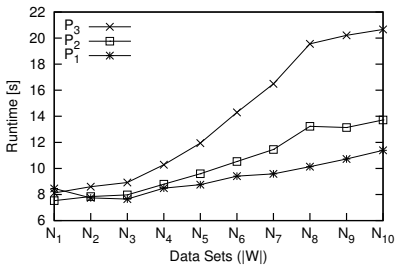
The graphs in Figure 4.14 show the runtime depending on the data size, $|E|$. The results show for all patterns roughly linear trends which confirm our hypothesis.



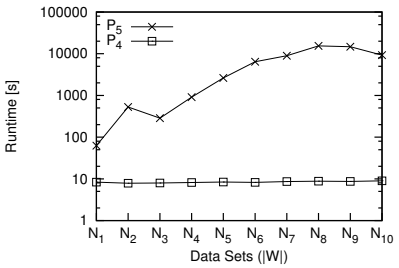
(a) Onco – pairwise mutually exclusive



(b) Onco – not pairwise mutually exclusive

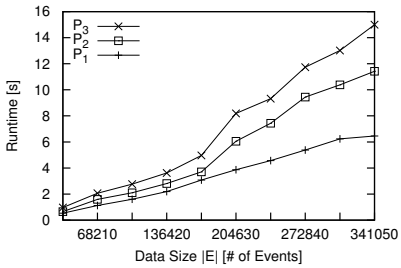


(c) NYSE – pairwise mutually exclusive

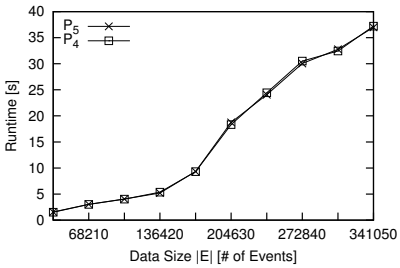


(d) NYSE – not pairwise mutually exclusive

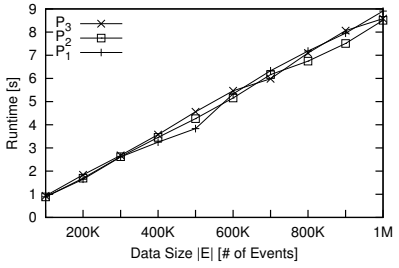
Figure 4.13: Varying the Size of the Match Window.



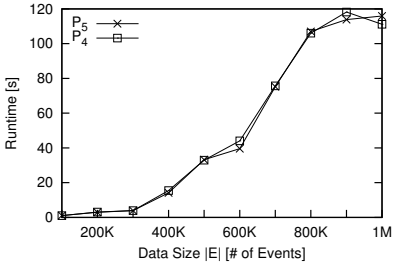
(a) Onco – pairwise mutually exclusive



(b) Onco – not pairwise mutually exclusive



(c) NYSE – pairwise mutually exclusive



(d) NYSE – not pairwise mutually exclusive

Figure 4.14: Varying the Size of the Event Stream.

4.8 Summary

In this chapter, we proposed an automaton-based evaluation algorithm for SES pattern matching. We introduced SES automata and described their construction for a pattern as well as their execution with an input event stream. We provided a detailed analysis of the runtime complexity of the SES automaton algorithm for different types of patterns. The runtime depends on the number of variables and the amount of Kleene plus in the pattern as well as on the number of events in a match window. Regarding the size of the event stream, the runtime is linear. We conducted an experimental evaluation study using real-world data. The results show that the SES automaton algorithm clearly outperforms a brute force approach, which is based on existing techniques. Furthermore, the results validate the complexity analysis.

CHAPTER 5

Two-Phase Evaluation Strategy

5.1 Introduction

In this chapter, we propose a novel two-phase evaluation strategy for event pattern matching that consists of a preprocessing phase and a pattern matching phase. The two-phase evaluation strategy is general enough to be applicable with SES automata described in Chapter 4 as well as with other event pattern matching algorithms as we will show in Section 5.4.

Example 5.1. As a running example for this chapter, we consider the event stream `Chemo` shown in Figure 5.1 that we already used in the previous chapters in combination with the following query:

> Q2: *For each patient, find the events that match (in any order) one administration of Cyclophosphamide (C) and one or more administrations of Prednisone (P) with monotonically increasing values, followed by a single blood count measurement (B); all events must occur within fifteen days.*

The corresponding SES pattern for Query Q2 is

$$P_2 = (\langle\{c, p^+\}, \{b\}\rangle, \Theta, 15\ \mathrm{d}).$$

In the preprocessing phase, incoming events are buffered in the match window, where different pruning techniques, such as filtering, partitioning, and testing for necessary match conditions are applied. The aim of the comparably cheap preprocessing is to reduce the number of input events that need to be processed in the much more expensive pattern matching phase.

Chemo

	PID	*L*	*V*	*U*	*T*
e_1	1	C	1672.5	mg	3 Jul
e_2	1	B	7100	$1/\mu$l	4 Jul
e_3	1	P	111.5	mg	5 Jul
e_4	2	B	10100	$1/\mu$l	6 Jul
e_5	1	D	84	mgl	7 Jul
e_6	2	P	88	mg	8 Jul
e_7	2	D	84	mgl	9 Jul
e_8	2	C	1320	mg	10 Jul
e_9	2	P	98	mg	11 Jul
e_{10}	1	P	116.5	mg	12 Jul
e_{11}	2	P	88	mg	15 Jul
e_{12}	1	B	3400	$1/\mu$l	17 Jul
e_{13}	2	B	4000	$1/\mu$l	18 Jul
e_{14}	2	B	4900	$1/\mu$l	19 Jul
e_{15}	1	B	3000	$1/\mu$l	22 Jul

Figure 5.1: Events of Chemotherapy Treatments.

First, to eliminate irrelevant events that will never participate in a match we propose a *filtering mechanism*, which can be done very efficiently. Filtering out irrelevant events is particularly effective for the skip-till-next-match and skip-till-any-match event selection strategies that both allow to skip events [4].

Second, a critical aspect of current solutions is that each input event is immediately processed by the pattern matching algorithm. For example, in automaton-based algorithms, each event creates a new automaton instance since each event can potentially start a match. This leads to a large number of automaton instances with a significant amount of state information to be maintained. A first step to improve this is to start an automaton only when it leaves the start state [4]. We achieve a substantial reduction of the number of automata by buffering input events in the match window, W, and lazily instantiating an automaton (or equivalently call an event pattern matching algorithm $\mathcal{A}$) only when the buffered events satisfy a set of *necessary match conditions*. We adopt a *summary statistics* based on event counting to efficiently check the necessary match conditions.

Third, although filtering removes irrelevant events, the match window might still contain events that cannot be part of any match starting at the first event in W. For instance, in our example all events in a match must be of the same patient. Thus, all events in W with a patient ID different from the one of the first event do not need to be forwarded to $\mathcal{A}$. We achieve this by *partitioning* incoming events into different windows, based on equality constraints that are derived from the query specification.

In the second phase, an *event pattern matching algorithm*, $\mathcal{A}$, is called for each match window that satisfies the necessary match conditions. This two-phase strategy with a lazy call of the matching algorithm significantly reduces the number of events that need to be processed by $\mathcal{A}$ and the number of calls to $\mathcal{A}$. While pattern matching

algorithms tend to be expensive in terms of runtime and memory complexity, preprocessing can be done very efficiently.

The rest of this chapter is organized as follows. In Section 5.2 we describe the lazy evaluation strategy that first buffers events in a match window and then applies pruning techniques on the match window. Section 5.3 presents an algorithm that implements the two-phase evaluation strategy. In Section 5.4, we report an empirical evaluation using real-world data. Section 5.5 concludes the chapter with a summary.

5.2 Lazy Evaluation using Match Windows

In this section we propose a lazy evaluation strategy for event pattern matching. Instead of eagerly matching incoming events, we advocate a two-phase strategy composed of a preprocessing step and a pattern matching step (see Figure 5.2). First, incoming events are buffered in a match window, which allows to apply pruning techniques on the window, such as filtering, partitioning, and testing for necessary match conditions. Second, an event pattern matching algorithm, $\mathcal{A}$, is called that needs to consider only the events in the match window. The preprocessing phase aims at reducing the number of events that need to be processed by $\mathcal{A}$.

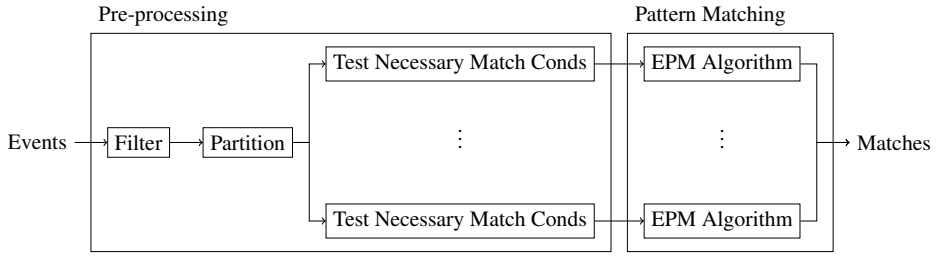


Figure 5.2: Two-phase Evaluation Strategy.

The match window, which buffers incoming events and aids a lazy call of $\mathcal{A}$ is defined in Definition 4.3 in Chapter 4. For convenience, we replicate Example 4.5.

Example 5.2. Figure 5.3 illustrates two match windows, W_1 and W_2, for the event stream `Chemo` and a duration of fifteen days. The match windows start at event $\mathtt{e}_1$ and $\mathtt{e}_2$, respectively. Both match windows contain 13 events, each with a maximal time span of fifteen days between the first and the last event.

A match window $W = \langle e_1, \ldots, e_n \rangle$ is a buffer that collects all incoming events that need to be considered for a match that starts at e_1. This includes all events until the time span between the first event e_1 and the current input event exceeds τ. At this point an event pattern matching algorithm is called to compute the matches that start at e_1. Afterwards, e_1 is removed from W.

$$E = \underbrace{\mathtt{e_1},\ \overbrace{\mathtt{e_2},\ \mathtt{e_3},\ \mathtt{e_4},\ \mathtt{e_5},\ \mathtt{e_6},\ \mathtt{e_7},\ \mathtt{e_8},\ \mathtt{e_9},\ \mathtt{e_{10}},\ \mathtt{e_{11}},\ \mathtt{e_{12}},\ \mathtt{e_{13}}}^{W_2},\ \mathtt{e_{14}},\ \cdots}_{W_1}$$

Figure 5.3: Match Windows for `Chemo` and $\tau = 15$ d.

Definition 5.1. (*Event Pattern Matching Algorithm*) Let $W = \langle e_1, \ldots, e_n \rangle$ be a match window for a pattern P. An *event pattern matching algorithm*, $\mathcal{A}(P, W)$, takes P and W as input and returns the set of matches that start at e_1.

$\mathcal{A}$ can be any event pattern matching algorithm that returns the matches that start at the first event e_1 in the window. To make the discussion more concrete, we assume the automaton-based SES pattern matching algorithm described in Chapter 4. In the evaluation section we apply our framework also to the ZStream [46] event processing system, which is based on the use of join trees rather than automata.

Example 5.3. Consider pattern P_2 and the match windows W_1 and W_2 in Figure 5.3. $\mathcal{A}(P_2, W_1)$ returns a single match $\{c/\langle \mathtt{e_1} \rangle, p/\langle \mathtt{e_3}, \mathtt{e_{10}} \rangle, b/\langle \mathtt{e_{12}} \rangle\}$ that starts at $\mathtt{e_1}$. In contrast, $\mathcal{A}(P_2, W_2)$ for the second match window returns the empty set since no match starts at $\mathtt{e_2}$.

An eager instantiation of automata for each incoming event might lead to a large number of concurrent instances in memory, whereas a lazy instantiation strategy reduces this number.

5.2.1 Filtered Match Windows

While the lazy initialization of automata reduces the number of concurrently active automaton instances, filtering aims at reducing the number of events that need to be processed by $\mathcal{A}$. For this we analyse each incoming event for the properties specified in the pattern.

Not all events in the match window are candidates for matching a variable in the pattern query. Events that cannot contribute to any match can be filtered out before they are passed to $\mathcal{A}$. The following theorem specifies a condition that must be satisfied by each event in a match.

Theorem 5.1. Let $(\langle B_1, \ldots, B_k \rangle, \Theta, \tau)$ be a pattern with variables $v_1, \ldots, v_m$, $\Theta^{\mathrm{p}}_{v_1}, \ldots, \Theta^{\mathrm{p}}_{v_m} \subseteq \Theta$ be the properties of the variables, and γ be a match of P in E. Each event e in γ satisfies $\Theta^{\mathrm{p}}_{v_1}(e) \vee \cdots \vee \Theta^{\mathrm{p}}_{v_m}(e)$.

Proof. The constraints Θ consist of relationships and properties, i.e., $\Theta = \Theta^{\mathrm{r}} \cup \Theta^{\mathrm{p}}$. The properties can further be partitioned by variable, i.e., $\Theta^{\mathrm{p}} = \Theta^{\mathrm{p}}_{v_1} \cup \cdots \cup \Theta^{\mathrm{p}}_{v_m}$. By Definition 3.2, a match γ satisfies $I(\Theta\gamma)$. This can be transformed into $I(\Theta^{\mathrm{p}}_{v_1}\gamma) \wedge \cdots \wedge$

$I(\Theta^{\mathrm{p}}_{v_m}\gamma) \wedge I(\Theta^{\mathrm{r}}\gamma)$. Since $I(\Theta^{\mathrm{p}}_{v_i}\gamma) \equiv I(\Theta^{\mathrm{p}}_{v_i}\{v_i/\langle e_1,\ldots,e_n\rangle\}) \equiv I(\Theta^{\mathrm{p}}_{v_i}\{v_i/\langle e_1\rangle\}) \wedge \cdots \wedge I(\Theta^{\mathrm{p}}_{v_i}\{v_i/\langle e_n\rangle\})$, each event in a match satisfies all properties of the variable it is bound to. □

From Theorem 5.1 we can conclude that an event $e \in E$ can only be part of a match if $\Theta^{\mathrm{p}}_{v_1}(e) \vee \cdots \vee \Theta^{\mathrm{p}}_{v_m}(e)$ is satisfied for some v_i. Therefore, we can filter out events that do not satisfy this condition, which leads to a reduction of the events that must be processed by algorithm $\mathcal{A}$. Notice that only a subset of Θ needs to be checked.

Example 5.4. Consider pattern $P_2 = (\langle\{c, p^+\}, \{b\}\rangle, \Theta, 15\,\mathrm{d})$ and match window W_1 in Figure 5.3. All events e_i that contribute to a match must satisfy $\Theta^{\mathrm{p}}_c(e_i) \vee \Theta^{\mathrm{p}}_p(e_i) \vee \Theta^{\mathrm{p}}_b(e_i)$, which is equivalent to $e_i.L =$ 'C' $\vee\ e_i.L =$ 'P' $\vee\ e_i.L =$ 'B'. Referring to relation `Chemo`, we can easily verify that all events except $\mathtt{e_5}$ and $\mathtt{e_7}$ satisfy this condition. Thus, $\mathtt{e_5}$ and $\mathtt{e_7}$ can safely be filtered out and need not to be buffered. W_1 and W_2 from Figure 5.3 become $W_1' = \langle \mathtt{e_1}, \ldots, \mathtt{e_4}, \mathtt{e_6}, \mathtt{e_8}, \ldots, \mathtt{e_{12}}\rangle$ and $W_2' = \langle \mathtt{e_2}, \ldots, \mathtt{e_4}, \mathtt{e_6}, \mathtt{e_8}, \ldots, \mathtt{e_{13}}\rangle$, respectively.

5.2.2 Candidate Match Windows

While filtering checks for each individual event whether it satisfies the properties of at least one variable, here we consider all events in a match window together and formulate conditions that must be satisfied for a match that starts with the first event in the match window. Match windows that do not satisfy these conditions need not to be passed to $\mathcal{A}$.

Necessary Match Conditions

The following theorem specifies necessary conditions for a match window to contain a match that starts at the first event.

Theorem 5.2. *(Necessary Match Conditions)* Let $(\langle B_1,\ldots,B_k\rangle, \Theta, \tau)$ be a pattern with variables $v_1,\ldots,v_m$. A match window, $W = \langle e_1,\ldots,e_n\rangle$, that contains a match starting at e_1 satisfies the following conditions:

$$|W| \geq m, \tag{5.1}$$

$$\exists v \in B_1\ (\ \Theta^{\mathrm{p}}_v(e_1)\ ), \tag{5.2}$$

$$\forall v \in \{v_1,\ldots,v_m\}\ \exists e_i \in W\ (\ \Theta^{\mathrm{p}}_v(e_i)\ ), \tag{5.3}$$

$$\forall v_i \in B_i, v_j \in B_{i+1}\ (\ \exists e_i, e_j \in W\ (\ \Theta^{\mathrm{p}}_{v_i}(e_i) \wedge \Theta^{\mathrm{p}}_{v_j}(e_j) \wedge e_i.T < e_j.T\ )\ ). \tag{5.4}$$

Proof. Condition 5.1: Since a given event can only appear once in a match, W needs to contain at least as many input events as variables in the query pattern. Condition 5.2: By Definition 3.2, the earliest event in a match is bound to a variable in B_1. Thus, the

first event in W must satisfy the properties Θ_v^{p} of a variable $v \in B_1$. Condition 5.3: By Definition 3.2, each variable $v_1, \ldots, v_m$ is bound to at least one input event that satisfies the properties $\Theta_{v_i}^{\mathrm{p}}$. Hence, W must contain such an event for each variable. Condition 5.4: By Definition 3.2, input events that are bound to variables v_i, v_j in consecutive B_is need to occur in the same chronological order as the B_is in P. □

Theorem 5.2 allows to selectively pass match windows to algorithm $\mathcal{A}$, thereby reducing the number of calls to $\mathcal{A}$. A match window that does not satisfy the necessary match conditions cannot contain a match that starts with the first event in the window.

Example 5.5. Consider $P_2 = (\langle\{c, p^+\}, \{b\}\rangle, \Theta, 15\ \mathrm{d})$ and match window $W_1' = \langle \mathtt{e}_1, \ldots, \mathtt{e}_4, \mathtt{e}_6, \mathtt{e}_8, \ldots, \mathtt{e}_{12}\rangle$ after filtering out $\mathtt{e}_5$ and $\mathtt{e}_7$. W_1' is a candidate match window since the necessary match conditions are satisfied: the cardinality of the match window, $|W_1'| = 11$, is greater than the number of variables, which is 3 (Condition 5.1); event $\mathtt{e}_1$ satisfies all properties of c in B_1 (Condition 5.2); $\mathtt{e}_1$, $\mathtt{e}_3$, and $\mathtt{e}_{12}$ satisfy all properties of c, p, and b, respectively (Condition 5.3), and their order corresponds to the order of B_1 and B_2 (Condition 5.4). Next, consider $W_2' = \langle \mathtt{e}_2, \ldots, \mathtt{e}_4, \mathtt{e}_6, \mathtt{e}_8, \ldots, \mathtt{e}_{13}\rangle$, which is not a candidate match window, since $\mathtt{e}_2$ does not satisfy all properties of c or p (Condition 5.2). Hence, W_2' cannot contain a match that starts at $\mathtt{e}_2$.

Summary Statistics for Match Windows

Verifying Condition 5.1 and 5.2 of the necessary match conditions is efficient and can be done incrementally as the individual events arrive. This is not the case for Condition 5.3 and 5.4, for which all events in the match window must be considered. We maintain a summary statistics over the match window that can be updated incrementally and allows to efficiently verify the conditions.

Definition 5.2. (*Summary Statistics*) Let W be a match window and $P = (\langle B_1, \ldots, B_k\rangle, \Theta, \tau)$ be a pattern with variables $v_1, \ldots, v_m$. A *summary statistics*, $S = \langle (v_1, cnt_1), \ldots, (v_m, cnt_m)\rangle$, is a set of variable-counter pairs such that for each $v_i \in B_i$

$$\begin{aligned} cnt_i = |\{\ e_i \in W :\ & \\ & \Theta_{v_i}^{\mathrm{p}}(e_i) \wedge \forall v_j \in B_{i-1} \exists e_j \in W\ (\ \Theta_{v_j}^{\mathrm{p}}(e_j) \wedge e_j.T < e_i.T\ )\ \}|. \end{aligned} \tag{5.5}$$

The summary statistics maintains a counter for each variable. The counter for a variable $v_i \in B_i$ records the number of events in W that (1) satisfy all properties of v_i and (2) chronologically follow events in W that satisfy the properties of the variables in the previous set B_{i-1}.

Example 5.6. Consider $P_2 = (\langle\{c, p^+\}, \{b\}\rangle, \Theta, 15\ \mathrm{d})$ and match window $W = \langle \mathtt{e}_1, \mathtt{e}_2, \mathtt{e}_3\rangle$. The summary statistics for W is $S = \langle (c, 1), (p, 1), (b, 0)\rangle$. Events $\mathtt{e}_1$

and $\mathtt{e}_3$ satisfy all properties of c and p, respectively, and there is no previous set B_{i-1} in P_2. Although $\mathtt{e}_2$ satisfies all properties of variable b, its counter is zero since there is no event in W that occurs before $\mathtt{e}_2$ and satisfies the properties of p.

Theorem 5.3. Let $P = (\langle B_1, \ldots, B_k \rangle, \Theta, \tau)$ be a pattern with variables $v_1, \ldots, v_m$, W be a match window, and $S = \langle (v_1, cnt_1), \ldots, (v_m, cnt_m) \rangle$ be a summary statistics. The necessary match conditions 5.3 and 5.4 are satisfied if $cnt_1 > 0 \wedge \cdots \wedge cnt_m > 0$.

Proof. cnt_i is greater than zero if and only if there exists at least one event in the set specified by Condition 5.5. Therefore, $cnt_1 > 0 \wedge \cdots \wedge cnt_m > 0$ is equivalent to

$$\begin{aligned}
&\forall v \in B_1\ \exists e \in W\ (\ \Theta^{\mathrm{p}}_v(e)\ ) \wedge \\
&\forall v \in B_2, v' \in B_1\ \exists e, e' \in W\ (\ \Theta^{\mathrm{p}}_v(e) \wedge \Theta^{\mathrm{p}}_{v'}(e') \wedge e'.T < e.T\ ) \\
&\wedge \cdots \wedge \\
&\forall v \in B_k, v' \in B_{k-1}\ \exists e, e' \in W\ (\ \Theta^{\mathrm{p}}_v(e) \wedge \Theta^{\mathrm{p}}_{v'}(e') \wedge e'.T < e.T\ ).
\end{aligned}$$

By applying the rule $\forall x(A \wedge B) \equiv \forall x(A) \wedge \forall x(A \wedge B)$ to all quantified conjuncts we get

$$\begin{aligned}
&\forall v \in B_1\ \exists e \in W\ (\ \Theta^{\mathrm{p}}_v(e)\ ) \wedge \cdots \wedge \forall v \in B_k\ \exists e \in W\ (\ \Theta^{\mathrm{p}}_v(e)\ ) \wedge \\
&\forall 1 < i \leq k\ (\ \forall v \in B_i, v' \in B_{i-1}\ \exists e, e' \in W\ (\ \Theta^{\mathrm{p}}_v(e) \wedge \Theta^{\mathrm{p}}_{v'}(e') \wedge e'.T < e.T\ )\ ),
\end{aligned}$$

which corresponds to Condition 5.3 and 5.4. □

The summary statistics is incrementally updated when input events enter or exit W. When a new event e is added, the properties are evaluated. If e satisfies all properties of a variable $v \in B_i$ and the counters of all variables from the previous set B_{i-1} are greater than zero (i.e., W contains a matching event for these variables), cnt_i is incremented by one. When the first event, e, is dequeued from the match window, the counter for each variable whose properties are satisfied by e is decremented. If the counter for a variable $v \in B_i$ becomes zero, the counters of all variables in the sets B_j, $j > i$, are reset to zero.

Example 5.7. Figure 5.4 shows the summary statistics for a match window W that buffers events of patient 1. The last four columns show the fulfillment of the necessary match conditions. Event $\mathtt{e}_1$ increments the counter for c. Event $\mathtt{e}_2$ matches b, but has no effect on the statistics, since the counter for p is zero. After reading $\mathtt{e}_5$, all counters are greater than zero, and the necessary match conditions are satisfied. Since the next event exceeds the duration of 15 days, the pattern matching algorithm is called with W. Next, $\mathtt{e}_1$ is removed from W and the summary statistics is updated. Since the counter for c becomes zero, the counter for b is reset to zero.

The summary statistics provides an approximate solution and might produce false positives (e.g. due to relationships between variables that are not encoded in the statistics). We show experimentally that the number of match candidates can be significantly reduced.

		Match conditions			
W	S	(5.1)	(5.2)	(5.3)	(5.4)
$\langle \mathtt{e}_1 \rangle$	$\langle (c,1),(p,0),(b,0) \rangle$	F	T	F	F
$\langle \mathtt{e}_1, \mathtt{e}_2 \rangle$	$\langle (c,1),(p,0),(b,0) \rangle$	F	T	F	F
$\langle \mathtt{e}_1, \mathtt{e}_2, \mathtt{e}_3 \rangle$	$\langle (c,1),(p,1),(b,0) \rangle$	T	T	T	F
$\langle \mathtt{e}_1, \mathtt{e}_2, \mathtt{e}_3, \mathtt{e}_9 \rangle$	$\langle (c,1),(p,2),(b,0) \rangle$	T	T	T	F
$\langle \mathtt{e}_1, \mathtt{e}_2, \mathtt{e}_3, \mathtt{e}_9, \mathtt{e}_{11} \rangle$	$\langle (c,1),(p,2),(b,1) \rangle$	T	T	T	T
$\langle \mathtt{e}_2, \mathtt{e}_3, \mathtt{e}_9, \mathtt{e}_{11} \rangle$	$\langle (c,0),(p,2),(b,0) \rangle$	T	F	F	F

Figure 5.4: Summary Statistics for $P_2 = (\langle \{c, p^+\}, \{b\} \rangle, \Theta, 15\text{ d})$.

5.2.3 Partitioned Match Windows

The last optimization aims to further reduce the number of events in a match window that are processed by algorithm $\mathcal{A}$. The core idea is to remove events that, while fulfilling the properties of a variable, can be excluded from a match since they violate some relationships. If all events in a match must have identical values in one or more attributes, the corresponding equality relationships in Θ can be used to partition a match window W into a set of match windows $W_1, \ldots, W_n$.

To formalize the partitioning of match windows based on equality relationships, we first define the transitive closure of these relationships. Let $(\langle B_1, \ldots, B_k \rangle, \Theta, \tau)$ be a pattern with variables $v_1, \ldots, v_m$ and $\Theta_{eq} = \{\theta : \theta \in \Theta \wedge \theta \equiv v_i.A_i = v_j.A_i\}$ be the set of all relationships with equality constraints over a single attribute. The *transitive closure* of Θ_{eq} is defined as

$$\Theta_{eq}^{+} = \bigcup_{l \in \mathbb{N}} \Theta_{eq}^{l},$$

where $\Theta_{eq}^{0} = \Theta_{eq}$ and $\Theta_{eq}^{l} = \{v_i.A_i = v_j.A_i : \exists v_k((v_i.A_i = v_k.A_i) \in \Theta_{eq}^{l-1} \wedge (v_k.A_i = v_j.A_i) \in \Theta_{eq}^{l-1})\}$.

Definition 5.3. (*Partitioned Match Windows*) Let W be a match window for a pattern $(\langle B_1, \ldots, B_k \rangle, \Theta, \tau)$ with variables $v_1, \ldots, v_m$. Furthermore, let $X = \{ A : \forall i, j \in [1, m]((v_i.A = v_j.A) \in \Theta_{eq}^{+}) \}$. The set of attributes X partitions W into $W_1, \ldots, W_n$ such that for each W_i the following holds:

$$\forall e_i, e_j \in W_i \, \forall A \in X \, (\, e_i.A = e_j.A \,).$$

X is a maximal set of *partitioning attributes* that have to assume identical values for all events in a match. Accordingly, all events in a partition have identical values in the attributes X.

Example 5.8. Consider $P_2 = (\langle \{c, p^+\}, \{b\} \rangle, \Theta, 15\text{ d})$. The transitive closure of the equality relationships is $\Theta_{eq}^{+} = \{ \, c.PID = p.PID, \, c.PID = b.PID, \, p.PID = b.PID,$

$c.PID = c.PID$, $p.PID = p.PID$, $b.PID = b.PID$ }. The set of partitioning attributes is $X = \{PID\}$. That is, all events in a match must have the same patient ID. Partitioning $W_1' = \langle \mathtt{e}_1, \ldots, \mathtt{e}_4, \mathtt{e}_6, \mathtt{e}_8, \ldots, \mathtt{e}_{12} \rangle$ by PID results in two partitions, $\langle \mathtt{e}_1, \mathtt{e}_2, \mathtt{e}_3, \mathtt{e}_{10}, \mathtt{e}_{12} \rangle$ containing all events of patient 1 and $\langle \mathtt{e}_4, \mathtt{e}_6, \mathtt{e}_8, \mathtt{e}_9, \mathtt{e}_{11} \rangle$ with all events of patient 2.

Partitioning input events allows two types of optimizations. First, since each partition W_i can be processed independently by algorithm $\mathcal{A}$, parallel event processing is possible. Second, after partitioning, the relationships $v_i.A_i = v_j.A_i$ for all $A_i \in X$ need not to be considered anymore by $\mathcal{A}$. They can be removed from Θ before calling $\mathcal{A}$ to reduce conditions that need to be verified in $\mathcal{A}$.

5.3 Algorithm

Algorithm 3 implements the two-phase evaluation strategy for event pattern matching from the previous section. The input parameters are a pattern P, an event stream E, and an event pattern matching algorithm $\mathcal{A}$. The algorithm returns the set of matches.

In the initialization phase, an empty hash table H and result set R are created, and the set X of partitioning attributes is determined, which serves as a key for the hash table. H stores triples of the form (K, W, S), where K are values of the partitioning attributes X, W is a match window, and S is the summary statistics over W.

The main loop iterates over all input events in chronological order. Events that do not satisfy the properties of any variable are immediately filtered out. For events that pass the filter, the corresponding entry in H is retrieved; if no such entry exists, a new entry is added with key set to the values of the partitioning attributes of the current event, an empty match window, and a summary statistics with all counters set to zero. Before inserting the current event e in W, the algorithm ensures that W does not exceed τ when the event is added. If e's distance from the first event in W exceeds τ, the necessary match conditions are verified; if they are satisfied, algorithm $\mathcal{A}$ is called. Afterwards, the first event is removed from W and the summary statistics is updated. This step is repeated until e fits into W and can be added; the summary statistics is updated.

After reading all input events, the remaining events in the match windows need to be processed. As long as the necessary match conditions are satisfied, $\mathcal{A}$ is called and the first event is removed.

The summary statistics can efficiently be implemented using bit vectors. Let $P = (\langle B_1, \ldots, B_k \rangle, \Theta, \tau)$ be a pattern with variables $v_1, \ldots, v_m$. We use a bit vector, $\bar{\mathtt{e}}_i$, of length m to mask each incoming event e_i. Each position in the bit vector corresponds to a variable in P. If a bit is `1`, the event satisfies all properties of the corresponding variable; otherwise it is set to `0`. Similarly, we use a bit vector, $\bar{\mathtt{s}}$, where each position corresponds to a counter in the summary statistics. A `0` bit means that the corresponding counter is greater than zero; otherwise the counter is zero. Finally, to check whether

Algorithm 3: `Match`$(P, E, \mathcal{A})$

Input: pattern $P = (\langle B_1, \ldots, B_k \rangle, \Theta, \tau)$ with m variables, event stream E, event pattern matching algorithm $\mathcal{A}$
Output: set of matches

```
1  H ← ∅;
2  R ← ∅;
3  Compute set X = (X_1, ..., X_k) of partitioning attributes;
4  foreach e ∈ E ordered by T do
5      if Θ^p_{v_1}(e) or ... or Θ^p_{v_m}(e) then
6          K' ← (e.X_1, ..., e.X_k);
7          if ∄(K, W, S) ∈ H with K = K' then
8              Add (K', ∅, ⟨(v_1, 0), ..., (v_m, 0)⟩) to H;
9          Let (K, W, S) be the entry in H with K = K';
10         while |W| > 0 and e.T − W[1].T > τ do
               // match window exceeds τ with event e
11             if |W| ≥ m and ∃v ∈ B_1(Θ^p_v(W[1])) and ∀(v, cnt) ∈ S(cnt > 0) then
12                 R ← R ∪ A(W, P);
13             dequeue(W);
14             Update S;
15         enqueue(W, e);
16         Update S;
17 foreach (K, W, S) ∈ H do
18     while |W| ≥ m and ∃v ∈ B_1(Θ^p_v(W[1])) and ∀(v, cnt) ∈ S(cnt > 0) do
19         R ← R ∪ A(W, P);
20         dequeue(W);
21         Update S;
22 return R;
```

the counters of all variables of a set B_i are greater than zero we specify a bit mask, $\bar{\mathtt{b}}_i$, for each B_i. It has all bits set to `0` except for the bits that represent the positions of the variables in the previous set, B_{i-1}. Using these bit vectors in combination allows an efficient update and querying of the summary statistics.

Example 5.9. Consider $P_2 = (\langle \{c, p^+\}, \{b\} \rangle, \Theta, 15\ \mathrm{d})$ and the events $\mathtt{e}_1$, $\mathtt{e}_2$, and $\mathtt{e}_3$. Event $\mathtt{e}_1$ satisfies all properties of variable c, but not for p and b. The corresponding bit vector is $\bar{\mathtt{e}}_1 = \mathtt{100}$. For the events $\mathtt{e}_2$ and $\mathtt{e}_3$ the bit vectors are, respectively, $\bar{\mathtt{e}}_2 = \mathtt{001}$ and $\bar{\mathtt{e}}_3 = \mathtt{010}$. The bit vector for the summary statistics is $\bar{\mathtt{s}} = \mathtt{111}$ at the beginning and `001` after processing $\mathtt{e}_1$, $\mathtt{e}_2$, and $\mathtt{e}_3$ (cf. Figure 5.4). The bit mask for set B_1 is $\bar{\mathtt{b}}_1 = \mathtt{000}$; for B_2 it is $\bar{\mathtt{b}}_2 = \mathtt{110}$.

5.4 Experiments

In this section, we report the results of an empirical evaluation using real-world data. The experiments have three purposes: (1) to show the effect of the pruning techniques for the match window on the automaton-based SES pattern matching algorithm that finds matches that conforms to skip-till-next-match and on the join-based ZStream pattern matching algorithm [46] that finds all matches (skip-till-any-match); (2) to show the scalability of the two-phase evaluation strategy in the pattern, and (3) to show the scalability of the two-phase evaluation strategy in the data.

5.4.1 Setup and Data

We implemented our two-phase evaluation strategy with the SES algorithm and ZStream algorithm in C. The event stream is stored in an Oracle database, Enterprise Edition 11.1, which is accessed over the OCI API. The experiments were performed on a PC with four AMD Opteron 285 processors with 1.8 and 2.6 GHz and 16 GB memory, on which a 64-bit Linux 2.6.32 is installed.

We use two different real-world data sets. The Onco data set contains 341055 chemotherapy events from the Department of Hematology at the Hospital Meran-Merano. The NYSE data set contains 1M share trades in stock markets [49] over 34 hours.

In the experiments, we analyze the scalability by varying the number of variables, the length of τ, and the size of the event stream. For the experiments with a varying number of variables, we use the following two pattern queries:

- $P_{\text{seq}} = (\langle\{v_1\}, \{v_2\}, \ldots, \{v_k\}\rangle, \Theta, \tau)$
- $P_{\text{set}} = (\langle\{v_1, v_2, \ldots, v_k\}\rangle, \Theta, \tau)$

The number of variables varies from $k = 1, \ldots, 10$. For each step, we randomly choose ten distinct patterns out of all possible patterns with k variables, and we take the average of the measured throughput. The duration τ is 10 days for the Onco data set and 20 ms for the NYSE data set. P_{seq} is a sequential pattern where the matching events must occur in the same order as the variables in the pattern. In P_{set} the matching events may occur in any order.

For the experiments with a varying length of τ, we use the following patterns.

- $P_{\text{onco}} = (\langle\{c, d\}, \{p^+, r^+\}, \{b, h\}\rangle, \Theta_{\text{onco}}, \tau)$
- $P_{\text{nyse}} = (\langle\{a^+, g^+\}, \{i^+\}\rangle, \Theta_{\text{nyse}}, \tau)$

P_{onco} roughly resembles a cycle of a chemotherapy treatment. b matches white blood cell counts, h red blood cell counts, and the rest different medication administrations. Partitioning is applied on the patient ID. P_{nyse} specifies that a, g, and i match share

trades of Apple, Google, and IBM, respectively. The price of the Apple and Google shares must be increasing between trades (i.e., $prev(a.\text{price}) < a.\text{price}$), followed by a decrease of IBM shares (i.e., $prev(i.\text{price}) > i.\text{price}$). Apple and Google share trades can occur interleaved. We vary τ from 5 days to 50 days for the Onco data set and from 10 to 100 ms for the NYSE data set.

For the experiments with a varying size of E, we use the pattern P_{nyse} and vary the size of E from 100 to 1M events in steps of a factor 10.

5.4.2 Scalability in the Pattern

In this experiment, we study the scalability of our framework by varying the number of variables and the length of τ in the pattern, respectively. Our hypothesis is that our two-phase evaluation strategy significantly increases the throughput.

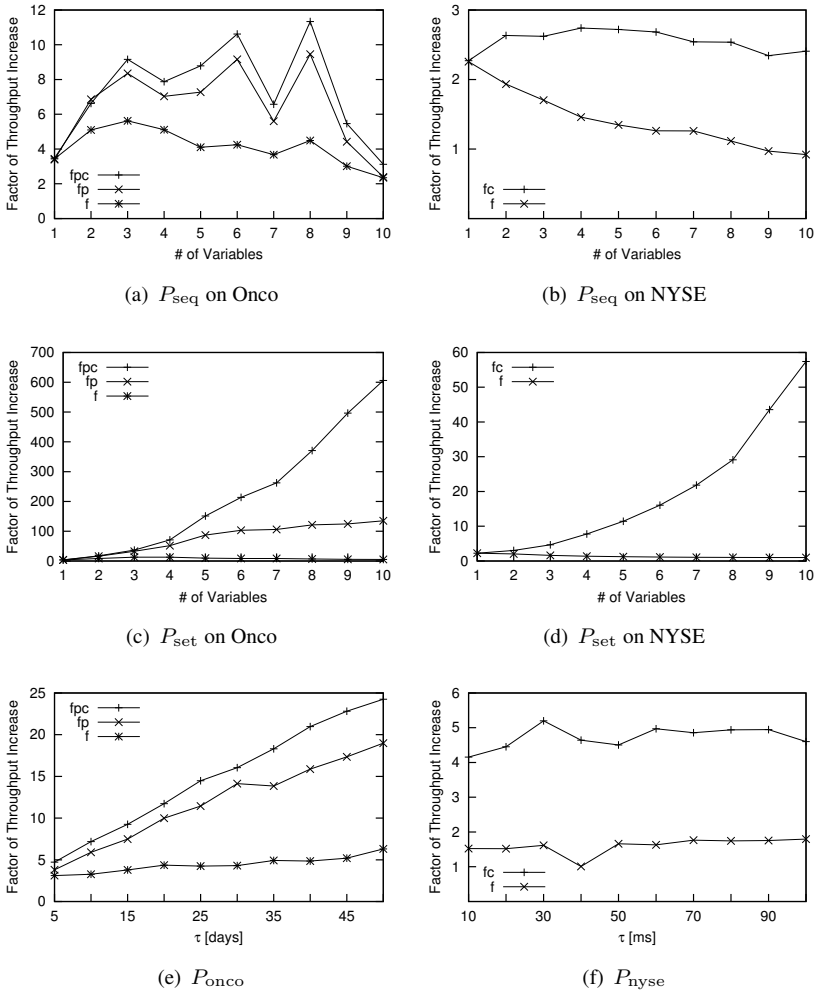
We first run the SES and ZStream algorithms with a plain match window without any pruning techniques (baseline), and apply then filtering (f), partitioning (p), and testing for necessary match conditions (c). The performance of the optimizations is measured in terms of throughput increase with respect to the baseline algorithm.

Figures 5.5 and 5.6 show the factor of the throughput increase of the various pruning techniques relative to the baseline algorithm. The first observation is that the match windows achieve a significant improvement over the baseline solution, up to several orders of magnitudes for P_{set}.

Second, the optimizations are more effective for SES than for ZStream. The reasons are that ZStream implicitly filters new events when they are added to the join tree, and it uses hash tables for the joins on equality predicates. However, filtering, partitioning and testing for necessary conditions together still increase the throughput three to four times.

The third observation is that for SES the optimizations are much more effective for P_{set} (more than two orders of magnitude) than for P_{seq}. With P_{set} and without any pruning techniques, every event that matches any variable starts an automaton instance, yielding a large number of automaton instances, many of which do not lead to a match. With P_{seq}, only events that match the first variable in the pattern start an automaton instance. The two-phase evaluation strategy has a larger potential to increase the throughput of P_{set} than of P_{seq}.

The fourth observation is that the throughput for P_{onco} increases with increasing duration τ for SES. Events in the Onco data set follow roughly a chemotherapy protocol. Individual chemotherapy treatments are shifted in time, hence the patients are not distributed uniformly over the data. With increasing τ, events of more patients are encountered, hence more partitions of match windows can be created. SES can fully exploit the increasing number of partitions.



(a) P_{seq} on Onco

(b) P_{seq} on NYSE

(c) P_{set} on Onco

(d) P_{set} on NYSE

(e) P_{onco}

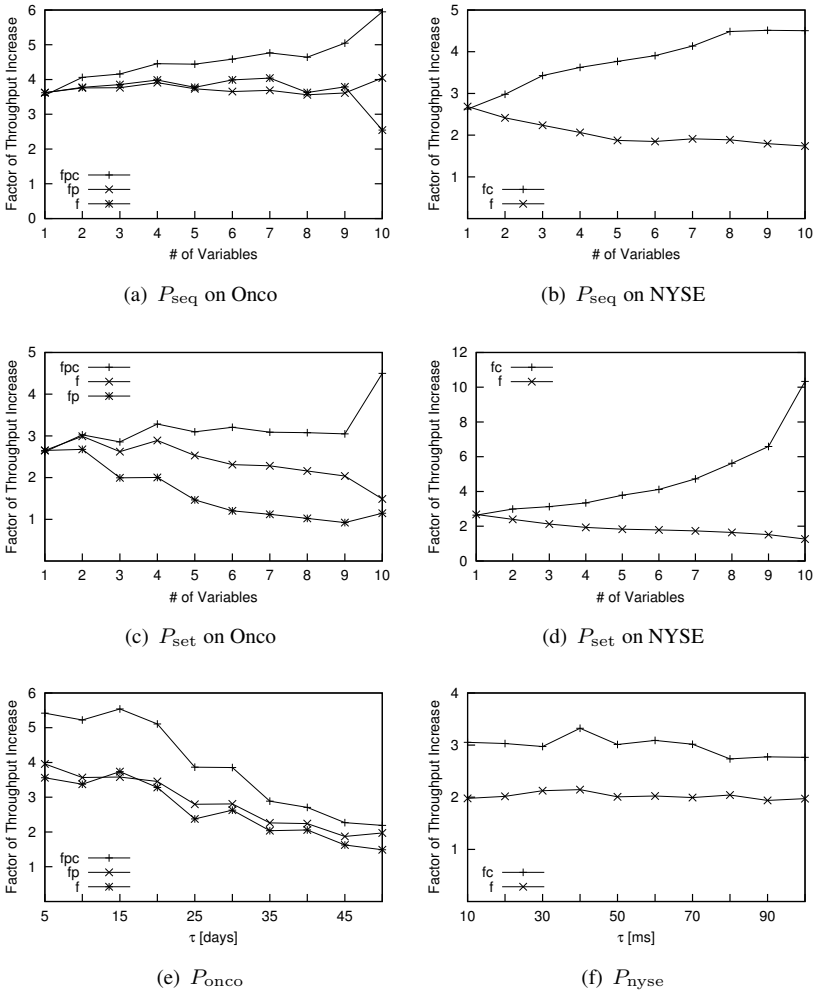
(f) P_{nyse}

Figure 5.5: Varying the Pattern with SES.

5.4.3 Scalability in the Data

In this experiment, we study the scalability of our framework by varying the number of events in the input stream. Our hypothesis is that the throughput increase remains constant with increasing size of the stream. Again, we run first the baseline SES and ZStream algorithms on NYSE, and apply then filtering (f) and testing for necessary match conditions (c).

Figure 5.7 shows the factor of the throughput increase relative to the baseline algorithm. As expected, the throughput increase for our two-phase evaluation strategy remains roughly constant. The reason is that neither the filtering nor the testing for necessary match conditions depends on the length of the event stream. Notice the log-



(a) P_{seq} on Onco

(b) P_{seq} on NYSE

(c) P_{set} on Onco

(d) P_{set} on NYSE

(e) P_{onco}

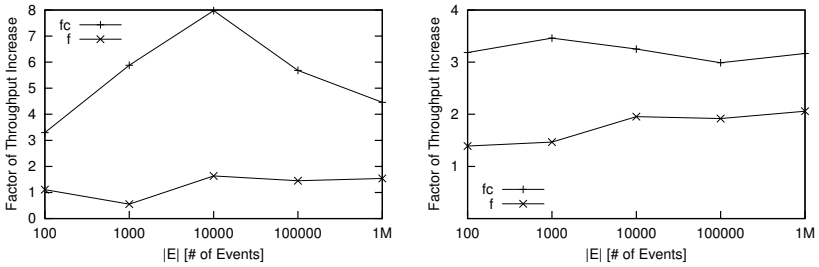
(f) P_{nyse}

Figure 5.6: Varying the Pattern with ZStream.

arithmic scale on the horizontal axis.

5.5 Summary

In this chapter we presented a novel pattern matching strategy that consists of two phases, a preprocessing phase and a pattern matching phase. In the preprocessing phase, incoming events are buffered in a match window, which allows to apply different pruning techniques, such as filtering, partitioning, and testing for necessary match conditions, and aids a lazy evaluation of a pattern matching algorithm. Filtering eliminates events that cannot contribute to a match from the match window. Partitioning reduces



(a) P_{nyse} with SES (b) P_{nyse} with ZStream

Figure 5.7: Varying the Size of the Data.

the events in a match window for each call of the event pattern matching algorithm and allows for parallel processing. Testing for necessary match conditions reduces the number of calls to the event pattern matching algorithm. We conducted extensive experiments using two real-world data sets and two existing event pattern matching algorithms. The results show that our framework significantly increases the throughput for both algorithms.

CHAPTER 6

Improving the Skipping of Noise

6.1 Introduction

In this chapter, we propose a robust skip-till-next-match event selection strategy that improves the skipping of noise with respect to the skip-till-next-match event selection strategy. To find all matches according to robust skip-till-next-match, we present a backtracking mechanism that extends automaton-based event pattern matching algorithms. Such an approach avoids large intermediate results.

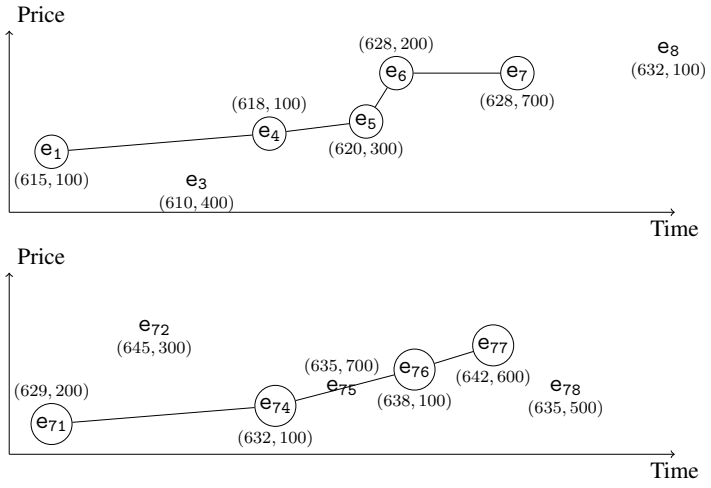
The widely used skip-till-next-match event selection strategy skips all irrelevant (or uninteresting [28]) events until the next relevant event is read. It has been used in scenarios where some events in the input stream are noise to the pattern and therefore should be ignored. Examples for application scenarios that use skip-till-next-match are medical data analysis as presented in the previous chapters, stock market analysis [4, 29], credit card fraud detection [55] and publish/subscribe systems [28].

The skip-till-next-match event selection strategy strongly depends on how relevant and irrelevant events are determined. In previous work, determining relevant events is tightly bound to the greedy matching of input events. That is, only the current partial match consisting of the events matched so far together with the corresponding constraints in the pattern are used to determine whether an input event is relevant or not, and once this is determined, the decision is not revisited. In contrast, we advocate to determine relevant events by considering all constraints in the pattern together with a complete match. At the technical level, we implement this through an efficient backtracking mechanism that allows to change an event from being relevant to irrelevant. We are motivated by the observation that in some cases an event that seems to be relevant with respect to a partial match might turn out to be actually irrelevant later on

Stocks

	S	P	V	T
e_1	GOOG	615	100	9:32:344
e_2	IBM	204	200	9:32:357
e_3	GOOG	610	400	9:32:368
e_4	GOOG	618	100	9:32:380
e_5	GOOG	620	300	9:32:396
e_6	GOOG	628	200	9:32:401
e_7	GOOG	628	700	9:32:421
e_8	GOOG	632	100	9:32:450
⋮	⋮	⋮	⋮	⋮
e_{71}	GOOG	629	200	14:15:555
e_{72}	GOOG	645	300	14:15:572
e_{73}	MSFT	28	100	14:15:581
e_{74}	GOOG	632	100	14:15:592
e_{75}	GOOG	635	700	14:15:605
e_{76}	GOOG	638	100	14:15:613
e_{77}	GOOG	642	600	14:15:628
e_{78}	GOOG	635	500	14:15:640

(a) Event Stream Stocks



(b) Google Stock Trades over Time (Price, Volume) and Two Matches for Q3

Figure 6.1: Stock Trade Events.

when more input events are read. We illustrate this in the following example.

Example 6.1. Consider event stream Stocks in Figure 6.1(a) that records stock trades. The attributes represent stock symbol (S), price per stock (P), volume of the trade (V), and occurrence time (T). For instance, event e_1 represents the trade of 100 Google stocks at a price of \$615 at time 9:32:344. Figure 6.1(b) shows Google trades plotted over time and labeled with stock price and trade volume. To analyze Google stock

trades, consider the following pattern query:

> Q3: *Within a period of 100 ms, find three or more Google (*GOOG*) stock trades with strictly monotonic increasing prices, followed by one Google stock trade with volume larger than each of the price-increasing trades.*

Query Q3 is formulated as SES pattern

$$P_3 = (\langle \{s_1\}, \{s_2\}, \{s_3^+\}, \{s_4\} \rangle, \Theta, 100\,\text{ms}).$$

Figure 6.1(b) shows two matches for pattern P_3. The first match consists of the events $\mathtt{e}_1$, $\mathtt{e}_4$, $\mathtt{e}_5$, $\mathtt{e}_6$, and $\mathtt{e}_7$: $\mathtt{e}_1$ starts the match; $\mathtt{e}_4$, $\mathtt{e}_5$, $\mathtt{e}_6$ are subsequent events with strictly increasing prices ($\$615 < \$618 < \$620 < \$628$); and $\mathtt{e}_7$ is the final event that has a larger trade volume than $\mathtt{e}_1$, $\mathtt{e}_4$, $\mathtt{e}_5$, and $\mathtt{e}_6$ ($100 < 700$, $100 < 700$, $300 < 700$, $200 < 700$). The events $\mathtt{e}_2$ and $\mathtt{e}_3$ are skipped as irrelevant and are not part of the match: $\mathtt{e}_2$ is not a Google stock trade ('IBM' $\neq$ 'GOOG'), and $\mathtt{e}_3$ has a decreasing instead of an increasing price relative to $\mathtt{e}_1$ ($\$615 > \$610$).

The second match consists of $\mathtt{e}_{71}$, $\mathtt{e}_{74}$, $\mathtt{e}_{76}$, and $\mathtt{e}_{77}$ with $\$629 < \$632 < \$638$ and $200 < 600$, $100 < 600$, $100 < 600$ (see Figure 6.1(b)). Events $\mathtt{e}_{72}$, $\mathtt{e}_{73}$, and $\mathtt{e}_{75}$ are irrelevant and therefore not included in the match: though $\mathtt{e}_{72}$ has an increasing price relative to $\mathtt{e}_{71}$ ($\$629 < \$645$), there are no events in the `Stocks` stream that continue this trend to complete the match; event $\mathtt{e}_{73}$ is not a Google stock trade ('MSFT' $\neq$ 'GOOG'); and the volume of $\mathtt{e}_{75}$ is larger than the one of $\mathtt{e}_{77}$ that follows the price-increasing trend ($700 > 600$).

The skip-till-next-match event selection strategy finds the first match with the events $\mathtt{e}_1$, $\mathtt{e}_4$, $\mathtt{e}_5$, $\mathtt{e}_6$, and $\mathtt{e}_7$, but misses the second match with the events $\mathtt{e}_{71}$, $\mathtt{e}_{74}$, $\mathtt{e}_{76}$, and $\mathtt{e}_{77}$. The second match is missed because $\mathtt{e}_{72}$, which has a higher price than the previous event $\mathtt{e}_{71}$, is greedily matched. The subsequent events $\mathtt{e}_{74}$, $\mathtt{e}_{75}$, $\mathtt{e}_{76}$, and $\mathtt{e}_{77}$ are skipped since they do not continue the positive trend. Note that event $\mathtt{e}_{72}$ is considered relevant because it satisfies the constraints with $\mathtt{e}_{71}$ (i.e., follows the price trend), and subsequent events that will be included in the match are not considered. If $\mathtt{e}_{72}$ would be skipped as irrelevant, the subsequent events can be matched successfully. The same problem with the greedy matching strategy occurs for $\mathtt{e}_{75}$. Initially, this event is considered relevant since it continues the positive price trend. Later, when processing $\mathtt{e}_{77}$ it must be reclassified as irrelevant to permit the second match.

Robust skip-till-next-match finds both matches. Regarding the second match, $\mathtt{e}_{72}$ and $\mathtt{e}_{75}$ are correctly determined as irrelevant events since all constraints in the pattern together with all events in the complete match are considered. More specifically, our strategy recognizes that $\mathtt{e}_{72}$ has a price and $\mathtt{e}_{75}$ a volume that violate the constraints in the pattern.

Our robust skip-till-next-match event selection strategy considers all constraints in the pattern together with a complete match to determine whether events are relevant or not. Robust skip-till-next-match finds all matches of a pattern in an event stream that

contain the events occurring earliest after the start of the match. Robust skip-till-next-match finds all matches that are found with skip-till-next-match in addition to those matches that are not found due to the erroneous matching of irrelevant events. The robust skip-till-next-match event selection strategy has the property that the result set can be derived from the set of all possible matches of the pattern (which corresponds to the skip-till-any-match event selection strategy in Chapter 1) without accessing the original event stream. In general, this is not possible for the set of matches that conform to skip-till-next-match. This property is useful in interactive analysis, where first all possible matches are analysed and then in a following analysis step the matches should be restricted.

The rest of this chapter is organized as follows. In Section 6.2, we formally define robust skip-till-next-match and compare it to skip-till-next-match. Section 6.3 presents the backtracking mechanism for automaton-based event pattern matching algorithms. The algorithm that implements the backtracking mechanism is described in Section 6.4. In Section 6.5 we report the results of an empirical evaluation using real-world data. Section 6.6 summarizes the chapter.

6.2 Robust Skip-till-next-match

In this section, we introduce the robust skip-till-next-match event selection strategy and compare it to skip-till-next-match. We formally define both strategies and formulate theorems to characterize them. To facilitate the definition of the event selection strategies, we introduce the prefix of a pattern and the prefix of a match.

A *prefix of a pattern* $(\langle B_1, \ldots, B_k\rangle, \Theta, \tau)$ is defined as a pattern $(\langle \widehat{B}_1, \ldots, \widehat{B}_m\rangle, \widehat{\Theta}, \tau)$, where $\widehat{B}_j = B_j$ and $\widehat{B}_m \subseteq B_m$, $\widehat{B}_m \neq \emptyset$, for $j < m \leq k$, and $\widehat{\Theta}$ is the set of conditions in Θ that only involve variables from $\widehat{B}_1 \cup \cdots \cup \widehat{B}_m$. The set of all prefixes of a pattern P is denoted as $preP(P)$.

Example 6.2. Consider our running example and pattern $P_3 = (\langle\{s_1\}, \{s_2\}, \{s_3^+\}, \{s_4\}\rangle, \Theta, 100 \text{ ms})$. A prefix of P_3 is $\widehat{P}_3 = (\langle\{s_1\}, \{s_2\}\rangle, \widehat{\Theta}_2, 100 \text{ ms})$, where $\widehat{\Theta}_2 = \{s_1.S = \text{'GOOG'}, s_2.S = \text{'GOOG'}, s_1.P < s_2.P\}$. The set of all prefixes of P_3 is given as

$$
\begin{aligned}
preP(P_3) = \{&(\langle\{s_1\}\rangle, \widehat{\Theta}_1, 100 \text{ ms}),\\
&(\langle\{s_1\}, \{s_2\}\rangle, \widehat{\Theta}_2, 100 \text{ ms}),\\
&(\langle\{s_1\}, \{s_2\}, \{s_3^+\}\rangle, \widehat{\Theta}_3, 100 \text{ ms}),\\
&(\langle\{s_1\}, \{s_2\}, \{s_3^+\}, \{s_4\}\rangle, \Theta, 100 \text{ ms})\}.
\end{aligned}
$$

where

$$\begin{aligned}
\widehat{\Theta}_1 &= \{s_1.S = \text{'GOOG'}\},\\
\widehat{\Theta}_2 &= \{s_1.S = \text{'GOOG'}, s_2.S = \text{'GOOG'}, s_1.P < s_2.P\},\\
\widehat{\Theta}_3 &= \{s_1.S = \text{'GOOG'}, s_2.S = \text{'GOOG'}, s_3.S = \text{'GOOG'},\\
&\qquad s_1.P < s_2.P, s_2.P < s_3.P, prev(s_3.P) < s_3.P\}.
\end{aligned}$$

A *prefix of a match* $\{v_1/\vec{e}_1, \ldots, v_k/\vec{e}_k\}$ that binds the events $\langle e_1, \ldots, e_n\rangle$ is defined as a match $\{v_1/\vec{\widehat{e}}_1, \ldots, v_l/\vec{\widehat{e}}_l\}$ of a pattern prefix $\widehat{P} \in preP(P)$, $l \leq k$, with $\vec{\widehat{e}}_j \subseteq \vec{e}_j$, that binds the events $\langle e_1, \ldots, e_i\rangle$, $i \leq n$. The set of all prefixes of a match γ is denoted as $preM(\gamma)$.

Example 6.3. Consider match $\gamma = \{s_1/\langle \mathsf{e}_{71}\rangle,\ s_2/\langle \mathsf{e}_{74}\rangle,\ s_3/\langle \mathsf{e}_{76}\rangle,\ s_4/\langle \mathsf{e}_{77}\rangle\}$. A prefix of match γ is $\{s_1/\langle \mathsf{e}_{71}\rangle,\ s_2/\langle \mathsf{e}_{74}\rangle\}$. It satisfies the pattern prefix $\widehat{P}_3 = \langle\{s_1\}, \{s_2\}\rangle, \widehat{\Theta}_2, 100\,\text{ms})$ in the previous example. The set of all prefixes of γ is given as

$$\begin{aligned}
preM(\gamma) = \{&\{s_1/\langle \mathsf{e}_{71}\rangle\},\\
&\{s_1/\langle \mathsf{e}_{71}\rangle, s_2/\langle \mathsf{e}_{74}\rangle\},\\
&\{s_1/\langle \mathsf{e}_{71}\rangle, s_2/\langle \mathsf{e}_{74}\rangle, s_3/\langle \mathsf{e}_{76}\rangle\},\\
&\{s_1/\langle \mathsf{e}_{71}\rangle, s_2/\langle \mathsf{e}_{74}\rangle, s_3/\langle \mathsf{e}_{76}\rangle, s_4/\langle \mathsf{e}_{77}\rangle\}\}.
\end{aligned}$$

6.2.1 Formal Definition

The first event selection strategy that we formally define is skip-till-next-match as used in [4].

Definition 6.1. (*Skip-till-next-match [4]*) Let $P = (\langle B_1, \ldots, B_k\rangle, \Theta, \tau)$ be a pattern with $B = B_1 \cup \cdots \cup B_k$, E be an event stream, γ be a match that binds the events $\langle e_1, \ldots, e_n\rangle$ and $\widehat{\gamma}$ be a prefix of γ that binds the events $\langle e_1, \ldots, e_i\rangle$, $i \leq n$. A match γ conforms to the *skip-till-next-match* event selection strategy if and only if

$$\begin{aligned}
\nexists \widehat{\gamma} \in preM(\gamma) \setminus \gamma, \widehat{P} \in preP(P), v \in B, e \in E\ (&\\
e_i.T < e.T < e_{i+1}.T \wedge \widehat{\gamma} \uplus \{v/\langle e\rangle\} \text{ is a match of } \widehat{P}).&
\end{aligned} \tag{6.1}$$

A match conforming to skip-till-next-match does not contain a prefix that, if extended by an event from the event stream E which occurs after the prefix and before the chronologically next event in γ, satisfies a prefix of pattern P. From a procedural point of view, skip-till-next-match specifies that events in E are skipped until the next event is found which along with the events bound so far matches a prefix of P.

Example 6.4. Consider match $\gamma = \{s_1/\langle \mathtt{e_1}\rangle,\ s_2/\langle \mathtt{e_4}\rangle,\ s_3/\langle \mathtt{e_5}, \mathtt{e_6}\rangle,\ s_4/\langle \mathtt{e_7}\rangle\}$. The only prefix of γ that can be extended by a binding with an event that occurs after the prefix and before the chronologically next event in γ is $\widehat{\gamma} = \{s_1/\langle \mathtt{e_1}\rangle\}$. For $\widehat{\gamma}$ only prefix $\widehat{P}_3 = (\langle\{s_1\}, \{s_2\}\rangle, \widehat{\Theta}, 100\ \mathrm{ms})$ of the example pattern P_3 needs to be considered. Neither the binding $s_2/\langle \mathtt{e_2}\rangle$ nor $s_2/\langle \mathtt{e_3}\rangle$ extend $\widehat{\gamma}$ to a match of $\widehat{P}_3$. Consequently, Condition 6.1 is satisfied and the match conforms to skip-till-next-match. In contrast, match $\gamma = \{s_1/\langle \mathtt{e_{71}}\rangle,\ s_2/\langle \mathtt{e_{74}}\rangle,\ s_3/\langle \mathtt{e_{76}}\rangle,\ s_4/\langle \mathtt{e_{77}}\rangle\}$ does not conform to skip-till-next-match, because the prefix $\widehat{\gamma} = \{s_1/\langle \mathtt{e_{71}}\rangle\}$ of γ can be extended with binding $s_2/\langle \mathtt{e_{72}}\rangle$ to $\{s_1/\langle \mathtt{e_{71}}\rangle, s_2/\langle \mathtt{e_{72}}\rangle\}$, which satisfies the prefix $\widehat{P}_3 = (\langle\{s_1\}, \{s_2\}\rangle, \widehat{\Theta}, 100\ \mathrm{ms})$ of pattern P_3. The complete list of matches for P_3 that conform to skip-till-next-match is as follows:

$$\begin{array}{llll} \{\ s_1/\langle \mathtt{e_1}\rangle, & s_2/\langle \mathtt{e_4}\rangle, & s_3/\langle \mathtt{e_5}, \mathtt{e_6}\rangle, & s_4/\langle \mathtt{e_7}\rangle\ \}, \\ \{\ s_1/\langle \mathtt{e_3}\rangle, & s_2/\langle \mathtt{e_4}\rangle, & s_3/\langle \mathtt{e_5}, \mathtt{e_6}\rangle, & s_4/\langle \mathtt{e_7}\rangle\ \}, \\ \{\ s_1/\langle \mathtt{e_4}\rangle, & s_2/\langle \mathtt{e_5}\rangle, & s_3/\langle \mathtt{e_6}\rangle, & s_4/\langle \mathtt{e_7}\rangle\ \}. \end{array}$$

Next, we formally define the robust skip-till-next-match event selection strategy.

Definition 6.2. (*Robust Skip-till-next-match*) Let $P = (\langle B_1, \ldots, B_k\rangle, \Theta, \tau)$ be a pattern with $B = B_1 \cup \cdots \cup B_k$, E be an event stream, and Γ be the set of all possible matches of P in E according to Definition 3.2. Furthermore, let γ be a match that binds the events $\langle e_1, \ldots, e_n\rangle$ and $\widehat{\gamma}$ be a prefix of γ that binds the events $\langle e_1, \ldots, e_i\rangle$, $i \leq n$. A match γ conforms to the *robust skip-till-next-match* event selection strategy if and only if

$$\begin{aligned} \nexists \widehat{\gamma} \in \mathit{preM}(\gamma) \setminus \gamma, \gamma' \in \Gamma, v \in B, e \in E\ (\ & \\ & e_i.T < e.T < e_{i+1}.T \wedge (\widehat{\gamma} \uplus \{v/\langle e\rangle\}) \in \mathit{preM}(\gamma')\ ). \end{aligned} \tag{6.2}$$

A match that conforms to robust skip-till-next-match does not contain a prefix that, if extended by an event from event stream E which occurs after the prefix and before the chronologically next event in γ, corresponds to a prefix of some match in Γ. From a procedural point of view, robust skip-till-next-match specifies that events in E are skipped until the next event is found which satisfies P along with the events matched so far and the future events that will be matched.

Example 6.5. Consider match $\gamma_1 = \{s_1/\langle \mathtt{e_{71}}\rangle, s_2/\langle \mathtt{e_{74}}\rangle, s_3/\langle \mathtt{e_{76}}\rangle, s_4/\langle \mathtt{e_{77}}\rangle\}$ and the set of all possible matches of P_3 in `Stocks`,

$$\begin{array}{lllll} \Gamma = \{ & \{\ s_1/\langle \mathtt{e_1}\rangle, & s_2/\langle \mathtt{e_4}\rangle, & s_3/\langle \mathtt{e_5}, \mathtt{e_6}\rangle, & s_4/\langle \mathtt{e_7}\rangle\ \}, \\ & \{\ s_1/\langle \mathtt{e_1}\rangle, & s_2/\langle \mathtt{e_4}\rangle, & s_3/\langle \mathtt{e_5}\rangle, & s_4/\langle \mathtt{e_7}\rangle\ \}, \\ & \{\ s_1/\langle \mathtt{e_1}\rangle, & s_2/\langle \mathtt{e_4}\rangle, & s_3/\langle \mathtt{e_6}\rangle, & s_4/\langle \mathtt{e_7}\rangle\ \}, \\ & \{\ s_1/\langle \mathtt{e_3}\rangle, & s_2/\langle \mathtt{e_4}\rangle, & s_3/\langle \mathtt{e_5}, \mathtt{e_6}\rangle, & s_4/\langle \mathtt{e_7}\rangle\ \}, \\ & \{\ s_1/\langle \mathtt{e_3}\rangle, & s_2/\langle \mathtt{e_4}\rangle, & s_3/\langle \mathtt{e_5}\rangle, & s_4/\langle \mathtt{e_7}\rangle\ \}, \\ & \{\ s_1/\langle \mathtt{e_3}\rangle, & s_2/\langle \mathtt{e_4}\rangle, & s_3/\langle \mathtt{e_6}\rangle, & s_4/\langle \mathtt{e_7}\rangle\ \}, \\ & \{\ s_1/\langle \mathtt{e_4}\rangle, & s_2/\langle \mathtt{e_5}\rangle, & s_3/\langle \mathtt{e_6}\rangle, & s_4/\langle \mathtt{e_7}\rangle\ \}, \\ & \{\ s_1/\langle \mathtt{e_{71}}\rangle, & s_2/\langle \mathtt{e_{74}}\rangle, & s_3/\langle \mathtt{e_{76}}\rangle, & s_4/\langle \mathtt{e_{77}}\rangle\ \}, \\ & \{\ s_1/\langle \mathtt{e_{71}}\rangle, & s_2/\langle \mathtt{e_{74}}\rangle, & s_3/\langle \mathtt{e_{76}}\rangle, & s_4/\langle \mathtt{e_{78}}\rangle\ \} \quad \}. \end{array}$$

Match γ_1 conforms to robust skip-till-next-match, because prefix $\widehat{\gamma} = \{s_1/\langle \mathsf{e}_{71} \rangle\}$ of γ_1 extended by bindings $s_2/\langle \mathsf{e}_{72} \rangle$ or $s_2/\langle \mathsf{e}_{73} \rangle$ does not resemble a prefix of a match in Γ. Other extensions of prefixes of γ_1 with bindings are not possible. In contrast, match $\gamma_2 = \{s_1/\langle \mathsf{e}_{71} \rangle, s_2/\langle \mathsf{e}_{74} \rangle, s_3/\langle \mathsf{e}_{76} \rangle, s_4/\langle \mathsf{e}_{78} \rangle\}$ does not conform to robust skip-till-next-match, because prefix $\widehat{\gamma} = \{s_1/\langle \mathsf{e}_{71} \rangle, s_2/\langle \mathsf{e}_{74} \rangle, s_3/\langle \mathsf{e}_{76} \rangle\}$ of γ_2 can be extended by binding $s_4/\langle \mathsf{e}_{77} \rangle$ resulting in match γ_1 that is a member of $preM(\gamma_1)$. The complete list of matches for P_3 found with robust skip-till-next-match is as follows:

$$
\begin{array}{llll}
\{\ s_1/\langle \mathsf{e}_1 \rangle, & s_2/\langle \mathsf{e}_4 \rangle, & s_3/\langle \mathsf{e}_5, \mathsf{e}_6 \rangle, & s_4/\langle \mathsf{e}_7 \rangle\ \}, \\
\{\ s_1/\langle \mathsf{e}_3 \rangle, & s_2/\langle \mathsf{e}_4 \rangle, & s_3/\langle \mathsf{e}_5, \mathsf{e}_6 \rangle, & s_4/\langle \mathsf{e}_7 \rangle\ \}, \\
\{\ s_1/\langle \mathsf{e}_4 \rangle, & s_2/\langle \mathsf{e}_5 \rangle, & s_3/\langle \mathsf{e}_6 \rangle, & s_4/\langle \mathsf{e}_7 \rangle\ \}, \\
\{\ s_1/\langle \mathsf{e}_{71} \rangle, & s_2/\langle \mathsf{e}_{74} \rangle, & s_3/\langle \mathsf{e}_{76} \rangle, & s_4/\langle \mathsf{e}_{77} \rangle\ \}.
\end{array}
$$

The main difference between skip-till-next-match and robust skip-till-next-match lies in the events that disqualify a match from conforming to the event selection strategies, i.e., the event $e \in E$ in Condition 6.1 and 6.2. An event e disqualifies a match from conforming to skip-till-next-match if e belongs to a partial match that satisfies the constraints of a prefix of the pattern, but does not necessarily belong to a complete match. That is, e may be an irrelevant event that together with the events matched so far satisfies some constraints in the pattern. In contrast, a match is disqualified from conforming to robust skip-till-next-match by an event e that must belong to another complete match. No irrelevant events are involved. That is, robust skip-till-next-match is robust against irrelevant events in the event stream because it correctly recognizes them.

6.2.2 Properties

The relationship between skip-till-next-match and robust skip-till-next-match is described in the following theorem.

Theorem 6.1. Let Γ_{stnm} and Γ_{rstnm} be the matches of a pattern P in an event stream E that conform to skip-till-next-match and robust skip-till-next-match, respectively. For Γ_{stnm} and Γ_{rstnm} the following holds:

$$\Gamma_{\mathrm{stnm}} \subseteq \Gamma_{\mathrm{rstnm}}. \tag{6.3}$$

Proof. Assume a pattern P and an event stream E, and let Γ be the set of all possible matches of P in E as defined in Definition 3.2. Condition 6.3 holds, if, for any match γ of P in E, Condition 6.2 is satisfied whenever Condition 6.1 is satisfied, i.e.,

$$\gamma \in \Gamma_{\mathrm{stnm}} \rightarrow \gamma \in \Gamma_{\mathrm{rstnm}}. \tag{6.4}$$

Since the prefix of a match satisfies the prefix of a pattern by definition, whenever γ does not contain a prefix that extended as described in Condition 6.1 matches a prefix

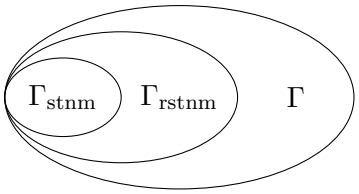


Figure 6.2: Containment of Sets of Matches.

of P (i.e., $\gamma \in \Gamma_{\text{stnm}}$), γ does not contain a prefix that extended as described in Condition 6.2 is a prefix of some match in Γ (i.e., $\gamma \in \Gamma_{\text{rstnm}}$). Thus, Condition 6.4 is satisfied for any γ. □

The set of matches that conform to robust skip-till-next-match contains all matches that conform to skip-till-next-match (see Figure 6.2). That is, an event pattern matching algorithm that complies with robust skip-till-next-match finds all matches that are found with an algorithm that complies with skip-till-next-match.

A nice property of robust skip-till-next-match is that the result set of a query can be derived from the set of all possible matches without accessing the event stream, as stated in the following theorem.

Theorem 6.2. Given only the set of all possible matches Γ of a pattern P in an event stream E, (a) the set of matches Γ_{rstnm} that conform to robust skip-till-next-match can be found, (b) the set of matches Γ_{stnm} that conform to skip-till-next-match cannot be found.

Proof. Assume the set of all possible matches Γ of a pattern P in an event stream E and the set of matches $\Gamma_{\text{rstnm}} \subseteq \Gamma$ that conform to robust skip-till-next-match as well as the set of matches $\Gamma_{\text{stnm}} \subseteq \Gamma_{\text{rstnm}}$ that conform to skip-till-next-match.

To proof (a), consider a match $\gamma \in \Gamma$. Match γ does not conform to robust skip-till-next-match if and only if Condition 6.2 is not satisfied. Condition 6.2 is not satisfied if and only if $e_i.T < e.T < e_{i+1}.T \wedge (\widehat{\gamma} \uplus \{v/\langle e \rangle\}) \in preM(\gamma')$ is satisfied. Since this term can only be satisfied by events that are contained in a match, only the events in the matches in Γ need to be considered rather than all events in E. Thus, the set of matches Γ_{rstnm} can be derived only from the set of all possible matches Γ.

Part (b) of the theorem is proofed by contradiction. Assume that it is possible to find the matches in Γ_{stnm} given only the set of all possible matches Γ. Match $\{s_1/\langle \mathsf{e}_{71} \rangle, s_2/\langle \mathsf{e}_{74} \rangle, s_3/\langle \mathsf{e}_{76} \rangle, s_4/\langle \mathsf{e}_{77} \rangle\}$ from the running example, cannot be excluded from Γ_{stnm} without considering event e_{72}. Event e_{72} is not contained in any match in Γ. This contradicts the assumption. □

Theorem 6.2 enables to derive matches that conform to robust skip-till-next-match from the result set of event pattern matching algorithms that find all possible matches of a pattern in an event stream such as ZStream [46], NEEL [44], and Event Analyzer [42]

in a post processing step without accessing the event stream an additional time. This is not possible for matches that conform to skip-till-next-match. This property of robust skip-till-next-match is useful in interactive analysis where first all possible matches are analysed and then in a following analysis step the matches should be restricted.

6.3 Automaton with Backtracking

In this section we present an automaton-based solution for the evaluation of event pattern matching using the robust skip-till-next-match event selection strategy. We begin with a basic automaton and show that under certain conditions matches are missed. Then we propose a backtracking mechanism that extends the basic automaton to find all matches according to robust skip-till-next-match. To make the discussion more concrete, we assume a SES automaton as described in Chapter 4. However, the discussion can be equally applied to other automaton-based algorithms that produce matches according to skip-till-next-match [4, 28].

Example 6.6. Figure 6.3 shows the SES automaton, represented as a directed graph, for our pattern query $P_3 = (\langle\{s_1\}, \{s_2\}, \{s_3^+\}, \{s_4\}\rangle, \Theta, 100\text{ ms})$. The automaton has states

$$Q = \{\ \emptyset, \{s_1\}, \{s_1, s_2\}, \{s_1, s_2, s_3\}, \{s_1, s_2, s_3, s_4\}\ \},$$

transitions

$$\begin{aligned}\Delta = \{\ &\delta_1 = (\emptyset, s_1, \Theta_1),\\ &\delta_2 = (\{s_1\}, s_2, \Theta_2),\\ &\delta_3 = (\{s_1, s_2\}, s_3, \Theta_3),\\ &\delta_4 = (\{s_1, s_2, s_3\}, s_3, \Theta_4),\\ &\delta_5 = (\{s_1, s_2, s_3\}, s_4, \Theta_5)\ \},\end{aligned}$$

start state $q_\text{s} = \emptyset$, and accepting state $q_f = \{s_1, s_2, s_3, s_4\}$.

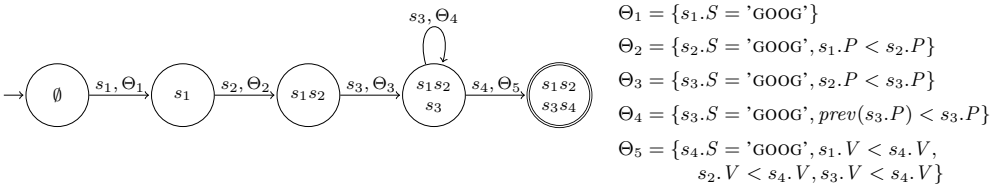


Figure 6.3: SES Automaton for $P_3 = (\langle\{s_1\}, \{s_2\}, \{s_3^+\}, \{s_4\}\rangle, \Theta, 100\text{ ms})$.

In the following examples we show that an automaton might miss matches that conform to the robust skip-till-next-match event selection strategy due to the erroneous matching of irrelevant events.

Example 6.7. Consider the automaton for pattern P_3 in Figure 6.3 and events $\mathtt{e}_{71}$, ... $\mathtt{e}_{78}$ in event stream `Stocks` (see Figure 6.1). The automaton starts in state $\emptyset$, binds $\mathtt{e}_{71}$ to s_1, and changes to state $\{s_1\}$. Then, $\mathtt{e}_{72}$ matches s_2 since the price of the Google trade is larger than in $\mathtt{e}_{71}$; the automaton changes to state $\{s_1, s_2\}$. Event $\mathtt{e}_{73}$ is skipped since it cannot match any variable in P_3. Since none of the following events, $\mathtt{e}_{74}$, $\mathtt{e}_{75}$, $\mathtt{e}_{76}$, $\mathtt{e}_{77}$, and $\mathtt{e}_{78}$, continues the upward trend as required by transition condition Θ_3, the automaton expires without producing a match. However, $\{s_1/\langle \mathtt{e}_{71}\rangle, s_2/\langle \mathtt{e}_{74}\rangle, s_3/\langle \mathtt{e}_{76}\rangle, s_4/\langle \mathtt{e}_{77}\rangle\}$ represents a valid match according to the robust skip-till-next-match strategy.

Example 6.8. Now consider again the automaton in Figure 6.3 and the events $\mathtt{e}_{71}$, ... $\mathtt{e}_{78}$ in `Stocks`, but this time without $\mathtt{e}_{72}$. The automaton starts in state $\emptyset$, binds $\mathtt{e}_{71}$ to s_1, and changes to state $\{s_1\}$. Event $\mathtt{e}_{73}$ is skipped since it cannot match any variable in P_3. Then, $\mathtt{e}_{74}$ matches s_2 and $\mathtt{e}_{75}$ as well as $\mathtt{e}_{76}$ match s_3 since the price of the Google trades are increasing; the automaton changes to state $\{s_1, s_2\}$ and to $\{s_1, s_2, s_3\}$. Since neither $\mathtt{e}_{77}$ nor $\mathtt{e}_{78}$ has a larger volume than $\mathtt{e}_{75}$ as required by Θ_5, the automaton expires without producing a match. Also here, $\{s_1/\langle \mathtt{e}_{71}\rangle, s_2/\langle \mathtt{e}_{74}\rangle, s_3/\langle \mathtt{e}_{76}\rangle, s_4/\langle \mathtt{e}_{77}\rangle\}$ represents a valid match according to the robust skip-till-next-match strategy.

The reason for the missing match in the above examples is that s_2 binds $\mathtt{e}_{72}$ in Example 6.7 and $\mathtt{e}_{75}$ in Example 6.8, since they represents an increase in the price with respect to the events matched so far. Only later on when the next Google stock trades are read, it turns out that the partial match cannot be completed, hence $\mathtt{e}_{72}$ and $\mathtt{e}_{75}$ shall be skipped as irrelevant events. By skipping $\mathtt{e}_{72}$ and binding s_2 to $\mathtt{e}_{74}$ in Example 6.7 and skipping $\mathtt{e}_{75}$ in Example 6.8 the match $\{s_1/\langle \mathtt{e}_{71}\rangle, s_2/\langle \mathtt{e}_{74}\rangle, s_3/\langle \mathtt{e}_{76}\rangle, s_4/\langle \mathtt{e}_{77}\rangle\}$ would be found.

The following theorem specifies conditions when an automaton misses matches that are valid under the robust skip-till-next-match strategy.

Theorem 6.3. Let E be an event stream and $(\langle B_1, \ldots, B_k\rangle, \Theta, \tau)$ be a pattern with variables $u \in B_i$, $v \in B_j$, $w \in B_l$, $i \leq j \leq l$ and the relationships between u and v, $\Theta^{\mathrm{r}}_{u,v} \subseteq \Theta$, are not empty. Furthermore, let γ be a match of P in E according to the robust skip-till-next match event selection strategy that contains bindings $u/\vec{e}_u$, $v/\vec{e}_v$, and $w/\vec{e}_w$ with $e_{u_1}.T < e_{v_1}.T < e_{w_1}.T$, and let $\vec{e}$ denote all events in γ. An automaton misses match γ if $\exists e \in E \setminus \vec{e}$ such that

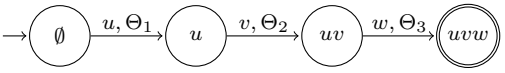
$$e_{u_1}.T < e.T < \min(e_{v_n}.T, e_{w_1}.T), \tag{6.5}$$

$$\begin{aligned}\Theta^{\mathrm{p}}_v(e) \wedge \Theta^{\mathrm{r}}_{v,v}(\langle\{e_{v_i} \in \vec{e}_v : e_{v_i}.T < e.T\} \cup \{e\}\rangle) \\ \wedge\, \Theta^{\mathrm{r}}_{u,v}(\vec{e}_u, \langle\{e_{v_i} \in \vec{e}_v : e_{v_i}.T < e.T\} \cup \{e\}\rangle).\end{aligned} \tag{6.6}$$

Proof. Assume an event stream E and a pattern $P = (\langle B_1, \ldots, B_k\rangle, \Theta, \tau)$ that consists of exactly three variables, $u \in B_i$, $v \in B_j$, and $w \in B_l$ with $i \leq j \leq l$. Furthermore, let $\gamma = \{u/\vec{e}_u, v/\vec{e}_v, w/\vec{e}_w\}$ be a match with $e_{u_1}.T < e_{v_1}.T < e_{w_1}.T$ that conforms

to the robust skip-till-next-match strategy. Finally, let $e \in E \setminus \langle \vec{e}_u \cup \vec{e}_v \cup \vec{e}_w \rangle$ be an event that satisfies Conditions 6.5 and 6.6.

An automaton N for pattern P contains the following path from the start state to the accepting state (possible looping transitions are not shown), and this path needs to be traversed to find match γ.



$$\Theta_1 = \Theta^{\mathrm{p}}_u \cup \Theta^{\mathrm{r}}_u \quad \Theta_2 = \Theta^{\mathrm{p}}_v \cup \Theta^{\mathrm{r}}_v \cup \Theta^{\mathrm{r}}_{u,v} \quad \Theta_3 = \Theta^{\mathrm{p}}_w \cup \Theta^{\mathrm{r}}_w \cup \Theta^{\mathrm{r}}_{u,w} \cup \Theta^{\mathrm{r}}_{v,w}$$

By Conditions 6.5 and 6.6, after reading event e, the automaton is in state $\{u, v\}$ since e occurs after the first event that is bound to u, satisfies all properties of v, and satisfies all relationships with the events matched to u and v so far. In this situation, N cannot reach the accepting state $\{u, v, w\}$. If N reached $\{u, v, w\}$, γ would not comply with the robust skip-till-next-match strategy (Condition 6.2) because e would occur after a prefix $\widehat{\gamma}$ of γ and before the chronologically next event in γ, and it would extend $\widehat{\gamma}$ to a prefix of the match of P that would be found if N reached the accepting state. This contradicts our assumption that γ conforms to robust skip-till-next-match. The proof can easily be generalized to more than three variables. □

Example 6.9. Consider Example 6.7. The variables s_1, s_2, and s_3 correspond to u, v, and w, the event sequences $\langle \mathtt{e_{71}} \rangle$, $\langle \mathtt{e_{74}} \rangle$, and $\langle \mathtt{e_{76}} \rangle$ to $\vec{e}_u$, $\vec{e}_v$, and $\vec{e}_w$, and event $\mathtt{e_{72}}$ corresponds to e in Theorem 6.3. Event $\mathtt{e_{72}}$ occurs after $\mathtt{e_{71}}$ and before $\mathtt{e_{74}}$ (Condition 6.5). The event sequences $\langle \mathtt{e_{71}} \rangle$ and $\langle \mathtt{e_{72}} \rangle$, bound to s_1 and s_2, respectively, satisfy the properties $\Theta^{\mathrm{p}}_{s_2} = \{s_2.S = \text{'GOOG'}\}$ and the relationships $\Theta^{\mathrm{r}}_{s_2,s_2} = \emptyset$ and $\Theta^{\mathrm{r}}_{s_1,s_2} = \{s_1.P < s_2.P\}$ (Condition 6.6).

In order to find all matches that comply with the robust skip-till-next-match event selection strategy, we extend the basic automaton with a backtracking mechanism. We reuse the concept of a match window from Definition 4.3 to buffer input events.

An automaton instance is started on the first event in the match window. If the automaton is not in the accepting state after reading all events in window W, backtracking applies. The automaton reverts the last transition. That is, it returns to the previous state and removes from match buffer β the event e_i that has been bound by the reverted transition. Then, the automaton resumes reading from the match window at event e_{i+1}, and event e_i is skipped. The backtracking mechanism allows to reclassify e_i from relevant to irrelevant.

To enable backtracking, the automaton needs to keep track of the transitions taken and the events that triggered these transitions. We propose a so-called execution tree to record transitions and events. The tree stores dependencies between different automaton instances that branched due to nondeterminism during the execution. Such information allows to stop backtracking before producing matches that do not conform to the robust skip-till-next-match strategy.

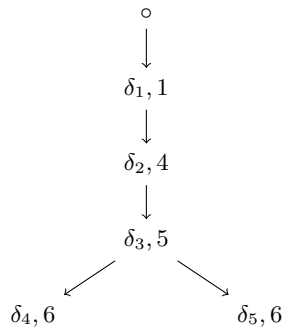


Figure 6.4: Execution Tree.

Definition 6.3. (*Execution Tree*) Let $N = (Q, \Delta, q_s, q_f)$ be an automaton for a pattern, and W be the corresponding match window over an event stream. An *execution tree*, X, is a directed, acyclic graph with nodes V and edges D, where each node has at most one incoming edge. A node represents a pair (δ, c) with $\delta \in \Delta$, $1 \leq c \leq |W|$. The special node $(\circ)$ is the root of X. An edge is a pair $(\delta_i, c_i) \rightarrow (\delta_j, c_j)$ with $(\delta_i, c_i), (\delta_j, c_j) \in V$.

A node, (δ, c), in an execution tree records that the event at position c in the match window W triggered transition δ. The direction of the edges represents the chronology of the transitions taken. Hence, a path from the root to a leave node in an execution tree represents the sequence of events matched so far. If an event triggers multiple transitions (i.e., nondeterminism occurs), X branches into multiple nodes, called *siblings*. Each match window has exactly one corresponding execution tree.

Example 6.10. Figure 6.4 shows the execution tree for the automaton in Figure 6.3 and the match window $W = \langle \mathsf{e}_{71}, \mathsf{e}_{72}, \mathsf{e}_{73}, \mathsf{e}_{74}, \mathsf{e}_{76}, \mathsf{e}_{77}, \mathsf{e}_{78} \rangle$. For the sake of exposition, we omit event e_{75} from the match window. The execution tree specifies that event $W[1] = \mathsf{e}_{71}$ triggered transition $\delta_1 = (\emptyset, s_1, \Theta_1)$, $W[4] = \mathsf{e}_{74}$ triggered transition $\delta_2 = (\{s_1\}, s_2, \Theta_2)$, and $W[5] = \mathsf{e}_{76}$ triggered transition $\delta_3 = (\{s_1, s_2\}, s_3, \Theta_3)$. At node $(\delta_3, 5)$, the tree branches into sibling nodes $(\delta_4, 6)$ and $(\delta_5, 6)$, which is due to event $W[6] = \mathsf{e}_{77}$ that triggered two transitions, $\delta_4 = (\{s_1, s_2, s_3\}, s_3, \Theta_4)$ and $\delta_5 = (\{s_1, s_2, s_3\}, s_4, \Theta_5)$.

An automaton with backtracking executes as follows. For each match window, W, an execution tree X that contains only the root node $\circ$ is created. The automaton reads the events in W one-by-one, starting at the head $W[1]$. If an input event e at position c in W triggers a transition $\delta = (q, v, \Theta_\delta)$, a binding $v/\langle e \rangle$ is added to match buffer β, the automaton instance changes from state q to state $q \cup \{v\}$, and a new leave node (δ, c) is added to X. The automaton instance keeps a pointer to the new leaf node. If e triggers multiple transitions, nondeterminism arises. For each transition but one, an automaton

instance branches from the original automaton. Each instance takes a transition and appends a leave node to the execution tree X. Since X is shared among all instances, the appended nodes, $(\delta_1, c), \ldots, (\delta_n, c)$, are siblings with distinct transitions, $\delta_i \neq \delta_j$, but equal position c. If e does not trigger any transition and the automaton instance is not in the start state, the instance stays in its current state without updating β and X.

When all events in W are processed, automaton instances that reached the accepting state contain a match in the match buffer β. The match is added to the result and the instance terminates. For an automaton instance that is not in the accepting state, backtracking applies. First, transition $\delta = (q, v, \Theta_\delta)$ and position c is retrieved from the leaf node of the execution tree X. Then the automaton steps back to state q and removes event $W[c]$ from the binding of variable v in β. The leaf node is removed from X and the instance points to the parent of the removed node. Finally, the instance resumes reading events at $W[c+1]$. If the instance does not lead to a match after reading all events from $W[c+1]$ to the end of W, backtracking applies again, etc.

Uncontrolled backtracking without an additional stop condition would lead to matches that do not conform to robust skip-till-next-match, as stated in the following lemma.

Lemma 6.1. An execution tree X leads to matches that conform to robust skip-till-next-match if and only if for all pairs of siblings, (δ_i, c_i), (δ_j, c_j), the following holds: $c_i = c_j$.

Proof. Assume that the positions c_i and c_j of two sibling nodes are different, and consider two matches, where one corresponds to a path through node (δ_i, c_i) and one to a path through node (δ_j, c_j). The two matches have a common prefix up to their parent node. The subsequent event in the two matches (i.e., the events at position c_i and c_j, respectively) are different, and one event occurs before the other. Without loss of generality, assume that the event at c_i occurs before the event at c_j. The match with the event at c_j would not satisfy Definition 6.2 of the robust skip-till-next-match strategy because it contains a prefix that, if extended by the event at position c_i, is a prefix of the match with the event at position c_i. □

According to Lemma 6.1, backtracking must be blocked when it reaches a leaf node, (δ_i, c_i), which has siblings. Backtracking at this point would replace (δ_i, c_i) with (δ_j, c_j), where $c_i < c_j$ (i.e., the event at position c_i is skipped), and the siblings would store different positions which contradicts the lemma. Hence, backtracking is blocked and node (δ_i, c_i) is removed from the tree. If backtracking reaches the last node of a set of sibling nodes, backtracking is allowed, since all other sibling nodes have already been removed and Lemma 6.1 is not violated. Another stop condition for backtracking is when the child of the root node is reached. Further backtracking at this point would revert the automaton into the start state and restart the execution at the second event of the match window not considering all events in the event stream within a time span of duration τ. To summarize: if backtracking reaches a node in X with siblings or

the child of the root node, the automaton instance removes the node from the tree and terminates.

Example 6.11. Figure 6.5 illustrates a few steps of executing the automaton in Figure 6.3. Again for the sake of exposition, we omit event e_{75} from the match window. Each step shows the triggered transitions δ_i together with the transition graph, the current state q, the match buffer β, and the execution tree X after the transition is taken. For instance, in Figure 6.5(b) the automaton instance is in state $\{s_1\}$, and e_{72} triggers the transition $(\{s_1\}, s_2, \Theta_2)$. The instance moves to state $\{s_1, s_2\}$, adds the binding $s_2/\langle \mathsf{e}_{72} \rangle$ to β, and appends node $(\delta_2, 2)$ to node $(\delta_1, 1)$ in X. In Figure 6.5(c) backtracking applies, since the automaton instance is not in the accepting state after processing the last event e_{78} in match window W. Node $(\delta_2, 2)$ is removed from X, the binding $s_2/\langle \mathsf{e}_{72} \rangle$ is removed from β, and the current state is reset to $\{s_1\}$. The execution resumes at $W[3] = \mathsf{e}_{73}$. In Figure 6.5(e), e_{77} is read again and triggers two transitions, i.e., nondeterminism arises. Two automaton instances exist and the execution tree branches. In Figure 6.5(f), e_{78} is read and W reaches its end. The automaton instance in state $\{s_1,s_2,s_3,s_4\}$ is accepted and produces match $\{s_1/\langle \mathsf{e}_{71} \rangle, s_2/\langle \mathsf{e}_{74} \rangle, s_3/\langle \mathsf{e}_{76} \rangle, s_4/\langle \mathsf{e}_{77} \rangle\}$, whereas the one in state $\{s_1,s_2,s_3\}$ is not accepted. Backtracking cannot be applied to the automaton instance, since the corresponding node $(\delta_4, 6)$ in the execution tree X has a sibling. If backtracking were applied to the instance, node $(\delta_4, 6)$ would be replaced with $(\delta_5, 7)$ which contradicts Lemma 6.1. The instance would reach the accepting state with match $\{s_1/\langle \mathsf{e}_{71} \rangle, s_2/\langle \mathsf{e}_{74} \rangle, s_3/\langle \mathsf{e}_{76} \rangle, s_4/\langle \mathsf{e}_{78} \rangle\}$ which does not conform to robust skip-till-next-match, because its prefix $\{s_1/\langle \mathsf{e}_{71} \rangle, s_2/\langle \mathsf{e}_{74} \rangle, s_3/\langle \mathsf{e}_{76} \rangle\}$ can be extended by e_{77} that occurs earlier than e_{78} which yields a prefix of the match found by the accepted instance.

6.4 Algorithm

Algorithm 4 implements automaton-based event pattern matching with backtracking. It accepts an automaton N and a match window W as input and returns the set of all matches of pattern P in W according to the robust skip-till-next-match strategy.

To enable backtracking, a set Ω of automaton instances, (q, β, x, c), is maintained, where q is the current state, β is the match buffer, x is a pointer to the leaf node in the execution tree X, and c is the position in W where to resume the execution. At the beginning, Ω is initialized with a single automaton instance that is in the start state q_s and has an empty match buffer $\emptyset$; the execution tree contains only the root node $\circ$, and the start position in W is 1. As long as Ω is not empty, the algorithm iterates over the automaton instances in Ω. For each event in W from position c to the end of W, the current instance and all instances that branched from it due to nondeterminism are processed (lines 6–18). For each automaton instance, $(q, \beta, (\delta_p, c_p), c)$, the algorithm iterates through all transitions that leave the current state q. For each transition $\delta = (q, v, \Theta_\delta)$, the condition Θ_δ is tested (line 12). If Θ_δ is satisfied, a new node (δ, i) is

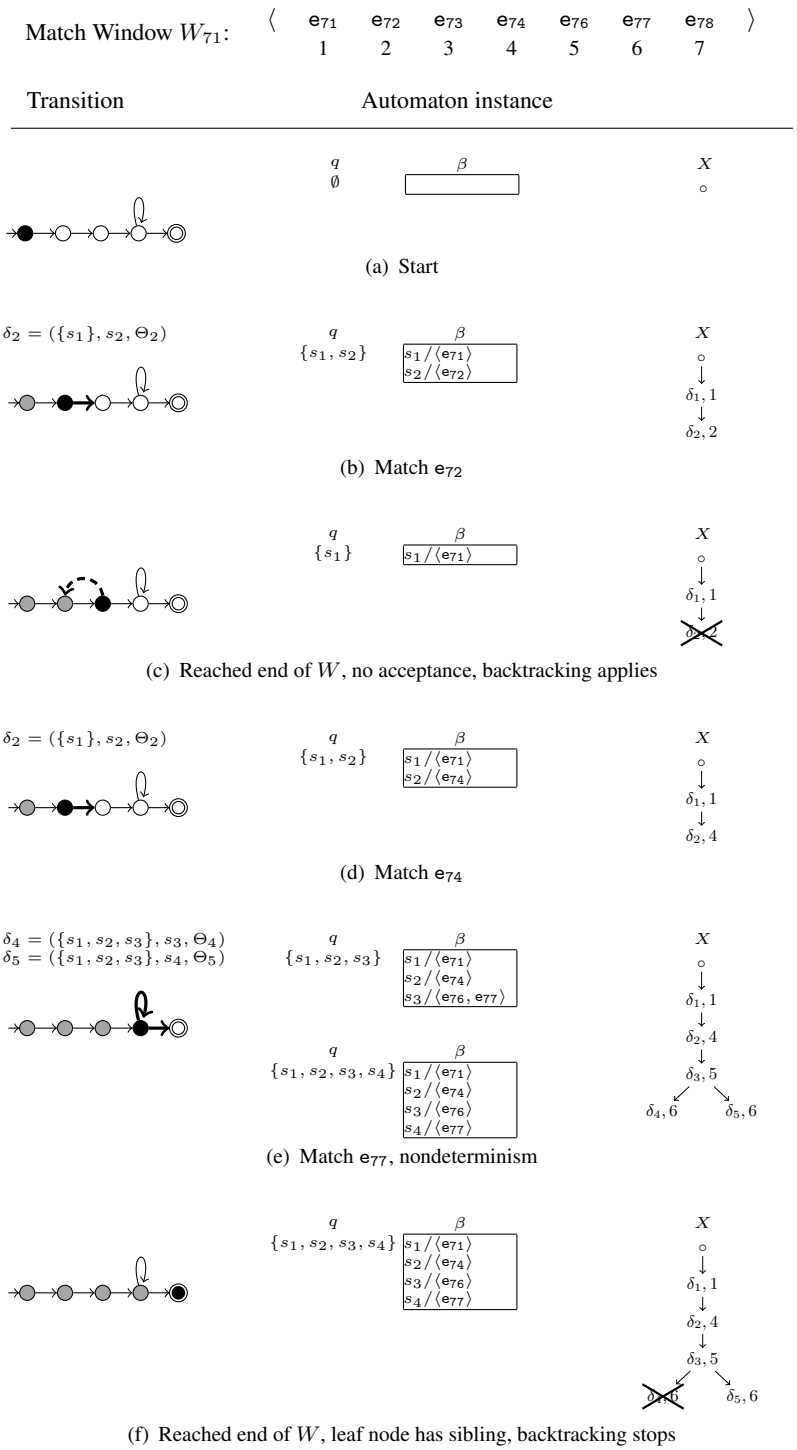


Figure 6.5: Execution of SES Automaton for $P_3 = (\langle\{s_1\}, \{s_2\}, \{s_3^+\}, \{s_4\}\rangle, \Theta, 100\text{ms})$.

Algorithm 4: `Match`(N, W)

Input: automaton $N = (Q, \Delta, q_\mathrm{s}, q_f)$, match window W
Output: set of matches

```
1  X ← (V = {∘}, D = ∅); Ω ← {(q_s, ∅, ∘, 1)};
2  while |Ω| > 0 do
3      Ω_bt ← ∅;
4      foreach (q, β, (δ_p, c_p), c) ∈ Ω do
5          Ω_W ← {(q, β, (δ_p, c_p), c)}; i ← c;
6          while i ≤ |W| and |Ω_W| > 0 do
7              Ω_e ← ∅;
8              foreach (q, β, (δ_p, c_p), c) ∈ Ω_W do
9                  Ω_δ ← ∅;
10                 foreach δ = (q_δ, v, Θ_δ) ∈ Δ s.t. q_δ = q do
11                     β' ← β ⊎ {v/⟨W[i]⟩};
12                     if I(Θ_δ β') is true then
13                         Append (δ, i) as child of (δ_p, c_p) to X;
14                         Ω_δ ← Ω_δ ∪ {(q ∪ {v}, β', (δ, i), c)};
15                 if |Ω_δ| = 0 and q_c ≠ q_s then
16                     Ω_δ ← {(q, β, (δ_p, c_p), c)};
17                 Ω_e ← Ω_e ∪ Ω_δ;
18             i ← i + 1; Ω_W ← Ω_e;
19         foreach (q, β, (δ_p, c_p), c) ∈ Ω_W do
20             if q = q_f then
21                 R ← R ∪ {β};
22             else
23                 (δ, c) ← parent of (δ_p, c_p);
24                 if (δ_p, c_p) is not child of root and has no siblings then
25                     (q_δ, v, Θ_δ) ← δ_p;
26                     Remove last event from binding v/e⃗ in β;
27                     Ω_bt ← Ω_bt ∪ {(q_δ, β, (δ, c), c_p + 1)};
28                 Remove (δ_p, c_p) from X;
29     Ω ← Ω_bt;
30 return R;
```

appended to the execution tree X with the current transition and the current position in W (line 13). Next, the automaton instance moves to the target state of the transition and the match buffer is updated with the new binding. If no transition is taken and the current state is different from the start state, the event is ignored and the automaton instance remains in the current state (lines 15–16). Once the end of the match window is reached, the algorithm iterates over the processed automaton instances. For each instance that is in the accepting state q_f, the content of its match buffer β is added to

the result set R. For each instance that is not in q_f, backtracking applies. If (δ_p, c_p) is not a child of the root and has no siblings, the automaton instance is reverted to the previous execution state and the position of the next event in W is set (lines 24–27). Node (δ_p, c_p) is removed from the execution tree X.

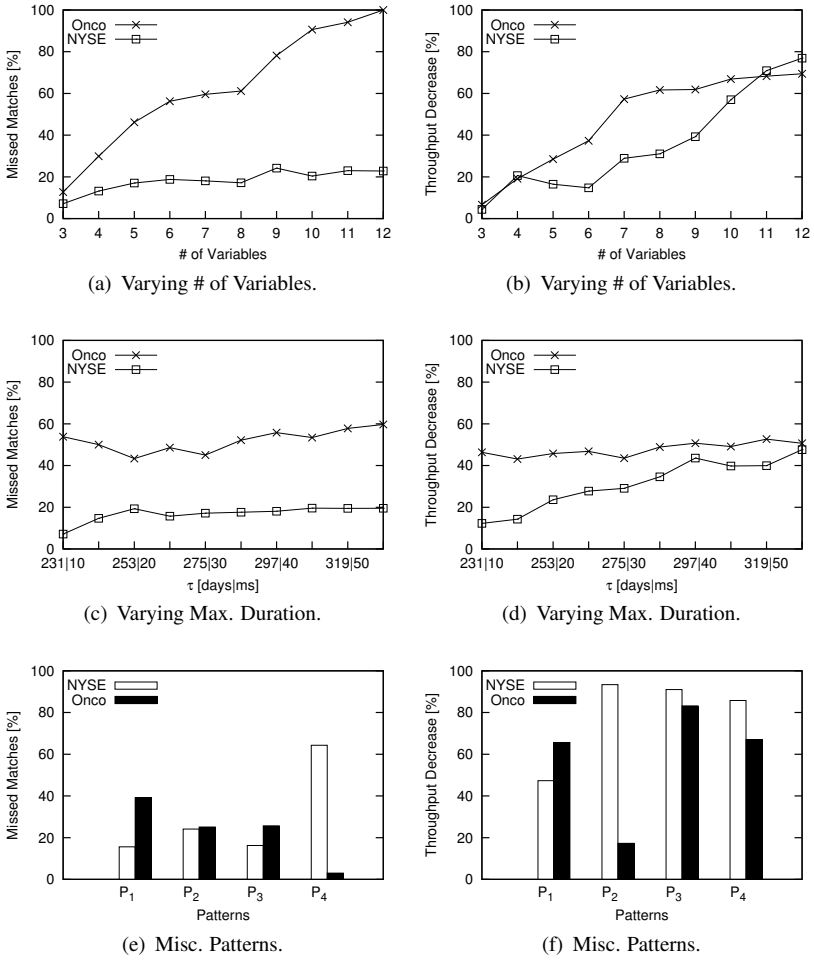
6.5 Experiments

In this section, we report the results of an empirical evaluation using real-world data. The experiments have two purposes: (1) to compare the skip-till-next-match to the robust skip-till-next-match event selection strategy and (2) to show the advantages of our backtracking mechanism over an alternative solution.

6.5.1 Setup and Data

We implemented the automaton-based algorithm with and without backtracking as well as an automaton-based solution that first computes all possible matches followed by a post processing step to eliminate matches that do not comply to robust skip-till-next-match. All algorithms are implemented in C. The event stream is stored in an Oracle database, Enterprise Edition 11.1, which is accessed over the OCI API. The experiments were performed on a PC with four AMD Opteron 285 processors with 1.8 and 2.6 GHz and 16 GB memory, on which a 64-bit Linux 2.6.32 is installed. We use two different real-world data sets. The NYSE data set contains 1M share trades in stock markets [49] of 34 hours. The Onco data set contains 341055 chemotherapy events from the Department of Hematology at the Hospital Meran-Merano.

We use different types of patterns. For the experiments with varying number of variables, we use pattern $P_{\text{vars}} = (\langle\{v_1\}, \{v_2\}, \ldots, \{v_k\}\rangle, \Theta, \tau)$, where the number of variables varies from $k = 3, \ldots, 12$. The duration is $\tau = 30$ ms with the NYSE data and $\tau = 462$ days with the Onco data. For the experiments with varying length of τ, we use pattern $P_\tau = (\langle\{v_1\}, \ldots, \{v_8\}\rangle, \Theta, \tau)$, where τ varies from 10–55 ms in steps of 5 ms with the NYSE data and from 231–319 days in steps of 11 days with the Onco data. The variables in both patterns specify events with a downwards trend in one attribute (Google stock trade price and white blood cell count, respectively). Furthermore, we use the following four miscellaneous patterns. Pattern $P_1 = (\langle\{v_1\}, \ldots, \{v_{10}^+\}\rangle, \Theta, \tau)$ finds 10 or more events with increasing values in one attribute. Pattern $P_2 = (\langle\{v_1\}, \ldots, \{v_5^+\}, \{v_6^+\}\rangle, \Theta, \tau)$ finds five or more events with increasing values in one attribute followed by one or more events that may cause nondeterminism. Pattern $P_3 = (\langle\{v_1^+\}, \{v_2\}, \{v_3^+\}, \{v_4\}, \{v_5^+\}, \{v_6\}, \{v_7^+\}\rangle, \Theta, \tau)$ roughly resembles the double-bottom pattern common in stock market analysis [53], i.e., decreasing values (v_1,v_2) followed by increasing values (v_2,v_3,v_4), then again decreasing values (v_4,v_5,v_6), and finally increasing values (v_6,v_7). Pattern $P_4 = (\langle\{v_1, w_1\}, \{v_2, w_2\}, \{v_3, w_3\}, \{v_4, w_4\}, \{v_5^+, w_5^+\}\rangle, \Theta, \tau)$ finds five or more events of two distinct types, both with increasing values in one attribute that occur interleaved. In $P_1, \ldots, P_4$ we use



(a) Varying # of Variables. (b) Varying # of Variables.

(c) Varying Max. Duration. (d) Varying Max. Duration.

(e) Misc. Patterns. (f) Misc. Patterns.

Figure 6.6: Skip-till-next-match vs. Robust Skip-till-next-match

the Google stock trade price and the white blood cell count, plus the Apple stock trade price and the red blood cell count in P_4. With the NYSE data the maximal duration, τ, is 30 ms in P_1 and P_2, 20 ms in P_3, and 50 ms in P_4. With the Onco data τ is 462 days in P_1, 231 days in P_2, 308 days in P_3 and 77 days in P_4.

6.5.2 Skip-till-next-match vs. Robust Skip-till-next-match

We compare the skip-till-next-match strategy (basic automaton) to the robust skip-till-next-match strategy (automaton with backtracking). Our hypothesis is that the former strategy finds substantially more matches than the latter one, though at the cost of a smaller throughput. We measure the number of matches missed with skip-till-next-match as the percentage of the number of matches found with robust skip-till-next-

		NYSE	Onco
P_1	bt	1	1
	all+pp	731331	3473
P_2	bt	2	10
	all+pp	1017279	66877
P_3	bt	39	32
	all+pp	94376	1904820
P_4	bt	1	1
	all+pp	168534	333950

Table 6.1: Bt vs. All+pp – Intermediate Matches for Misc. Patterns

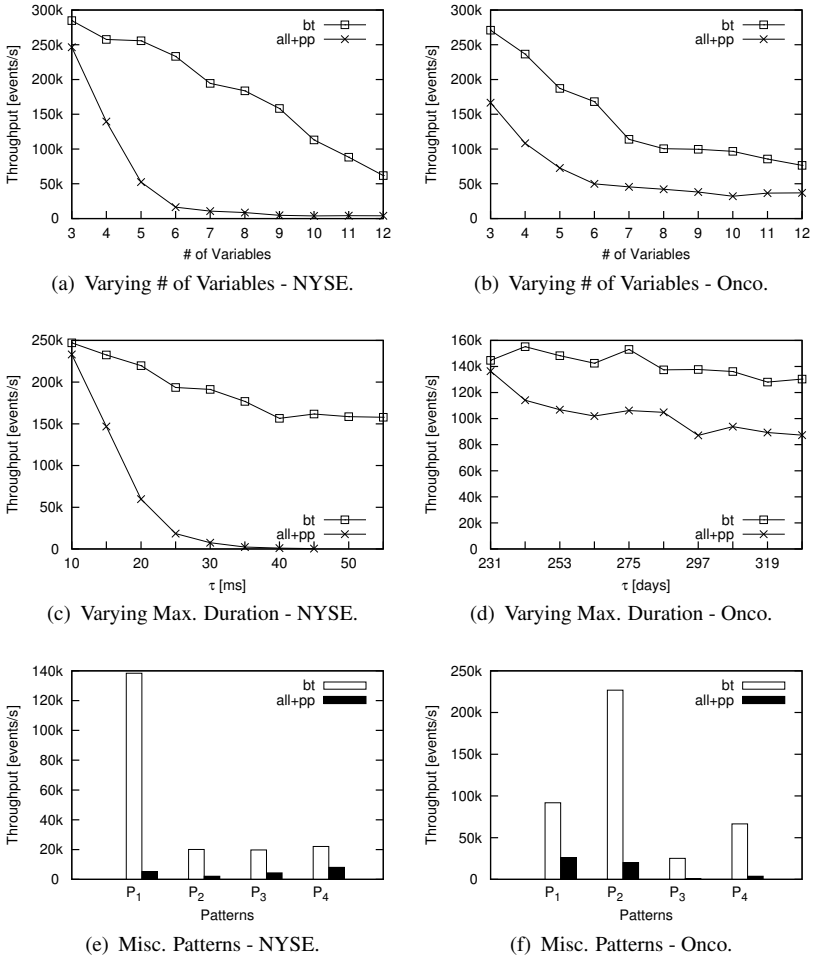
match. Similar, the decrease of the throughput with robust skip-till-next-match is measured as the percentage of the throughput with skip-till-next-match.

Figure 6.6 shows the missed matches with skip-till-next-match in the left column and the throughput decrease in the right column. The first observation is that the percentage of missed matches can be substantial. On the other side, the throughput can also decrease substantially. The second observation is that the percentage of missed matches increases with the number of variables in the pattern, whereas it is roughly constant with increasing duration τ. The reason is that with increasing number of variables and constant τ, the number of matches decreases while the possibilities of missing matches increase. With increasing τ and constant number of variables, both, the number of matches and the possibilities of missing a match increase.

6.5.3 Backtracking vs. Find All + Post Processing

In this experiment, we compare the algorithm with backtracking (bt) to an automaton-based event pattern matching algorithm that finds all possible matches and eliminates the matches that do not conform to robust skip-till-next-match in a post processing step (all+pp). Both algorithms find the same matches. Our hypothesis is that bt produces a higher throughput and less intermediate results than all+pp. We measure the throughput and the maximal number of intermediate matches per match window. Intermediate matches are accepted automaton instances. In bt accepted automaton instances are already the final result, whereas in all+pp they still need to be filtered in the post processing step.

Figure 6.7 shows the throughput of both algorithms, and Figure 6.8 shows the maximal number of intermediate matches. In Figures 6.7(c) and 6.8(c) the curve for all+pp stops at 45 ms, because we had to abort the experiment due to excessive memory swapping. The first observation is that the algorithm with backtracking produces a higher



(a) Varying # of Variables - NYSE.

(b) Varying # of Variables - Onco.

(c) Varying Max. Duration - NYSE.

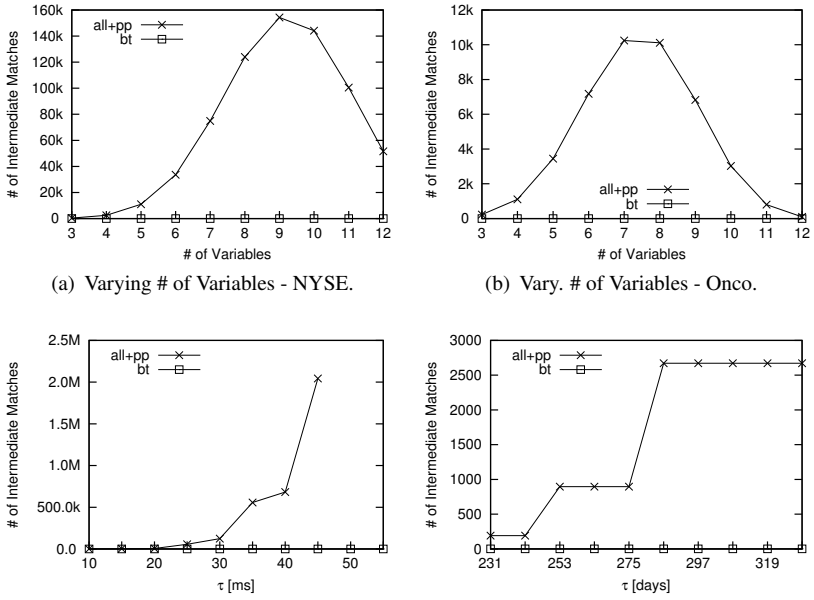
(d) Varying Max. Duration - Onco.

(e) Misc. Patterns - NYSE.

(f) Misc. Patterns - Onco.

Figure 6.7: Bt vs. All+pp – Throughput.

throughput in all experiments. For some patterns the throughput differs more than two orders of magnitude as in Figures 6.7(a) and 6.7(c). The second observation is that all+pp produces orders of magnitudes more intermediate matches than bt. This can especially be seen in Table 6.1, that contains the maximal number of intermediate matches per match window for the miscellaneous patterns. The algorithm bt produces between three and five orders of magnitude less intermediate matches than all+pp.



(a) Varying # of Variables - NYSE.

(b) Vary. # of Variables - Onco.

(c) Varying Max. Duration - NYSE.

(d) Vary. Max. Duration - Onco.

Figure 6.8: Bt vs. All+pp – Intermediate Matches.

6.6 Summary

In this chapter, we proposed a robust skip-till-next-match event selection strategy that improves skipping of noise in event pattern matching and finds matches which are missed by skip-till-next-match due to its greedy behavior. In contrast to skip-till-next-match, robust skip-till-next-match does not fail to identify irrelevant events in the event stream that prevent the detection of matches. To achieve this, all constraints in the pattern together with a complete match are considered when determining whether input events are relevant or irrelevant. We formally defined the skip-till-next-match strategy and the robust skip-till-next-match strategy. We proposed a backtracking mechanism that extends automaton-based event pattern matching to find all matches according to robust skip-till-next-match. We conducted extensive experiments using real-world data to show the effectiveness of our approach. The results show that the number of missed matches with skip-till-next-match can be quite substantial. In terms of runtime, our approach outperforms an alternative solution that first produces all possible matches followed by a post processing step.

CHAPTER 7

Conclusion

In this thesis we presented a new event pattern matching problem, termed sequenced event set (SES) pattern matching, that specifies a sequence of sets of events (SES pattern), rather than a sequence of single events as in previous work. While the order of the input events that match with the same event set is irrelevant, i.e., any permutation of the input events is matched, the order of the input events that match with distinct sets must strictly adhere to the order of the sets in the pattern. We introduced and formally defined the SES pattern matching problem. To solve the SES pattern matching problem, we presented SES automata and a corresponding algorithm. A further improvement of the evaluation of SES pattern queries is achieved by a two-phase evaluation strategy that consists of a cheap preprocessing phase followed by the more expensive pattern matching phase. We proposed a solution to improve the skipping of noise in the input stream during event pattern matching through a robust skip-till-next-match event selection strategy.

For query evaluation, a SES pattern is translated into a SES automaton, and the automaton is executed. A SES automaton is a nondeterministic finite state automaton enriched with a match buffer that collects bindings during the execution of the automaton. We analysed the runtime complexity of the SES automaton-based algorithm, where we distinguished four cases considering different patterns. The runtime depends on the number of variables and the amount of Kleene plus in the pattern as well as on the number of events in a match window. Regarding the size of the event stream, the runtime is linear. In extensive experiments, we showed that our algorithm clearly outperforms a brute force approach that matches sequences of sets of events by using a set of automata, each of which matches a sequence of single events. The brute force algorithm essentially corresponds to straightforward extensions of the automata

in [4, 28, 31]. Furthermore, we validated experimentally the results from our runtime complexity analysis.

In the two-phase evaluation strategy for event pattern matching, the preprocessing phase is very cheap compared to the expensive pattern matching phase. In the preprocessing phase, incoming events are buffered in a match window, which allows to apply different pruning techniques, such as filtering, partitioning, and testing for necessary match conditions, and aids a lazy evaluation of a pattern matching algorithm. Our evaluation strategy is general enough to be used with other existing event pattern matching algorithms besides SES automata. We conducted extensive experiments using two real-world data sets and two existing event pattern matching algorithms, more precisely our SES automaton-based algorithm that finds matches according to the skip-till-next-match event selection strategy and ZStream [46] that is based on join-trees and finds all possible matches (skip-till-any-match). The results show that our framework significantly increases the throughput for both algorithms.

We proposed a robust skip-till-next-match event selection strategy to improve the skipping of noise in the input stream. Skip-till-next-match greedily matches incoming events if they satisfy the constraints in the pattern along with the events matched so far; events that violate the constraints are eliminated as noise. Due to this greedy behavior, skip-till-next-match can miss matches that satisfy the pattern query. More specifically, a match is not found if an input event occurs within the time span of the match, satisfies the constraints in the pattern together with the current partial match consisting of the events matched so far, but violates contraints with future events yet to be matched. Instead of considering only past events for the identification of noise, we consider all constraints in the pattern together with a complete match. To compute pattern queries with the robust skip-till-next-match event selection strategy, we presented a backtracking mechanism for automaton-based event pattern matching algorithms [4, 28]. We conducted extensive experiments using real-world data to show the effectiveness of our approach. The results show that the number of missed matches with skip-till-next-match can be quite substantial. In terms of runtime, our approach outperforms an alternative solution that first produces all possible matches followed by a post processing step to filter out non-compliant matches.

Future Work. Previous work including ours used the skip-till-next-match event selection strategy to skip noise and to restrict the result set to a subset of all possible matches of a pattern in an event stream. As we showed in our work, skip-till-next-match might fail to identify events as noise and therefore can miss matches that satisfy the pattern query. Furthermore, skip-till-next-match does not guarantee to find the most useful matches. For example, if the pattern specifies multiple stock trades with a price-increasing trend, skip-till-next-match does not guarantee to find matches with the most price-increasing trades within the time span specified in the pattern. However, such matches might be the most useful ones for the analysis of event streams. It would be

interesting future work to investigate an event selection strategy that, while skipping noise, finds the most useful matches in an event stream. This work would consist of the formal definition of the event selection strategy as well as the development of algorithms and data structures that efficiently find matches according to such a strategy.

Regarding optimizations and extensions of the algorithms presented in this thesis, we plan to work on the following aspects.

First, the number of states in a SES automaton tends to be exponential in the size of the pattern because there exist a state for each subset of the sets specified in a pattern. The current SES automaton implementation maps each state to a distinct data structure in memory. This can be problematic when multiple SES automata are executed simultaneously, which might happen in a multi-user environment. Therefore, we plan to develop algorithms and data structures for SES automata that allow a significant reduction of the space requirements.

Second, the necessary match conditions presented in the two-phase evaluation strategy are limited to the skip-till-next-match and skip-till-any-match event selection strategies. Possible future work is the adaptation of the necessary match conditions and their testing to the (partitioned) contiguity event selection strategy.

Third, the main drawback of the backtracking mechanism for the robust skip-till-next-match event selection strategy is the decrease in the throughput. Future work will consist of the investigation of runtime optimizations for the backtracking mechanism.

Fourth, in this thesis, we considered streams with events that are totally ordered by their occurrence time. Simultaneous events are not allowed. However, event streams with simultaneous events are encountered in real-world applications, for instance, when events originate from different sources or when the granularity of the occurrence time in the events is too coarse. Simultaneous events in the event stream can cause matches not to be found during the matching process if they are not properly processed. We plan to develop event pattern matching algorithms that consider simultaneous events based on the methods presented in this thesis.

Bibliography

[1] D. J. Abadi, Y. Ahmad, M. Balazinska, M. Cherniack, J. hyon Hwang, W. Lindner, A. S. Maskey, E. Rasin, E. Ryvkina, N. Tatbul, Y. Xing, and S. Zdonik. The design of the Borealis stream processing engine. In *Proceedings of the 2nd Biennal Conference on Innovative Data Systems Research (CIDR '05)*, pages 277–289, 2005.

[2] D. J. Abadi, D. Carney, U. Çetintemel, M. Cherniack, C. Convey, S. Lee, M. Stonebraker, N. Tatbul, and S. Zdonik. Aurora: a new model and architecture for data stream management. *The VLDB Journal*, 12(2):120–139, 2003.

[3] A. Adi and O. Etzion. Amit - the situation manager. *The VLDB Journal*, 13(2):177–203, 2004.

[4] 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, 2008.

[5] M. K. Aguilera, R. E. Strom, D. C. Sturman, M. Astley, and T. D. Chandra. Matching events in a content-based subscription system. In *Proceedings of the 8th annual ACM symposium on Principles of distributed computing (PODC '99)*, pages 53–61, 1999.

[6] M. Akdere, U. Çetintemel, and N. Tatbul. Plan-based complex event detection across distributed sources. *Proceedings of the VLDB Endowment*, 1(1):66–77, 2008.

[7] M. Altinel and M. J. Franklin. Efficient filtering of XML documents for selective dissemination of information. In *Proceedings of the 26th International Conference on Very Large Data Bases (VLDB '00)*, pages 53–64, 2000.

[8] A. Arasu, S. Babu, and J. Widom. CQL: A language for continuous queries over streams and relations. In *Proceedings of the 9th International Workshop on Database Programming Languages (DBPL '03)*, pages 1–19, 2003.

[9] A. Arasu, S. Babu, and J. Widom. The CQL continuous query language: semantic foundations and query execution. *The VLDB Journal*, 15(2):121–142, 2006.

[10] R. Avnur and J. M. Hellerstein. Eddies: continuously adaptive query processing. In *Proceedings of the 2000 ACM SIGMOD international conference on Management of data (SIGMOD '00)*, pages 261–272, 2000.

[11] B. Babcock, S. Babu, M. Datar, R. Motwani, and J. Widom. Models and issues in data stream systems. In *Proceedings of the 21st ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems (PODS '02)*, pages 1–16, 2002.

[12] S. Babu and J. Widom. Continuous queries over data streams. *SIGMOD Rec.*, 30(3):109–120, 2001.

[13] Y. Bai, H. Thakkar, H. Wang, C. Luo, and C. Zaniolo. A data stream language and system designed for power and extensibility. In *Proceedings of the 15th ACM international conference on Information and knowledge management (CIKM '06)*, pages 337–346, 2006.

[14] R. S. Barga, J. Goldstein, M. H. Ali, and M. Hong. Consistent streaming through time: A vision for event stream processing. In *Proceedings of the 3rd Biennal Conference on Innovative Data Systems Research (CIDR '07)*, pages 363–374, 2007.

[15] L. Brenna, A. Demers, J. Gehrke, M. Hong, J. Ossher, B. Panda, M. Riedewald, M. Thatte, and W. White. Cayuga: a high-performance event processing engine. In *Proceedings of the 2007 ACM SIGMOD international conference on Management of data (SIGMOD '07)*, pages 1100–1102, 2007.

[16] L. Brenna, J. Gehrke, M. Hong, and D. Johansen. Distributed event stream processing with non-deterministic finite automata. In *Proceedings of the 3rd ACM International Conference on Distributed Event-Based Systems (DEBS '09)*, pages 3:1–3:12, 2009.

[17] D. Carney, U. Çetintemel, M. Cherniack, C. Convey, S. Lee, G. Seidman, M. Stonebraker, N. Tatbul, and S. Zdonik. Monitoring streams: a new class of data management applications. In *Proceedings of the 28th international conference on Very Large Data Bases (VLDB '02)*, pages 215–226, 2002.

[18] A. Carzaniga and A. L. Wolf. Forwarding in a content-based network. In *Proceedings of the 2003 conference on Applications, technologies, architectures, and protocols for computer communications (SIGCOMM '03)*, pages 163–174, 2003.

[19] S. Chakravarthy, V. Krishnaprasad, E. Anwar, and S.-K. Kim. Composite events for active databases: Semantics, contexts and detection. In *Proceedings of the 20th International Conference on Very Large Data Bases (VLDB '94)*, pages 606–617, 1994.

[20] B. Chandramouli, J. Goldstein, and D. Maier. High-performance dynamic pattern matching over disordered streams. *Proceedings of the VLDB Endowment*, 3(1-2):220–231, 2010.

[21] S. Chandrasekaran, O. Cooper, A. Deshpande, M. J. Franklin, J. M. Hellerstein, W. Hong, S. Krishnamurthy, S. Madden, V. Raman, F. Reiss, and M. A. Shah. TelegraphCQ: Continuous dataflow processing for an uncertain world. In *Proceedings of the 1st Biennal Conference on Innovative Data Systems Research (CIDR '03)*, 2003.

[22] J. Chen, D. J. DeWitt, F. Tian, and Y. Wang. NiagaraCQ: a scalable continuous query system for internet databases. In *Proceedings of the 2000 ACM SIGMOD international conference on Management of data (SIGMOD '00)*, pages 379–390, 2000.

[23] C. Cranor, Y. Gao, T. Johnson, V. Shkapenyuk, and O. Spatscheck. Gigascope: high performance network monitoring with an sql interface. In *Proceedings of the 2002 ACM SIGMOD international conference on Management of data (SIGMOD '02)*, pages 623–623, 2002.

[24] C. Cranor, T. Johnson, O. Spataschek, and V. Shkapenyuk. Gigascope: a stream database for network applications. In *Proceedings of the 2003 ACM SIGMOD international conference on Management of data (SIGMOD '03)*, pages 647–651, 2003.

[25] G. Cugola and A. Margara. Raced: an adaptive middleware for complex event detection. In *Proceedings of the 8th International Workshop on Adaptive and Reflective MIddleware (ARM '09)*, pages 5:1–5:6, 2009.

[26] G. Cugola and A. Margara. Complex event processing with t-rex. *J. Syst. Softw.*, 85(8):1709–1728, 2012.

[27] G. Cugola and A. Margara. Processing flows of information: From data stream to complex event processing. *ACM Comput. Surv.*, 44(3):15:1–15:62, 2012.

[28] A. J. Demers, J. Gehrke, M. Hong, M. Riedewald, and W. M. White. Towards expressive publish/subscribe systems. In *Proceedings of the 10th international conference on Extending in Database Technology (EDBT '06)*, pages 627–644, 2006.

[29] A. J. Demers, J. Gehrke, B. Panda, M. Riedewald, V. Sharma, and W. M. White. Cayuga: A general purpose event monitoring system. In *Proceedings of the 3rd Biennal Conference on Innovative Data Systems Research (CIDR '07)*, pages 412–422, 2007.

[30] N. Dindar, P. M. Fischer, M. Soner, and N. Tatbul. Efficiently correlating complex events over live and archived data streams. In *Proceedings of the 5th ACM International Conference on Distributed Event-Based Systems (DEBS '11)*, pages 243–254, 2011.

[31] N. Dindar, B. Güç, P. Lau, A. Ozal, M. Soner, and N. Tatbul. Dejavu: declarative pattern matching over live and archived streams of events. In *Proceedings of the 2009 ACM SIGMOD international conference on Management of data (SIGMOD '09)*, pages 1023–1026, 2009.

[32] L. Ding, S. Chen, E. A. Rundensteiner, J. Tatemura, W.-P. Hsiung, and K. S. Candan. Runtime semantic query optimization for event stream processing. In *Proceedings of the 2008 IEEE 24th International Conference on Data Engineering (ICDE '08)*, pages 676–685, 2008.

[33] L. Ding, K. Works, and E. A. Rundensteiner. Semantic stream query optimization exploiting dynamic metadata. In *Proceedings of the 2011 IEEE 27th International Conference on Data (ICDE '11)*, pages 111–122, 2011.

[34] EsperTech. `http://www.espertech.com`, retrieved on 2012-11-06.

[35] O. Etzion and P. Niblett. *Event Processing in Action*. Manning Publications, 2010.

[36] P. T. Eugster, P. A. Felber, R. Guerraoui, and A.-M. Kermarrec. The many faces of publish/subscribe. *ACM Comput. Surv.*, 35(2):114–131, 2003.

[37] F. Fabret, H. A. Jacobsen, F. Llirbat, J. Pereira, K. A. Ross, and D. Shasha. Filtering algorithms and implementation for very fast publish/subscribe systems. In *Proceedings of the 2001 ACM SIGMOD international conference on Management of data (SIGMOD '01)*, pages 115–126, 2001.

[38] P. M. Fischer, K. S. Esmaili, and R. J. Miller. Stream schema: providing and exploiting static metadata for data stream processing. In *Proceedings of the 13th International Conference on Extending Database Technology (EDBT '10)*, pages 207–218, 2010.

[39] S. Gatziu and K. R. Dittrich. Events in an active object-oriented database systems. In *Rules in Database Systems*, pages 23–39, 1993.

[40] N. H. Gehani, H. V. Jagadish, and O. Shmueli. Composite event specification in active databases: Model & implementation. In *Proceedings of the 18th International Conference on Very Large Data Bases (VLDB '92)*, pages 327–338, 1992.

[41] D. Gyllstrom, J. Agrawal, Y. Diao, and N. Immerman. On supporting kleene closure over event streams. In *Proceedings of the 2008 IEEE 24th International Conference on Data Engineering (ICDE '08)*, pages 1391–1393, 2008.

[42] L. Harada and Y. Hotta. Order checking in a CPOE using event analyzer. In *Proceedings of the 14th ACM international conference on Information and knowledge management (CIKM '05)*, pages 549–555, 2005.

[43] N. Khoussainova, M. Balazinska, and D. Suciu. Probabilistic event extraction from rfid data. In *Proceedings of the 2008 IEEE 24th International Conference on Data Engineering (ICDE '08)*, pages 1480 –1482, 2008.

[44] M. Liu, E. Rundensteiner, D. Dougherty, C. Gupta, S. Wang, I. Ari, and A. Mehta. High-performance nested cep query processing over event streams. In *Proceedings of the 2011 IEEE 27th International Conference on Data Engineering (ICDE '11)*, pages 123 –134, 2011.

[45] Y. Magid, G. Sharon, S. Arcushin, I. Ben-Harrush, and E. Rabinovich. Industry experience with the IBM active middleware technology (AMiT) complex event processing engine. In *Proceedings of the 4th ACM International Conference on Distributed Event-Based Systems (DEBS '10)*, pages 140–149, 2010.

[46] Y. Mei and S. Madden. Zstream: A cost-based query processor for adaptively detecting composite events. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of data (SIGMOD '09)*, pages 193–206, 2009.

[47] R. Meo, G. Psaila, and S. Ceri. Composite events in chimera. In *Proceedings of the 5th International Conference on Extending Database Technology (EDBT '96)*, pages 56–76, 1996.

[48] R. Motwani, J. Widom, A. Arasu, B. Babcock, S. Babu, M. Datar, G. S. Manku, C. Olston, J. Rosenstein, and R. Varma. Query processing, approximation, and resource management in a data stream management system. In *Proceedings of the 1st Biennal Conference on Innovative Data Systems Research (CIDR '03)*, 2003.

[49] NYSE. `http://www.nyxdata.com`, retrieved on 2012-11-06.

[50] Oracle Event Processing. `http://www.oracle.com/technetwork/middleware/complex-event-processing/overview/index.html`, retrieved on 2012-11-06.

[51] N. W. Paton and O. Díaz. Active database systems. *ACM Comput. Surv.*, 31(1):63–103, 1999.

[52] R. Sadri, C. Zaniolo, A. Zarkesh, and J. Adibi. Optimization of sequence queries in database systems. In *Proceedings of the 20th ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems (PODS '01)*, pages 71–81, 2001.

[53] R. Sadri, C. Zaniolo, A. Zarkesh, and J. Adibi. Expressing and optimizing sequence queries in database systems. *ACM Trans. Database Syst.*, 29(2):282–318, 2004.

[54] SAP Sybase Event Stream Processor. `http://www.sybase.com/products/financialservicessolutions/complex-event-processing`, retrieved on 2012-11-06.

[55] N. P. Schultz-Møller, M. Migliavacca, and P. Pietzuch. Distributed complex event processing with query rewriting. In *Proceedings of the 3rd ACM International Conference on Distributed Event-Based Systems (DEBS '09)*, pages 4:1–4:12, 2009.

[56] P. Seshadri, M. Livny, and R. Ramakrishnan. Seq: A model for sequence databases. In *Proceedings of the 11th International Conference on Data Engineering (ICDE '95)*, pages 232–239, 1995.

[57] P. Seshadri, M. Livny, and R. Ramakrishnan. The design and implementation of a sequence database system. In *Proceedings of the 22th International Conference on Very Large Data Bases (VLDB '96)*, pages 99–110, 1996.

[58] StreamBase. `http://www.streambase.com`, retrieved on 2012-11-06.

[59] 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, 2007.

[60] E. Wu, Y. Diao, and S. Rizvi. High-performance complex event processing over streams. In *Proceedings of the 2006 ACM SIGMOD international conference on Management of data (SIGMOD '06)*, pages 407–418, 2006.

[61] F. Zemke, A. Witkowski, M. Cherniak, and L. Colby. Pattern matching in sequences of rows. Technical report, 2007.

[62] D. Zimmer and R. Unland. On the semantics of complex events in active database management systems. In *Proceedings of the 15th International Conference on Data Engineering (ICDE '99)*, page 392, 1999.

Index