INSTITUT FÜR INFORMATIK
DER LUDWIG–MAXIMILIANS–UNIVERSITÄT MÜNCHEN

Report for German Research Foundation

A Method for Semantic Optimization of Complex Event Processing

Olga Poppe

Project manager:

Prof. Dr. François Bry

October 6, 2011

Abstract

Semantic optimization of database queries, i.e. the use of metadata for query optimization, is well investigated and has led to significant performance gains. This report shows that semantic optimization of database systems is applicable to complex event processing (CEP) and has even greater potential in this field.

However, in CEP in contrast to database systems, the validity of data and metadata, and the evaluation of queries are dependent from particular time intervals or application states. Many event-based applications have numerous involved workflow determined constraints.

For these reasons, a new kind of CEP metadata, called Instantiating Hierarchical Timed Automata (IHTA), is formally defined in this report. IHTA formalize application states very naturally. They capture complex causal, temporal, cardinality, and data dependencies between events and states. They are modular and represent an arbitrary number of concurrent processes.

To keep IHTA readable, only the workflow determined part of application semantics is captured by them. The rest of the application specific knowledge is expressed by constraints in the Event Stream Constraint Language (ESCL) developed in this work. ESCL constraints are short but expressive first-order logic formulas capturing causal, temporal, cardinality, data, and spatial dependencies between events and states. ESCL has strong formal foundations. Its declarative semantics is defined very similarly to the Tarski model theory. The operational semantics of ESCL is the algorithm semantically rewriting CEP queries with respect to ESCL constraints which is the adaption of the respective algorithm for database systems [31] to CEP. (IHTA will be transformed into ESCL constraints to be used for semantic query optimization by the same algorithm. This transformation is still subject for future work.)

To reduce the number of constraints which must be specified by the user in ESCL directly, ESCL cardinality constraints are propagated from base events (states) to the derived events (states) with respect to the queries deriving them. Propagation of other

kinds of ESCL constraints is to be investigated.

Acknowledgements

First and most important I would like to thank Prof. Dr. François Bry for his supervision during these two years. The topic of this work and many of the ideas described in this report come from him. He suggested many improvements and extensions of this work. Discussions with him were pathbraking and encouraging.

I kindly thank Sandro Giessl for his dedicated work on the formal definition of the Instantiating Hierarchical Timed Automata and its comparison with the existing models for application semantics.

I would also like to thank Son Thanh Dang for his strong-willed work on the propagation of ESCL cardinality constraints with respect to CEP queries.

This research is founded by the German Research Foundation (Deutsche Forschungsgemeinschaft) within the project "QONCEPT – Semantic Query Optimization in CEP Technologies" under reference number BR 2355/1-1.

Contents

Chapter 1

Introduction

Section 1.1 gives an overview of the presented approach. Section 1.2 describes the contributions and the organization of this report.

1.1 Overview of the Approach

Semantic optimization of database queries, i.e. the use of metadata for query optimization, is well investigated and has led to significant performance gains. This report shows that semantic optimization of database systems is applicable to complex event processing (CEP) when regarding events as data and complex event specifications as queries. Semantic optimization of CEP queries has even greater potential than semantic optimization of database queries since querying a stream is fundamentally different from querying a database.

Indeed, database queries are usually ad hoc. They are evaluated against known and finite data which is completely available when queries are put. All answers to a query are usually computed only once and expected almost immediately after the query is put. For these reasons, static semantic optimization of database queries must be efficient. In database systems, data is usually optimized, not the queries.

In contrast to that, CEP applications permanently evaluate a known set of standing queries against event data arriving on potentially infinite streams. Event data is never available at once and, therefore, query answers are computed in multiple (arbitrarily many) evaluation steps. For these reasons, static semantic optimization of CEP queries may be inefficient if it happens before events are available and the queries can be evaluated. Besides, repeated evaluation of CEP queries over a long period of time is worth their expensive static semantic optimization. In CEP, queries are usually optimized, not the

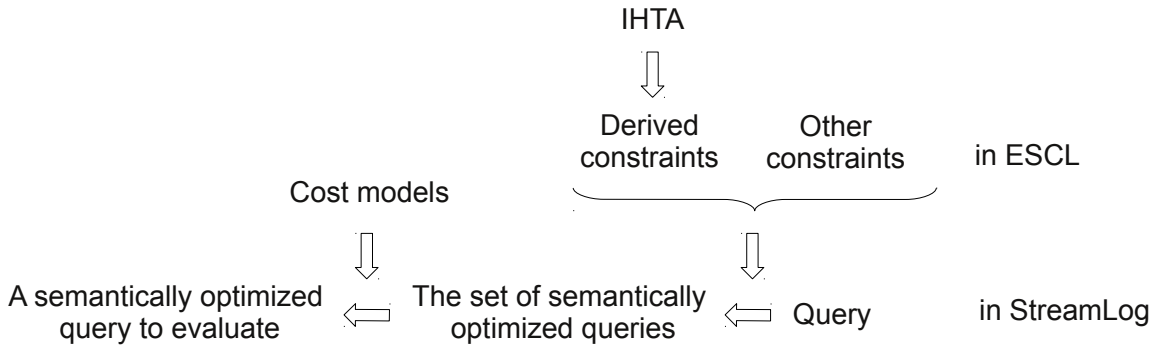

Figure 1.1: Overview of the Approach

data.

The second fundamental difference between database systems and CEP is that in CEP the validity of data and metadata, and the evaluation of queries are dependent from time.

There is no such dependency in databases. Indeed, data and constraints are valid until update, queries have usually no evaluation time. They are evaluated when they are put.

In CEP, in contrast, the occurrence time of an event is its inherit part, constraints are valid and queries are evaluated during particular periods of time which can also be application states, i.e. time intervals the beginning and the duration of which is unknown at compile time. Besides, many CEP applications have numerous involved workflow determined constraints.

Inspired by these observations, a semantic optimization method for CEP is proposed in this report. The approach consists of the following parts (compare Figure 1.1):

1. Event Stream Constraint Language (short ESCL)

   ESCL is a declarative first-order logic language tailored to the peculiarities of CEP (discussed in Section 2.2) in contrast to the other constraint languages used for semantic optimization of event queries. (The overview of the languages is given in Section 3.1.) ESCL constraints are readable, easy to use, and expressive. The language supports causal, cardinality, data, temporal, and spatial constraints on events, application states, and data saved in a conventional database. ESCL has strong formal foundation and is modularly defined.

2. Instantiating Hierarchical Timed Automata (short IHTA)

   IHTA is a model of CEP application semantics which will be used as metadata for semantic optimization of CEP. To the best of our knowledge, IHTA are the only kind of automata used in this way. But they are not restricted to this purpose.

Like all automata, IHTA capture the notion of state very naturally. IHTA are expressive. Indeed, they convey many temporal, causal, cardinality, and data relations between events and states in a readable way. IHTA are independent from the language specifying their transition labels. However, the expressiveness of IHTA and the kinds of constraints captured by IHTA depend on the language specifying their transition labels. IHTA are non-deterministic and able to work with events with the same occurrence time. IHTA are hierarchical and, therefore, modular which implies abstraction, readability, (ex-) changeability, and reuse of the modules. In contrast to other hierarchical models (considered in Section 3.2.1), IHTA can represent an arbitrary number of concurrent processes (as required in many CEP applications, see Section 2.1.1) by instantiation of module specifications.

3. Derivation of ESCL constraints from IHTA

   IHTA represent many CEP constraints in a readable way. In particular, the constraints determined by the application workflow are expressed by the automata in a concise and comprehensive for the user way. However, the algorithm semantically rewriting CEP queries with respect to the application semantics works on metadata expressed as a set of constraints, not automata. Therefore, IHTA will be automatically transformed into a set of ESCL constraints.

4. Propagation of ESCL constraints with respect to CEP queries

   Event-based systems work with large amount of event data continuously arriving on potentially infinite event streams. Under these circumstances, garbage collection is often indispensable. Hence, some base relations possibly became incomplete in the meantime. As a consequence, CEP views cannot be led back to their base events (states) without lost of results in some cases. Therefore, in order to semantically optimize CEP queries which are based on CEP views, constraints have to be defined for these views too and not only for their base events and states. To reduce the number of constraints which must be defined by the user, only the ECSL constraints for base events and states will be manually specified and automatically propagated to CEP views with respect to the queries deriving the views.

5. Derivation of ESCL constraints from CEP queries independently from constraints

   Some constraints will be derived from CEP queries without consideration of constraints. This also reduces the number of constraints specified by the user. Examples are given in Section 12.3.

6. Subsumption of CEP conjunctive queries

Subsumption of CEP queries is required for multi-query (semantic) optimization and for determination of the relevance of a constraint for a query (which is the base of the semantic optimization of CEP queries). The subsumption algorithm for Horn clauses proposed in [56] is adapted to CEP queries. This adaption is necessary because CEP queries are different from clauses, in particular temporal conditions must be taken into consideration by an algorithm deciding subsumption of CEP queries.

7. Semantic rewriting of CEP queries with respect to ESCL constraints

   The algorithm semantically rewriting CEP queries with respect to ESCL constraints is an adaption of the residue method semantically rewriting database queries with respect to database constraints which was proposed in [31]. The method is static and rather expensive (exponential in the worst case, see Section 10.2). That is why the residue method is not wide-spread in database systems where queries are usually ad hoc, data is completely available when queries are put, and hence, query answers are expected (almost) immediately. However, as motivated in Section 2.2.1, static semantic optimization of CEP queries may be inefficient if it happens before events are available and queries can be evaluated. Besides, repeated evaluation of CEP queries over a long period of time is worth their expensive static semantic optimization.

8. Choice of one of the alternatively semantically optimized queries to evaluate instead of each initial query using cost models

   The algorithm semantically rewriting CEP queries with respect to ESCL constraints returns a set of alternatively semantically optimized queries for each original query. Than, the evaluation costs of all queries of a set are determined using cost models, and one of them with the lowest (estimated) evaluation cost is processed instead of its respective initial query.

9. Implementation, experimental evaluation, visual editor

   Finally, the whole method will be implemented in Java and its profit will be experimentally demonstrated. A visual editor will be built to facilitate the use of the approach. The user will specify the application semantics by IHTA and ESCL constraints and put CEP queries to evaluate. IHTA will be automatically transformed into ESCL constraints. The entire set of constraints will be further used for the semantic rewriting of the CEP queries. The user will choose cost models out of the list of cost models supported by the system. According to these cost models, for each original query, a semantically optimized query with the lowest estimated cost

will be chosen to evaluate instead of the initial query. Finally, the answers for the optimized queries will be computed and printed out.

1.2 Contributions and Organization of this Report

At the moment, only some of the above points are completely elaborated and described in this report. They are:

1. Event Stream Constraint Language
2. Instantiating Hierarchical Timed Automata
3. Propagation of ESCL cardinality constraints with respect to CEP queries
4. Subsumption of CEP conjunctive queries
5. Semantic rewriting of CEP queries with respect to ESCL constraints

All the other parts of the method are still subjects for future research.

This report is organized as follows. Chapter 2 motivates the presented approach. Chapter 3 describes the related work and its differences to the approach. Chapter 4 defines the basic notions of the approach. The semantic optimization method for CEP introduced in this report is independent from an event query language. To illustrate it, queries are expressed in SreamLog. Chapter 5 is devoted to the language. Chapter 6 formally defines the Instantiating Hierarchical Timed Automata and Chapter 7 the Event Stream Constraint Language. In Chapter 8, the propagation of ESCL cardinality constraints with respect to CEP queries is formally defined. Chapter 9 adapts of the subsumption algorithm for Horn clauses [56] to CEP conjunctive queries. Chapter 10 adapts the residue method for semantic optimization of database queries [31] to CEP queries. Chapter 11 concludes this report. Chapter 12 describes the future research directions.

Section 2.3, Chapter 6, and Section 3.2 are parts of the project thesis of Sandro Giessl [61]. Chapter 8 is a part of the bachelor thesis of Son Thanh Dang. (The bachelor thesis is currently being elaborated.) These results were worked out and presented in a comprehensible way together with the students.

Chapter 2

Motivation

This chapter motivates the presented approach. Section 2.1 describes of the CEP use cases. Section 2.2 identifies the differences of the semantic optimization of CEP queries compared to the classical semantic optimization of database queries due to the peculiarities of CEP. Section 2.3 analyzes the requirements to metadata for semantic optimization of CEP.

2.1 Use Cases

Events usually arrive not accidentally on event streams but follow the predefined application workflow and/or obey to the physical laws.

As an example for an event-based application with a predefined workflow, consider the online auction use case in Section 2.1.1. This workflow consists of multiple relatively independent processes (or workflows) running either sequentially or concurrently. This workflow reflects the application logic determined by the rules according to which an auction takes place.

As an example of a CEP application determined by physical laws, the emergency detection in a metro station is presented in Section 2.1.2. It has rich application semantics but without complex workflow. In contrast to the online auction use case, only few application states can be identified in the emergency detection use case, they are for example *normal*, *critical*, and *emergency*.

The analysis of CEP applications has shown that each of them falls into one of these groups. For the sake of brevity in this report, only one use case per group is considered.

2.1.1 Online Auction

An arbitrary number of auctions may take place concurrently. Each of them runs according to the rules described in the following. In an auction, only enrolled users are allowed to place bids for an item. Bidders are enrolled during the first 20 minutes of an auction. During the bidder enrollment, users may optionally undergo a registration step and then authenticate in order to enroll themselves for the auction. Afterward, if less than two bidders are enrolled, the auction closes. Otherwise, the auction continues and at least one item is offered to the bidders, one item at a time. For each item, bidders may place their bids with a price which must be higher than the price of the previous bids. After 30 seconds without bids and hammer beats, there is a new hammer beat. When three hammer beats took place subsequently, i.e. when there was no bid between these hammer beats, further bids are forbidden. If there was at least one bid, the item is sold to the bidder who placed the bid with the highest price. When an item has been presented, further items may be offered until the auction terminates.

This use case inspired from [116] is interesting because it has a rather involved workflow with multiple states (bidder enrollments, item offers, etc.) during which different events arrive and numerous diverse and complex dependencies between events hold (like between bids and hammer beats). The use case demonstrates that events must be treated differently depending on states in which they occur. For example, for each hammer beat event for an item it is essential to know how many sequent hammer beat events for the same item precede it. Only after the third sequent hammer beat event for an item no further bid events for the same item may follow. All queries asking for bid events can compute their answers and irrelevant bid events can be deleted. One way to express this knowledge is to count the number of sequent hammer beat events for an item every 90 seconds. Such constraints are rather unreadable and in many cases even not expressible since the time window is often unknown. The way that we propose is to formalize the application workflow as Instantiating Hierarchical Timed Automata (Chapter 6) and to put queries and constraints in the context of states in which they are satisfiable or valid (Chapter 5, Chapter 7). Besides, this use case shows the need for modeling an unbounded number of parallel processes like bidder enrollments or auctions.

2.1.2 Emergency Detection in a Metro Station

Identification of different kinds of emergency situations in a metro station, description of real-life examples, and analysis of the requirements to the supervision and management systems can be found in [17]. In this section, we concentrate ourselves on the requirements

analysis to the metadata capturing application knowledge specific for this use case.

This use case is quite different form the online auction described in Section 2.1.1 because its semantics is not determined by a complex workflow but rather by physical laws. However, similarly to the online auction use case, emergency detection in metro station requires consideration of the application states. Consider the following example from [50].

Example 1.
Assume a metro station is on fire. In order to avoid alarm fatigue, two rules can be used: One detects candidates for fire alarms. The other generates a fire alarm from the alarm candidates if there is no fire alarm during the recent hour in this area.

However, compared to observing and querying the state of the station, the solution proposed in Example 1 is suboptimal for two reasons. First, the station can be on fire longer than an hour such that multiple fire alarms are generated for the same emergency. Second, the state of the station is also relevant for other rules, like the one in Example 2.

Example 2.
The event reporting that a lamp is broken in an area is ignored if this area is on fire. But under normal circumstances, the reaction to the event is the exchange of the lamp.

In contrast to the online auction, spatial constraints are indispensable for the emergency detection in a metro station. They must be often combined with temporal, data, and cardinality constraints on events, states, and data saved in a conventional database. Consider the following example.

Example 3.
In order to evacuate people from a burning station, it is essential to know how fast and in which direction smoke spreads through the station and the neighbor areas. This information can be saved in the form of the following constraints. One constraint can limit the distance between the locations two smoke events where sent from within a short period of time. Other constraint can access a database for the ventilation direction in an area and forecast the direction in which smoke spreads in this area. Both constraints must take the state of the neighbor areas into consideration.

In Chapter 7, we propose the Event Stream Constraint Language formulating such expressive constraints in a readable way.

2.2 Semantic Optimization of CEP vs. Semantic Optimization of Database Systems

Semantic optimization of CEP queries differs from classical semantic optimization of database queries due to the peculiarities of CEP described in this section.

2.2.1 Standing Queries and Moving Data

In database systems, evaluation is *query-driven*. This involves two points:

1. Data
   Large (but finite) data is permanently saved, available at once, and known.

2. Queries
   Ad hoc queries are evaluated once against the whole data. Query answers are expected almost immediately. Standing queries are rather typical for a data warehouse than for a database.

Since data is always available, efficient data storage, management, and access are important and well-investigated issues in database systems. Since queries are usually not available before their evaluation must be started, static semantic optimization of queries must be efficient. It is limited because of this reason.

In CEP, evaluation is *data-driven*. Data and queries can be characterized as follows:

1. Data
   Streams are unbounded, potentially infinite. Under these circumstances garbage collection is often indispensable since naive stream processing saving all events forever runs out of memory sooner or later. Garbage collection means the possibility to delete events if they cannot contribute to new answers of a query (any more) [27]. Event streams are never available at once. Their contents is often unknown.

2. Queries
   Standing, known, and available queries are continuously evaluated against events arriving on streams. Query answers are computed in multiple (possibly arbitrarily many) evaluation steps. Each step takes place as soon as events become available. Ad hoc queries are not typical for CEP. They usually have to wait until relevant events arrive. They cannot be evaluated immediately since (some) past relevant events are

probably already garbage collected. This happens in particular if automatic garbage collection is based on standing queries as defined in [27].

Since data is often not available before query evaluation starts and since events are often evaluated on-the-fly (without storage), only few approaches are devoted to the efficient management of event data, e.g. [23], [54]. Static semantic optimization of event queries may rely on more complex and expensive algorithms than static semantic optimization of database queries in particular if the event data is not available yet when queries are put. Besides, the high costs for query optimization are often negligible compared to the gain due to the repeatable query evaluation over a long period of time. Chapter 10 presents the algorithm for static semantic rewriting of CEP queries in pseudo code and its properties with respect to correctness, termination, and complexity.

2.2.2 Time

In database systems, tuples and integrity constraints are valid until update. The time at which a database query is evaluated is usually not an inherent part of the query.

In CEP, events occur, constraints are valid, and queries are evaluated during particular periods of time. Occurrence time of events, validity time of constraints, and evaluation time of queries are their inherent parts. Constraint validity time and query evaluation time can be determined by application states, i.e. their exact duration is usually known at runtime. Therefore, the metadata used for semantic optimization of CEP must (1) allow for formulating application semantics in the context of states and (2) capture complex temporal and causal relations between events and states as needed in many various CEP applications (consider examples in Section 2.1). To cope with these two requirements, Instantiating Hierarchical Timed Automata (IHTA) and the Event Stream Constraint Language ESCL are introduced in Chapter 6 and Chapter 7 respectively.

2.2.3 Multi-Query Optimization

An essential difference between semantic optimization of database queries and semantic optimization of CEP queries is the treatment of derived relations, i.e. views. View materialization in CEP is more profitable than view materialization in database systems for the following reasons:

1. Amount of data

   A database is usually finite. There is no need for garbage collection. All relations

are always complete. Therefore views can always be led back to their base relations without lost of tuples.

Event-based systems work with large amount of event data continuously arriving on potentially infinite event streams. Garbage collection is often indispensable. The relatively few results of queries are usually saved in order to delete relatively many base events and states. Because of garbage collection, some base relations possibly became incomplete in the meantime. As a consequence, views cannot be led back to their base relations without lost of tuples in some cases.

2. Potential of multi-query optimization

   In database systems, multi-query optimization is limited for the following reasons:

   (a) Multiple database queries are seldom available at the same time. Queries are usually ad hoc and the time at which a query is asked and processed does not coincide with that of other queries.

   (b) Each query is usually evaluated only once against the whole data such that expensive multi-query optimization is not profitable.

   Prepared queries are rather typical for a warehouse than for a database. Prepared queries are available at the same time and can be processed repeatedly but the time at which they are evaluated is not always known beforehand.

   Multi-query optimization in CEP is more important than in database systems for the following reasons:

   (a) A set of queries is available at the same time.

   (b) The set of all queries is evaluated repeatedly as soon as events arrive. The number of the evaluation steps of a query may be unbounded such that even complex and expensive static semantic query optimization becomes profitable. The time at which a query makes an evaluation step is usually known.

   Ad hoc queries are not typical for CEP.

3. Costs for view maintenance

   Maintenance of a database view is expensive since it involves manual insert, deletion, and update of tuples.

   View maintenance in CEP is cheaper than in database systems because it does not involve manual deletion of events and states (they are garbage collected). However, like derived tuples of a database, events and states can be inserted and states can be updated in a view. (Events are not updated.)

As a consequence, database views are not materialized and the queries are rewritten such that views are reduced to their base relations. Afterwards, these rewritten queries are semantically optimized with respect to constraints which are defined for base relations only [31].

In contrast, CEP views are materialized because they will not (and often cannot) be led back to the events (states) they were derived from. In order to semantically optimize queries which are based on CEP views, constraints have to be defined for these views and not only for their base events and states. In order to reduce the number of manually specified constraints some of them can be automatically propagated from base events and states to CEP views with respect to queries deriving them. Chapter 8 is devoted to the propagation of CEP cardinality constraints, i.e. constraints restricting the number of events (states) during particular time intervals.

Summarizing, semantic optimization of event queries differs from semantic optimization of database queries as follows: (1) Static semantic optimization of event queries may rely on more complex and expensive algorithms than static semantic optimization of database queries. (2) Metadata used for semantic optimization of CEP must cope with application states and complex temporal and causal relations between events and states. (3) Multi-query semantic optimization plays a more important role in CEP than in database systems.

2.3 Requirements to Metadata for Semantic Optimization of CEP

A formalism for application semantics of event-based systems should:

1. be stateful,
2. express involved temporal, causal, spatial, and other relations between events and states,
3. be modular,
4. represent an arbitrary number of concurrent processes,
5. be non-deterministic,
6. work with events which have the same occurrence time,

7. be approximate, and finally,

8. be visual and well readable.

These requirements are motivated in this section. The actual challenge of this work is to combine the above features in one model.

2.3.1 States

Application semantics is not only dependent on the recent event but rather on the history of many previous events: For example, in an auction it is not sufficient to consider only one hammer beat to decide if the item is sold or if further bids are allowed. Instead, for an accurate decision at least the last three events of an auction have to be considered, "counting" subsequent hammer beats. However, hammer beats only serve for a simple example here. Often, there are *multiple* sequences of events which lead to a certain condition and these sequences are not only *complex* but their *duration* or *length* might be *unbounded*. Therefore, specifying these conditions by manually describing sequences of events is tedious or even impossible. To overcome this, states have to be supported by a formalism for application-specific knowledge. A state can be understood as an abstraction of events received on an event stream so far.[1] There might be various application states in the online auction, e.g. "an item is currently offered" or "no further bids for an item are possible".

When events arrive, their treatment depends on the application states during which they occur. Besides, states allow for automatic termination of queries and garbage collection of irrelevant events, i.e. events which cannot contribute to an answer for a query (any more). Constraints on event streams are usually valid during certain states and not always. Hence, event queries and constraints should be formulated in the context of application states. This is more concise and less repetitive than expressing time windows or other temporal conditions for the evaluation time of queries and the validity time of constraints. Note that the states during which events queried in query or constraint bodies arrive are known and can be determined automatically. In such cases, event queries and constraints do not even have to be expressed in the context of states explicitly.

Since an arbitrary number of processes (e.g. auctions) can take place at the same time and each of the processes has its own current state, there is a need to relate each running process to its current state. To this end, states could carry data (e.g. an auction

[1] However in this work, a state is not considered as a history of all past events like for example in Transaction Logic [20].

identifier). Such a formalism would be more expressive than classical automata such as nondeterministic finite automata [108].

2.3.2 Involved Relations between Events and States

As motivated in Section 2.1.1, in many CEP applications, events follow specific workflows which can be rather complex. This in particular implies involved causal and temporal relationships. The following examples illustrate the relations between events but similar relations exist between states as well as between events and states.
Examples for causal relations: If at least two bidders are enrolled for an auction at least one item is presented in the auction. If there is at least one bid for an item, the item will be sold, i.e. there are at least three hammer heats and exactly one sell of the item.
Examples for temporal relations: In an auction, bids for an item happen only after the item has been presented. After a bid, there must be a subsequent bid within less than 30 seconds or a hammer beat after exactly 30 seconds.

Remember that an arbitrary number of auctions may take place at the same time and an arbitrary number of items is presented during an auction. In order to relate each event to a particular auction and a particular item, events carry data such as an auction identifier or an item identifier. This implies numerous functional dependencies between the data of different events.
Examples for functional dependencies between event data: If an item with a particular item identifier is presented only once in all auctions, an auction identifier is functionally dependent from an item identifier in all events and states.

Besides functional dependencies, cardinality of events (i.e. the number of events during time intervals) are implied by a complex workflow.
Examples for cardinality relations: There is exactly one item description for each item. There are no or at least three hammer beats for each item.

Note that a formalism for application semantics must be able to express an arbitrary combination of the above relations between events and states because they usually hold concertedly as the above examples show.

2.3.3 Modularization

Many workflows can be divided into independent sequential or concurrent processes. For example, in an auction, there is an arbitrary number of concurrent bidder enrollment processes followed by an arbitrary number of sequential item offer processes. This mo-

tivates the idea of splitting models into readable and manageable modules representing subprocesses.

Indeed, application processes can often be described from different perspectives. Describing the workflow of an auction in terms of bidder enrollments and item offers is a high-level description compared to the specific details of bidder enrollments and item offers which are low-level descriptions. A specific process with a low-level description (e.g. a bidder enrollment) can be regarded as subprocess of another high-level process (e.g. an auction).

A process and an arbitrary number of its subprocesses can (but do not have to) run simultaneously. If a process runs it is possible that none of its subprocesses are running. But if at least one subprocess is active its respective (super) process must also be running, e.g. an item can only be presented during an auction.

A high-level process can be understood as an abstraction of its subprocesses. Supporting this abstraction in the formalism is important to reduce complexity, to increase readability, and to support stepwise refinement (i.e. beginning with a high-level specification and iteratively extending it with low-level details). Other benefits of modularization are reusability and (ex-)changeability of modules without affecting others.

2.3.4 Arbitrary Number of Concurrent Processes

As mentioned above an unbounded number of concurrent processes (e.g. auctions or bidder enrollments within an auction) must be expressed by the formalism. Each concurrent process has its own current state and runs relatively independently from other concurrent processes on the same description level (aside from certain synchronization points such as the end of all bidder enrollment processes after 20 minutes since the auction beginning have passed).

Each concurrent process can arbitrarily change its state within its state space. Without support for concurrency, this behavior could be simulated by a higher-level process description to a limited extent as illustrated by the following example. Consider two concurrent processes with the state spaces $\{a, b, c\}$ and $\{d, e, f\}$ respectively. Their current states can be simulated by nine states in the description of a higher-level process: (a, d), (a, e), (a, f), (b, d), (b, e), (b, f), (c, b), (c, e), and (c, f). This would quickly result in an unreadable model of a *fixed* number of concurrent processes. Modeling an arbitrary number of concurrent processes is not possible with such an approach. Approaches like Hierarchical Timed Automata [41] and Statecharts [71] which are similar to the approach discussed in this paragraph will be discussed in Section 3.2.

One possible solution of this problem is a formalism in which the specifications of processes are modeled by low-level descriptions which, in higher-level descriptions, are marked to be concurrent. Concurrent processes must start, change their states, and terminate dynamically during the run of the automaton.

2.3.5 Non-determinism

In contrast to the auction use case, many event-driven applications are non-deterministic, i.e. more than one state is reachable from a state. This feature must be supported by the model of application semantics. There are two kinds of non-determinism [72]:

1. Don't care non-determinism (also called conjunctive non-determinism) means that all choices will lead to a successful search, so we "don't care" which one we take.
2. Don't know non-determinism (also called disjunctive non-determinism) means that some of the choices will lead to a successful search, but we "don't know" which one a priori. There are two possibilities to treat the problem:
   (a) Either we guess one of the choices, check whether it is right, and stop guessing as soon as we have found a successful choice (like Non-deterministic Finite State Automata, see Section 3.2.1). Otherwise continue checking the remaining choices.
   (b) or we follow all possible choices and drop the choices which turn out to be wrong (like Powerset construction method translating Non-deterministic Finite State Automata into Deterministic Finite State Automata, see Section 6.3.2).

Conjunctive non-determinism is not problematic because it is known beforehand that all choices are right. This is not the case by disjunctive non-determinism which is therefore more complicated. The second solution dealing with disjunctive non-determinism allows event stream verification on-the-fly, i.e. there is no need to save past events in order to test other choices. Therefore the second way of treating disjunctive nondeterminism is more preferable than the first one.

2.3.6 Other Requirements

Other requirements to a formalism for application semantics in CEP systems are the following:

- **Treatment of events with the same occurrence time**
  If the event rate is high or the granularity of the discrete time is coarse[2] it is possible that same events have exactly the same occurrence time. The formalism of application-specific knowledge must be able to treat such events.

- **Approximate specification**
  It is quite seldom that the application semantics is fully defined and completely known beforehand. Therefore a formalism capturing it should be approximate. It should be possible to describe different parts of the model at an arbitrary level of abstraction.

- **Readable visual representation**
  Readable visual representation of a complex formalism facilitates the understandability of the model, in particular by non-technical persons. There is a trade-off between readability and expressiveness of a model which is one of the main challenges.

[2]The problems concerning different time models are discussed in Section 4.1.

Chapter 3

Related Work

This chapter is devoted to the related work. Section 3.1 compares the approaches on semantic optimization of CEP with respect to the following criteria: (1) Kind of used metadata (2) Time at which semantic optimization happens (3) Language dependency and (4) Implemented semantic optimization technique. Section 3.2 analyses the visual formalisms modeling complex dynamic workflows with respect to the requirements described in Section 2.3. And finally, Section 3.3 gives an overview of the algorithms deciding subsumption of clauses.

3.1 Semantic Optimization of CEP

3.1.1 Kinds of Metadata, Constraint Languages

Regarding the way of expressing application semantics, the approaches are divided into two groups, namely semantic optimization by means of either *constraints* or *queries* themselves.

Constraints are formulated in DDL [62], DTD [113, 112], [82, 83], [44], [70], [65], [91], as regular expressions [49], [48], [58], [116], denial rules [47], or language independent formulas [113, 112], [44], [58], [27], [13], [121], [12, 92]. No approach uses a constraint language which is tailored to the peculiarities of CEP as discussed in Section 2.3. Most approaches consider particular kind(s) of constraints to a limited extent. These kinds of constraints are conditions on event data [49], [48], [13], [116], cardinality constraints [113, 112], [44], [62], [13], [121], [12, 92], temporal [113, 112], [44], [49], [48], [58], [62], [27], [13], [116], [47] and causal dependencies [113, 112], [47]. Constraints are expressed only on events. They do not involve application states.

Not only constraints but also queries capture application semantics. Hence, they can be used for the semantic optimization of CEP as done in [23], [54], [75], [36], [35], [88], [11], [28, 89].

No approach describes the application workflow by automata and uses the semantics captured by them (e.g., temporal and causal dependencies, states) for semantic optimization of CEP. However, [47], [91], [112], [65] transfer queries into finite automata and evaluate the automata while events arrive. This approaches do not use automata as metadata for semantic query optimization in CEP.

3.1.2 Static and Dynamic Approaches

According to the time at which semantic optimization happens, the approaches are classified into *static*, *dynamic*, or *both*.

Most approaches are static [82, 83], [44], [91], [58], [75], [62], [88], [28, 89], [23], [54], [36]. They rely upon application semantics which is known at compile time and does not change at runtime.

There are some dynamic approaches. In [13], constraints are derived from the event stream. [116], [48] use constraints arriving on streams. [121], [35], [12, 92] rely on stream rate. In [47], [27], [70], constraints are known at compile time. They do not change at runtime but they are applied at runtime. [11] dynamically selects time intervals upon which query results are shared.

[113, 112], [65], [49] allow both static and dynamic semantic optimization of event queries.

3.1.3 Language-specific and General Approaches

Concerning language dependency, there are numerous *specific* and few *general* approaches.

Some approaches are specific for XML data streams and XPath [70], [65] or XQuery [113, 112], [82, 83], [44], [91]. However the ideas of these approaches are independent from an event format or a query language. These approaches have poor event concept. An event is an XML document (or even a single tag) without occurrence time.

Other approaches are tailored to particular relational algebra operator(s) and therefore to particular kind(s) of queries. Much attention has been paid to join [49], [48], [27], [13], join combined with *count* [62], join combined with selection [36], [35], join or grouping [116], join, selection or projection [121], equi-join or aggregation [58], sliding-window

aggregates [88], [11], event sequence or conjunction [47]. Join and aggregation are expensive operations requiring storage of relevant events and blocking of queries which cannot be evaluated as long as not all events are available.

Few approaches are general [23], [54], [75], [28, 89], [12, 92].

3.1.4 Semantic Optimization Techniques

With respect to the implemented semantic optimization technique, there are approaches aiming at:
– *Load distribution* [58], [75],
– *Storage minimization* [82, 83], [13], [116], [35], [88], [28, 89],
– *Efficient data structures* [23], [54],
– *Computation sharing* [44], [36], [35], [88], [11],
– *Efficient join algorithms* [49], [48],
– *Semantic optimization heuristics* such as join elimination [62] and detection of (temporary) unsatisfiable (sub) queries [113, 112], [70], [27], [116], [47],
– *Query plan rewriting* [121], [12, 92], and finally
– *Query compilation* [113, 112], [82, 83], [44], [70], [65], [91].

The semantic optimization approach presented in this report is tailored to data-driven, time-aware CEP relying on unbounded streams. It highlights the necessity of application states for the semantic optimization of CEP. States make queries and constraints more expressive, flexible, readable, and their evaluation more efficient. According to the above classifications this approach is characterized as follows. It relies on Instantiating Hierarchical Timed Automata (sort IHTA, Chapter 6) and constraints expressed in the Event Stream Constraint Language ESCL (Chapter 7). IHTA express states very naturally and capture involved temporal and causal constraints for an arbitrary number of concurrent processes. ESCL constraints are short but expressive logic formulas completing IHTA with additional knowledge about events and states. The approach is static and general. It aims at query plan rewriting involving the implementation of semantic optimization heuristics, computation sharing, and storage minimization.

3.2 Models of Application Semantics

Automata, flowcharts, Petri nets, and UML behavior diagrams are visual formalisms modeling complex dynamic workflows. Even though these approaches share similar goals, they concentrate on different aspects. This chapter describes the formalisms briefly, analyses them with respect to the requirements described in Section 2.3, and compares them with the Instantiating Hierarchical Timed Automata (IHTA) proposed in Chapter 6.

3.2.1 Automata

Automata (also called (Finite) State Machines) are one of the most fundamental, widely used and well-studied modeling mechanisms in computer science since the 50's. There are different kinds of automata. We start with classical Finite State Automata in Section 3.2.1 and continue with their extensions which are relevant for IHTA, in particular the addition of variables, to form Extended Finite State Machines (Section 3.2.1), the extension to work on infinite input, to form ω-Automata (Section 3.2.1), the addition of clocks, to form Timed Automata (Section 3.2.1), the addition of hierarchical (nesting) capability with or without concurrency (communication), to form (Communicating) Hierarchical Finite State Machines (Section 3.2.1), and, finally, the differentiation between universal and existentional paths, to form Alternating Machines (Section 3.2.1).

Finite State Automata

Finite state automata [108] consist of (a finite number of) states and transitions between them. Transitions are labeled by symbols of an alphabet. If an automaton is in a state s and the label of an outgoing transition t of s matches the next symbol of the input word, t fires and the automaton is in the state t leads into. If each state of an automaton has at most one outgoing transition labeled with a symbol, the automaton is called *deterministic*. If there is a state with at least two outgoing transitions with the same label, the automaton is *non-deterministic*. There are some other features distinguishing these two kinds of finite state automata which are omitted hier for the sake of brevity. We refer the reader to [108]. An automaton describes the language consisting of words which are accepted by the automaton, i.e. which lead the automaton into one of its end states.

If we consider the set of all events as an alphabet and event streams as words, automata can be seen as a description (specification) of these event streams. This issue will be discussed in Chapter 4 in more detail.

There are different kinds of automata working on event streams and extending classical

Finite State Automata with additional features such as temporal constraints, hierarchy of automata, concurrency, and variables. The following sections are devoted to them. Instantiating Hierarchical Timed Automata (IHTA) we propose are also an extension of classical automata in these respects. IHTA are seen as a metadata for semantic optimization of event queries. Finite state automata were used to a limited extent for this purpose in [47], [91], [113], [65]. In these approaches, however, automata represent event queries rather than the application workflow as in our approach.

Extended Finite State Machines

Extended Finite State Machines [37] are Finite State Machines equipped with variables which are either input, output or local. The values of variables can be updated while a transition fires. Constraints on the values of these variables can be added to transitions. Since the domain of each variable is not restricted, Extended Finite State Machines cannot be translated into ordinary Finite State Machines.

In contrast to classical Finite State Machines, Extended Finite State Machines do not represent the entire state space explicitly and are, therefore, more compact, readable, and cheaper to build. The main idea introduced in [39] is to manipulate sets of states and transitions simultaneously and represent these sets by Boolean functions, i.e. the state space is represented symbolically rather than explicitly.

IHTA use two kinds of variables, namely data variables and identifiers. Both are local input variables which are bound while matching events or states, and can be rebound (overshadowed) later. No other variables are supported. In contrast to IHTA, Extended Finite State Machines are neither timed, nor hierarchical. They do not support concurrency and were not used as metadata for semantic query optimization.

ω-Automata

ω-Automata [114], [99] (also called stream automata) extend Finite State Automata to work with infinite input. There are different kinds of ω-Automata: Büchi automata [30], [102], Muller automata [15], [19], Rabin automata [74], Streett automata [74], [102], and parity automata [81], [102]. There are deterministic and non-deterministic variants of each of them. These automata accept regular ω-languages (generalizing regular languages to infinite words) but differ in acceptance criteria and succinctness of representation of a regular ω-language.

Like ω-Automata, IHTA work on infinite input. However, in contrast to IHTA, classical stream automata neglect temporal aspects of event processing. Therefore, no acceptance

criterion of ω-Automata agrees with the semantics of IHTA. Whether IHTA accepts input stream depends on the current time and events arrived so far (see Definition 23). ω-Automata also neglect event data. They are not hierarchical and do not support concurrency.

Timed Automata

Timed Automata originally introduced by Rajeev Alur and David L. Dill in [5] in 1990, are ω-Automata equipped with a finite set of clocks. A clock is a piece-wise continuous real-valued function of time that records the time elapsed since the recent reset of the clock.[1] A clock can be reset to a new value if a transition fires. Respective command is a part of the transition label. All clocks are synchronized.[2]

The edges of Timed Automata are labeled by events, temporal constraints on clocks, and reset commands on clocks. An event is an atomic symbol (i.e. events do not carry data) and a real-valued time point of occurrence associated with it. Many events may have the same time point of occurrence. A temporal constraint on a clock is a comparison of the clock value with a number. A reset command on a clock is an assignment of a new value to the clock. Timed Automata describe a timed language consisting of timed words (i.e. event streams) which are accepted by the automaton, i.e. which lead the automaton into one of its end states.

Timed Automata are used to model and reason about real-time systems such as network protocols, business processes, reactive systems, etc. They are well-studied from the perspective of formal language theory [5], [6], [4], [8], [16], [101], [57], and model checking [125], [24], [110], [73], [85], [25], [86]. Considerable amount of work has been done on the automatic inference of Timed Automata from data [118], [119], [120], [67], [66].

Timed Automata were initially developed for event stream verification, in particular, for the expression of constant bounds on the delays between events [5]. Later, Timed Automata were also used for solving scheduling problems, consider [1]. To the best of our knowledge, they were not considered as metadata for semantic optimization of event queries.

Like Timed Automata, IHTA are able to process events with the same occurrence time. In contrast to Timed Automata, IHTA work with events which carry data and have time intervals as their occurrence time. Transitions of IHTA are labeled with event queries which provide access to event data and/or temporal constraints which are defined on

[1] There are however Timed Automata which are based on discrete time model, e.g. [69].
[2] There are distributed Timed Automata working with asynchronous clocks, e.g. [46].

the beginning and the end of the occurrence time of matched events or on current time. Temporal conditions of Timed Automata which are expressed on clocks can simulate the temporal constraints of IHTA which involve the end but not the beginning (!) of the event occurrence time. Hence, transition labels of IHTA are more expressive than that of Timed Automata. Timed Automata do not support hierarchy of automata. Hierarchical Timed Automata extend Timed Automata by this feature.

Hierarchical Finite State Machines

It is difficult, if not impossible, to model systems of a certain size and complexity using flat automata. Most systems can be divided into relatively independent manageable and comprehensible processes. To this end, state-diagrams were extended by the notions of hierarchy, concurrency, and communication, to form the so-called Statecharts in [71] in 1987. Hierarchy or nesting means that each state of a model may be refined by another (possibly hierarchical) model. This enables viewing the description at different levels of detail, i.e. abstraction. Concurrency and communication mean that there can be multiple models running in parallel within the same state and communicating with each other by messages (events) to synchronize their behavior.

Since 1987 Hierarchical Finite State Machines have been further developed. [124] summarizes the work on such machines with and without concurrency done until 2000. Most of the results on expressiveness, complexity, and model checking come from [7], [9], [10]. Like ordinary Finite State Machines, Hierarchical Finite State Machines capture regular languages but they gain in exponential succinctness as compared to their respective flattened machines. In other words, hierarchical machines can be translated to classical Finite State Machines at an exponential cost. A concurrent (or communicating) Hierarchical State Machine combines concurrency and hierarchy in an arbitrary manner. It still defines only regular languages. Its flattening causes in general a double exponential blow up.

A hierarchical machine satisfies a property if its respective flattened machine does. This is the usual approach of the investigation of the properties of (communicating) hierarchical machines. But of course this flattening can be avoided as done in [9].

Among hierarchical machines, Hierarchical Timed Automata (HTA) [42], [41], [40] are the approach closest to IHTA we propose. The similarities between them are the following. First, both models support hierarchy of automata and are therefore modular (which implies readability, extensibility, (ex-)change, and reuse of modules as well as means for abstraction). States of (I)HTA which are (possibly hierarchical) automata themselves are called non-atomic. For two arbitrary states s_1 and s_2 of (I)HTA one of the following holds:

s_1 and s_2 are disjoined, s_1 is completely within s_2 or s_2 is completely within s_1.[3] Second, both models capture numerous complex temporal and causal relations between events and states.

However, IHTA can be seen as both an extension and a restriction of HTA in the following respects. Like all hierarchical machines we are aware of, HTA allow for modeling a fixed number of processes running concurrently. However in real life applications their number is often unbounded. Therefore, IHTA extend HTA by the possibility of modeling an arbitrary number of concurrent processes: On the beginning of a new process, the workflow of which is represented by a non-atomic state, the state is instantiated. An unbounded number of processes which run concurrently is expressed by multiple instances of the same non-atomic state existing at the same time. This feature of IHTA is, to our knowledge, new and not present in other formalisms. IHTA are therefore more expressive than hierarchical machines. The price we pay for the expressiveness is that IHTA cannot be flattened. In other words, the results of the investigation of the properties of hierarchical machines are not applicable to IHTA in general.

Further essential additional features of IHTA compared to HTA are, first, the event queries of IHTA allow access to event data in contrast to that of HTA and, second, temporal constraints of IHTA are defined on the beginning and the end of the occurrence time of matched events or on the current time and not on local clocks like the temporal constraints of HTA which is a less expressive approach as explained in Section 3.2.1.

Other features of HTA are not adopted by IHTA because they are not needed or not applicable to achieve our goals. They are:

1. *pseudo* states and *pseudo* transitions facilitating the hierarchy management but making the automaton bigger,

2. differentiation between *xor* and *and* superstates: If a *xor* superstate s is active then exactly one substate of s is active, if an *and* superstate s is active then all substates of s are active (IHTA support only the former kind of superstates),

3. *parallel* states expressing concurrent processes (concurrent processes are expressed by the instantiation of a non-atomic state of IHTA, whether it is necessary or preferable to instantiate different non-atomic states at the same time is an open issue),

4. *history* states permitting to resume the run of non-atomic states with the state they had before their suspension (the states of IHTA cannot be suspended),

[3] Statecharts [71] support a different kind of state hierarchy: Two states s_1 and s_2 may overlap so that neither s_1 is completely within s_2 no vice versa. Whether such a notion of hierarchy would by a preferable extension of IHTA remains to be investigated.

5. invariants associated with states (this could be an extension of IHTA which is thinkable and preferable for many applications),

6. local integer variables which can be assigned during transitions and transitions can be labeled with conditions on the values of these variables (variables of IHTA are also local, they can be of an arbitrary type, and allow access to the event data; no other variables are used in IHTA),

7. prioritization of event transitions over delay transitions (this feature is simulated by negation of an event query of IHTA, consider Chapter 5 for the details),

8. channel synchronization forcing two transitions associated with a channel to be performed in an atomic step (other kind of synchronization is supported by IHTA, namely atomic termination of all instances of a non-atomic state).

Alternating Machines

Alternating Turing Machines [33], [32], [109], [97] are Non-deterministic Turing Machines supporting two computation modes: Existential and universal. A computation in existential mode succeeds if any choice leads to an accepting state. A computation in universal mode succeeds only if all choices lead to an accepting state.[4] More exactly: Let several configurations $\beta_1, ..., \beta_k$ be reachable from a configuration α. If the branching is existential, α leads to acceptance if at least one configuration β_i leads to acceptance. If the branching is universal α leads to acceptance if *all* configurations $\beta_1, ..., \beta_k$ lead to acceptance.

Alternating Turing Machines can be understood as parallel machines where the successor configurations $\beta_1, ..., \beta_k$ run independently till completion, and their result (acceptance or rejection) is combined by the configuration α. In this point Alternating Turing Machines are similar to IHTA allowing multiple instances of the same non-atomic state s to run independently from each other till completion and an instance of the superstate of s to terminate these instances of s. Termination constraints restrict the number of successful processes represented by the instances of s. If these termination constraints are absent all processes must be successful which is comparable with (but not the same as) universal computation mode. The difference between concurrent processes of IHTA and universal branching of Alternating automata is that multiple concurrent instances are joined into one branch, while join of universal branches is not defined. If there is a non-deterministic choice in a (concurrent) process, at least one of the possibilities must be accepted for the process to be successful which corresponds to the existentional computation mode.

[4] Remember that classical Non-deterministic Finite State Automata support only the existential computation mode.

The theoretical findings of Alternating Turing Machines are used for time- and space-classification of problems. Problems of interest are, e.g. decision problems in logic and problems concerning the existence of winning strategies in combinatorial games, involving alternating quantifiers. To our knowledge, neither classical Alternating Turing Machines nor their extension by temporal aspects (i.e. clocks) called Alternating Timed Automata [87] were used as metadata for semantic query optimization. Alternating automata neglect event data and are not hierarchical.

Summary

Summarizing the overview of different kinds of state machines and their comparison with IHTA, we would like to emphasize that automata support the notion of application state very naturally since states are their explicit components. Considerable amount of work has been done to adjust automata to the peculiarities of event processing in general and the requirements of many event-based applications in particular such as ability to work on infinite input (ω-Automata, Section 3.2.1), temporal aspects (Timed Automata, Section 3.2.1), hierarchy of models (Hierarchical Automata, Section 3.2.1), and concurrency (Hierarchical Automata, Section 3.2.1, and Alternating Machines, Section 3.2.1). To the best of our knowledge, no existing automaton model supports access to event data, i.e. non-ground terms were not used as part of transition labels.[5] Most automata have weak time aspect since event occurrence time is a time point not a time interval. Therefore, only limited temporal constraints have been considered until now. Some automata allow multiple events with the same occurrence time (e.g., Timed Automata, Section 3.2.1). No existing automaton model is able to represent an arbitrary number of concurrent processes. Communication and synchronization of a fixed number of concurrent processes is a well-investigated issue (consider Hierarchical Automata in Section 3.2.1). Most automata support non-deterministic behavior. To the best of our knowledge, automata have never been used as metadata for semantic query optimization. ([47], [91], [112], [65] transfer queries into finite automata and evaluate the automata while events arrive rather than using automata as specifications of streams).

3.2.2 Flowcharts

Flowcharts [18], [76] express algorithms and the flow of application processes. Mainly because of their graphical notation they are easily understandable by non-technical persons and are therefore used e.g. for workflow specification and documentation purposes. They

[5] Tree automata [115], [45] work on terms represented as trees but a node of the tree triggers a transition, not the hole tree (i.e. term).

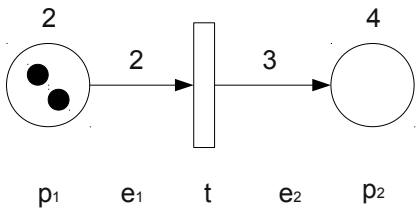

Figure 3.1: An (incomplete) Petri net

were first presented by Frank Gilbreth in 1921. There are different kinds of flowcharts, consider [60] for one of the newer classifications. The most famous examples of flowcharts are UML activity diagrams [104] and Business Process Model [68].

Boxes, diamonds, and directed edges between them are the main components of a flowchart. Boxes represent actions. Such an action can be a call of another program or start of a process, i.e. a hierarchy of flowcharts can be simulated. Diamonds are conditions. A diamond has usually two outgoing edges depending on whether its condition is satisfied or not. Directed edges represent the process flow.

Flowcharts do not explicitly support the concept of state. States can be simulated by boxes in some cases. We are not aware of timed flowcharts, i.e. flowcharts extended by temporal conditions. The process flow represented by flowcharts is not triggered by events.[6] A flowchart is usually deterministic. A flowchart can represent a fixed number of concurrent processes. Additional components are used for this purpose, they are responsible for fork and join of concurrent processes. Flowcharts were not used for semantic query optimization.

3.2.3 Petri Nets

Petri nets are suitable for modeling concurrent distributed systems, in particular expressing complex synchronization schemes. They were developed by Carl Adam Petri in 1939. Since that time much work has been done on mathematical definition, analysis of the expressive power, decidability, complexity, reachability, liveness, and boundedness of Petri nets. [100], [43] survey this research.

A Petri net consists of transitions, places, directed edges, and tokens. Places usually represent resources. Tokens usually model concurrent processes. They move through a Petri net from one place to another illustrating resource allocation by processes. A place may have a capacity restricting the number of tokens the place may contain at most. If the

[6] There is a kind of flowchart, called Event-driven Process Chain [4], [80] using the notion event in the other sense as in this work. Event is a passive element describing the circumstances under which a process works or a state a process results in.

capacity of a place is not restricted the place may contain an arbitrary number of tokens. Directed edges connect transitions with places and vice versa. Edges may not connect two places or two transitions directly. An edge may have a weight. Consider Figure 3.1. For the edge e_1 going out of the place p_1 the weight of e_1 specifies the number of tokens which will be removed from p_1 if the transition t fires. For the edge e_2 going into the place p_2 the weight of e_2 specifies the number of tokens which will be added to p_2 if the transition t fires. The transition t is enabled (i.e. can fire) if p_1 contains enough tokens and p_2 can include all new tokens. The transition t in Figure 3.1 is enabled, if it fires p_1 contains no tokens and p_3 contains 3 tokens.

Petri nets specify resource sharing and synchronization of an arbitrary number of concurrent processes rather than workflows of these processes (there are exceptions such as [117]). Petri nets do not support the notion of state explicitly (however, places can play this role in some cases). Transitions are not triggered by events. Therefore, Petri nets can hardly express relations between events and states.

There are extensions of classical Petri nets with respect to temporal aspects (Timed Petri nets [123], [122], [2]), hierarchy (Hierarchical Petri nets [126], [55], [96]), and both (Hierarchical Timed-extended Petri nets [59], [103]). There are different ways to introduce time into a Petri net: It can be associated with tokens, places, and/or transitions. A hierarchy construct used in Petri nets is called subnet. It is an aggregate of multiple places and transitions.

Petri nets are nondeterministic since multiple transitions can fire at the same time. To the best of our knowledge, they were not used as metadata for semantic query optimization.

3.2.4 Unified Modeling Language

Unified Modeling Language (short UML) [22], [95], [21] had been introduced by the Object Management Group and became an international industrial standardized modeling language for the documentation, specification, visualization, and modification of object-oriented software with respect to its structural and behavioral features. Figure 3.2 contains the classification of 14 kinds of diagrams supported by UML. Since we are interested in workflow specification, only behavior diagrams are described in the following.

UML state machine diagrams are (usually deterministic) Extended Finite State Machines (Section 3.2.1) enriched by actions, output, and hierarchically nested states. Transitions between states are labeled by events and optional actions which are executed when the events trigger the transitions. A set of parameters can be associated with an event.

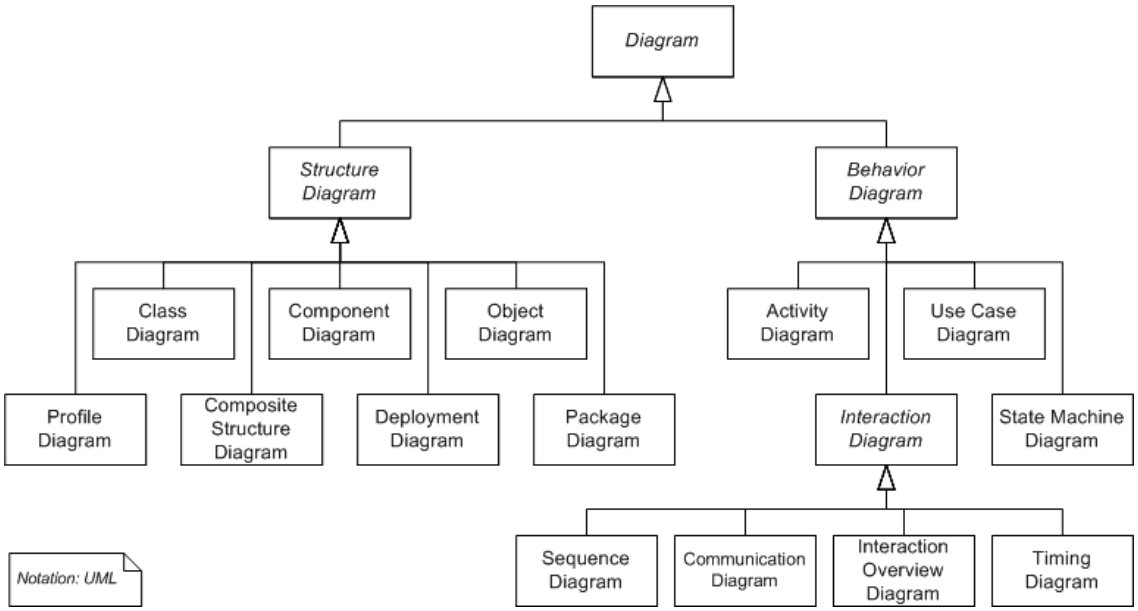

Figure 3.2: Typs of diagrams supported by UML. (Source: `http://en.wikipedia.org/wiki/Unified_Modeling_Language`.)

These parameters restrict the occurrence time and the number of the respective event. Event data is neglected. An action can be update of a variable, execution of an input or an output, call of a function, or creation of an event. Two hierarchical state decompositions are supported: AND and OR. AND-decomposition means that if a composite state is active, then all its substates are also active. A fixed number of concurrent processes can be modeled by this decomposition. UML state machines also provide means for communication and synchronization of the processes. (Exclusive-)OR-decomposition means that if a composite state is active, then exactly one of its substates is also active. As this description shows, UML state machine diagrams visualize programs rather than serve as metadata specifying event streams.

UML use case diagram provides a graphical overview of the systems functionality in terms of actors, their goals (represented by use cases) and dependencies between them to show what system functions are executed for what actor and to what purpose. The same actor can play different roles. *UML interaction diagrams* visualize the communication between the actors. In other words, UML use case and interaction behavior diagrams are not suited to specify event streams. *UML activity diagrams* are a kind of flowcharts described in Section 3.2.2.

3.3 Subsumption of Clauses

This section gives an overview of the algorithms deciding subsumption of clauses.

Let C and D be clauses. Let $|C|$ and $|D|$ denote the number of literals in C and D respectively. Let θ be a substitution, i.e. mapping of variables to terms. Let $C\theta$ denote the application of θ to each literal in C. Let $C\theta \subseteq D$ denote that each literal in $C\theta$ can be found in D.

There are two notions of subsumption [90]:

Definition 1 (**Subsumption of Clauses**). C subsumes D if C logically implies D.

Since implication of clauses is not decidable in general [107], their θ-subsumption is considered.

Definition 2 (θ-**Subsumption of Clauses**). C θ-subsumes D if $|C| \leq |D|$[7] and there is a substitution θ such that $C\theta \subseteq D$.

These definitions are not equivalent. If C θ-subsums D (according to Definition 2) then C subsumes D (according to Definition 1) but not vica versa. Consider the following example from [64] illustrating this point.

Example 4.
$C = \neg p(f(X)) \vee p(X)$ and $D = \neg p(f(f(c))) \vee p(c)$ where p is a predicate symbol, f is a function symbol, X is a variable, and c is a constant. Variables are written as capital letters and constants, predicate symbols, and functional symbols are written as small letters in the following.

C subsumes D since by two instantiations of C and by application of transitivity of implication to these instances, C logically implies D.
More exactly: $C = \neg p(f(X)) \vee p(X) = p(f(X)) \rightarrow p(X)$ is instantiated two times:
1) $X_1 = f(c)$ and $C_1 = p(f(f(c))) \rightarrow p(f(c))$
2) $X_2 = c$ and $C_2 = p(f(c)) \rightarrow p(c)$
Since implication is transitive, $p(f(f(c))) \rightarrow p(f(c))$ and $p(f(c)) \rightarrow p(c)$ imply $p(f(f(c))) \rightarrow p(c) = \neg p(f(f(c))) \vee p(c) = D$.

But C does not θ-subsume D. The variable X cannot be bound to $f(c)$ and c within the same substitution θ. Therefore, there is no substitution θ such that $C\theta \subseteq D$.

There can be many most general substitutions [29] of two clauses as the following example demonstrates.

[7] Some authors, e.g. [34], omit this condition.

Example 5.
Let $C = p(X)$ and $D = p(a) \vee p(b)$. C θ-subsumes D with two most general substitutions $\theta_1 = X \rightarrow a$ and $\theta_2 = X \rightarrow b$.

Subsumption is used as a consistency test and as a redundancy test in inductive logic programming and theorem proving [105]. In the context of semantic query optimization, subsumption is used to determine the relevance of a constraint for a query [31] and for multi-query optimization. In this section, algorithms computing θ-subsumption of clauses are considered. For the sake of brevity, "subsumption" and "substitution" are used as abbreviations for "θ-subsumption" and "most general substitution" respectively in the following.

Subsumption is NP-complete in general [77], i.e. it is a bottleneck of inductive logic programming learners, theorem provers, and semantic query optimizers. Many approaches to speeding up subsumption have been developed so far. They all are similar in their base idea: For each literal c in C find a literal d in D such that c subsumes d and compute all substitutions θ of C and D. However, the algorithms differ in the way they implement this idea and therefore in their efficiency. The complexity of subsumption results from ambiguity of variable binding, i.e. the cost of backtracking in case if a found substitution is wrong. A subsumption algorithm which does not care about this factor is rather expensive (exponential in the size of C and D as shown in [64]). Examples of such algorithms are two classical algorithms: The algorithm of Chang and Lee [34] and the algorithm of Stillman [111] which are briefly described in the following.

The algorithm of Chang and Lee [34] is based on a resolution strategy. First, all variables of D are replaced by constants to obtain D_{gr}. Second, all literals of D_{gr} are negated. $\neg D_{gr}$ is the result. Third, if $C \wedge \neg D_{gr}$ is a contradiction, C subsumes D. Because of the first step, the algorithm does not return a substitution of C and D and is not applicable for our purposes for this reason.

The algorithm of Stillman [111] tests for each literal c_i in C whether there is a literal d_i in D and a substitution θ_i such that $c_i\theta_i \supseteq d_i$. If this is the case, the next modified literal $c_{i+1}\theta_i$ in $C - c_i$ is tested with respect to any literal d_{i+1} in $D - d_i$. If no further substitution θ_{i+1} for $c_{i+1}\theta_i$ and d_{i+1} can be found, backtracking is applied. If each literal in C subsumes a literal in D, the resulting substitution is returned.

In other words, the search space of these classical algorithms is not restricted which makes them inefficient.

The algorithm of Gottlob and Leitsch [63, 64] tackles this problem in the following way: It analyses the connection between variables and predicates in C, divides the literals of

C into independent groups such that no group has common variables with other groups, computes subsumption of D by each group of literals in C separately (if a group consists of multiple literals, in each step, the literal with the maximal number of variables which also occur in other literals of the group is selected), and, finally, constructs the entire substitution for D and C if C subsumes D. The application of the *divide and conquer* strategy reduces the costs of subsumption (in particular, backtracking) to polynomial in many cases.

Kietz and Lübbe [78] noticed that the inefficiency of a subsumption algorithm is caused by the nondeterminism of choosing a substitution in each step. C is determinate with respect to D (or C deterministically subsums D) if there is an ordering of literals of C, such that in each step there is exactly one match that is consistent with all previous matches. Deterministic subsumption is polynomially computable because no backtracking is needed. Since C is non-determinate with respect to D in general, the algorithm divides C into its determinate and non-determinate parts which are also polynomially computable. The non-determinate part of C is further divided into locals corresponding to the independent groups of literals proposed by Gottlob and Leitsch [63, 64]. Locals are identified in polynomial time.

Scheffer, Herbich, and Wysotzki [105, 106] have further developed the idea of deterministic subsumption by taking into account not only single literals but also their context, i.e. other literals with the same variables. Two literals match if they have the same context. This context-based elimination of possible matches allows a proper superset of deterministic clauses to be tested for subsumption in polynomial time. The subsumption of nondeterministic part of a clause is mapped to a clique problem (i.e., search for the largest subset of mutually adjacent nodes in a graph) for which known optimized algorithms can be used.

Scheffer, Herbich, and Wysotzki [105, 106] were not the first who transported the problem of subsumption into a graph framework. Eisinger [53] introduced S-link into Kowalski's connection graph proof procedure [84]. S-link computes the merge of substitutions resulting from subsumptions between literals of clauses. Socker [107] and Kim and Cho [79] proposed pruning strategies that reduce the computational effort of finding a substitution compatible with each substitution yielded by subsumptions between literals of clauses. Bachmair, Chen, C.R.Ramakrishnan and I.V.Ramakrishnan [14] use search trees constructed according to the analysis of variable dependencies between literals (as defined by Gottlob and Leitsch [63, 64]) to separate the search for suitable substitutions from other computational tasks, such as computing substitutions and verifying their consistency.

Maloberti and Sebag [93, 94] face the subsumption problem by mapping it onto Con-

straint Satisfaction Problem (CSP). They developed the subsumption algorithm named *Django*. The algorithm is based on CSP heuristics and returns a binary answer *yes* or *no* as a result of the subsumption test (no substitution if the subsumption between clauses holds).

All subsumption algorithms described so far rely on backtracking and try to limit its effect by properly choosing the candidates for literal unification in each step. In contrast, Ferilli, Mauro, Basile, and Esposito [56] propose the subsumption algorithm avoiding backtracking by constructing multisubstitutions instead of single substitutions in each step. Each variable of a multisubstitution can be mapped to a set of values rather than to a single value. After each literal unification a merge of two multisubstitutions s_1 and s_2 takes place to compute a compatible substitution if s_1 and s_2 share at least one of the substitutions they represent. Otherwise, s_1 and s_2 are incompatible and $false$ is returned. This idea allows polynomial subsumption test in general case.

Chapter 4

Basic Notions

This chapter informally presents the basic notions of CEP used throughout this report. Section 4.1 is devoted to the notions concerning time. Section 4.2 introduces event stream.

4.1 Time

Time is represented by a linearly ordered **set of time points** $(\mathbb{T}, \leq)$, $\mathbb{T} \subseteq \mathbb{Q}^+$ where $\mathbb{Q}^+$ denotes the set of not negative rational numbers. The **set of time intervals** is $\mathbb{TI} = \{i = [b, e] \mid b \in \mathbb{T}, e \in \mathbb{T}, b \leq e\} \cup \{i = [b, e[\mid b \in \mathbb{T}, e \in \mathbb{T}, b < e\}$. For an interval i, $b(i)$ denotes its beginning and $e(i)$ its end, i.e., either $i = [b(i), e(i)]$ or $i = [b(i), e(i)[$. Note that two kinds of time intervals are considered. A **closed time interval**, i.e. a time interval including its bounds. A **right-open time interval**, i.e. a time interval including its beginning but not its end. Right-open time intervals are necessary to express tumbling windows and states. This will be explained in more detail in Section 4.2. Other kinds of intervals are not considered for simplicity reasons. Time intervals with infinite bounds are also not considered since their evaluation is problematic with respect to query termination and garbage collection of event data. The notion of time intervals introduced here is that of connected time intervals. Non-connected time intervals, as needed in many applications, are not explicitly introduced here because they can easily be expressed in the formalism presented (as a disjunction of multiple time intervals).

Using **continuous time** represented as $\mathbb{Q}^+$ instead of **discrete time** represented as, e.g., $\mathbb{N}^+$ has the following advantage. If discrete time is used one have to decide how far apart two sequential discrete time points are. If the granularity of discrete time points is too coarse too many time points which are different according to continuous time fall together according to discrete time. The evaluation of such events can be inefficient in

some cases. For example, IHTA (Chapter 6) consider all possible permutations of events with the same occurrence time. The computation of these permutations is rather expensive in general (factorial of n if n is the number of events with the same occurrence time). If the granularity of discrete time points is too fine their time point based evaluation also becomes a bottle neck.

A **window** is a repeating time interval. The following kinds of windows are considered in this work [50]:

- **Unbounded window** is a window corresponding to the time interval $[0, \infty)$.
- **Now window** is a window containing only the current time point.

  It corresponds to the time interval $[now, now]$ where now denotes the current time point.
- **Sliding window** is a continuously moving or jumping overlapping window.

  For two sequential sliding windows w_1 and w_2 it holds that (1) w_1 and w_2 have the same size, i.e. $e(w_1) - b(w_1) = e(w_2) - b(w_2)$, (2) w_2 begins after w_1 has begun, i.e. $b(w_1) < b(w_2)$, and (3) w_2 ends after w_1 has finished, i.e. $e(w_1) < e(w_2)$. Therefore, sliding windows overlap but do not coincide. A sliding window can be both closed and right-open. Each time point belongs to at least one sliding window. There are two kinds of sliding windows:

  - *Simple sliding window* moves continuously.

    It corresponds to the time interval $[now - d, now]$ (or $[now - d, now[$) where now denotes the current time point and d is a duration such as *2 hours* or *1 minute*.
  - *Sliding window moving at fixed granularity* does not move continuously but in steps of fixed length.

    For two sequential sliding windows w_1 and w_2 moving at fixed granularity d the second and the third conditions are defined as follows: (2) $b(w_1) + d = b(w_2)$, (3) $e(w_1) + d = e(w_2)$ where d is a duration. In other words, this kind of sliding window moves in steps of length d. d is usually smaller as the size of the window, i.e. $e(w_1) - b(w_1) = e(w_2) - b(w_2) > d$.
- **Tumbling window** is a jumping non-overlapping window.

  For two sequential tumbling windows w_1 and w_2 it holds that (1) w_1 and w_2 have the same size, i.e. $e(w_1) - b(w_1) = e(w_2) - b(w_2)$, (2) w_1 and w_2 are right-open time intervals, and (3) w_2 begins as soon as w_1 ends, i.e. $e(w_1) = b(w_2)$. Therefore,

tumbling windows do not overlap. Each time point belongs to exactly one tumbling window. Tumbling window of size s can be seen as a right-open sliding window moving at fixed granularity s where s is a duration.

- **Other**, for example, $[f(i), g(i)]$ (or $[f(i), g(i)[$) where $f()$ and $g()$ are functions and i is a set of time intervals. i can also be the current time point.

4.2 Event Stream

Let *GroundAtoms* be a set of ground atoms of a first order language. An **event** e is tuple of $a \in GroundAtoms$ and $i \in \mathbb{TI}$, written a^i. a carries **event data** of e. i is a closed time interval denoting the **occurrence time** of e.

Let *Events* be the set of events, i.e. $Events = GroundAtoms \times \mathbb{TI}$. $E \subseteq Events$ is called an event stream. All events of an event stream are totally ordered according to their occurrence time. Therefore one speaks about **sequences of events**.

We distinguish between simple and complex events. An event e is a **simple event** if it is not derived from a sequence of other events of the event stream. In this case the occurrence time of e is usually a time point. An event e is a **complex event** if it is derived from a sequence of simple or complex events. The occurrence time of e is a time interval comprising the occurrence times of all events of the sequence. To specify the derivation of complex events, queries in an **event query language** are used. Consider [51] for a recent overview of the existing event query languages.

As motivated by the online auction use case in Section 2.1.1, application state is a very important notion in CEP. A **state** s is a tuple of $d \in GroundAtoms$ and $j \in \mathbb{TI}$, written d^j. d carries **state data** of s. j is a right-open time interval denoting the **validity time** of s. Like complex events, states are derived from multiple events and/or states. In contrast to events which are "seen" by the event processing engine as soon as they have happened (i.e. at the end of their occurrence time), states can be processed as soon as they begin (i.e. at the beginning of their validity time). This implies that the end of the validity time of a state may be unknown when the state is processed.

A **relative timer event or state** is an event or a state the time interval of which is defined relatively to the time interval of another event or state. This idea is adapted from [50].

Example 6.
The relative timer event r specified as `event r: extend[i, d]` extends the occurrence time of

the event (or the validity time of the state) i by the length d. More exactly: $b(r) := b(i)$ and $e(r) := e(i) + d$.

Let $States$ be the set of states, i.e. $States = GroundAtoms \times \mathbb{TI}$. $S \subseteq Events \dot{\cup} States$ denotes a **stream** of events and states.

Chapter 5

StreamLog

The semantic optimization method for CEP introduced in this report is independent from an event query language. To illustrate it, event rules are expressed in SreamLog. This chapter is devoted to the language.

A **StreamLog rule** has the form $i : h \leftarrow b$ where:

- i is the **rule evaluation time**. It is a time interval, i.e. $i \in \mathbb{TI}$. Rules with the same evaluation time belong to the same **module**. During this time all rules of a module are evaluated, they can be suspended otherwise. i can be a non-repeating time interval, an event, an application state, or a repeating window (consider Chapter 4).
- h is the rule head. It specifies events or states derived by the rule. h is an atom.
- b is the rule body. It specifies events and states from which a new event or state will be derived. b is a conjunction or a disjunction of positive or negative literals and conditions on them.

To differentiate between atoms representing events and atoms representing states, atoms can be prefixed by the key works *event:* and *state:* respectively (as done in Event Stream Constraint Language described in Chapter 7). However, for simplicity reasons we assume that the sets of the outermost labels of atoms representing events and atoms representing states are disjoined.

Consider a simple example of a StreamLog rule in Listing 5.1.

Listing 5.1: StreamLog rule computing the number of bids per item during an auction

```
1 auction(auctionID(A)):
2   bidNumber(auctionID(A), itemID(I), number(count(A,I))) ←
3     bid(auctionID(A), itemID(I), bidderID(B), value(V))
```

The rule computes the number of bids per item in an auction. The atom in Line 1 specifies the time interval during which the rule is evaluated. This time interval is the duration of an auction, i.e. an application state. Line 3 is the rule body. It contains one non-ground atom specifying queried events. The variables of the atom are bound if the atom matches an event. This substitutions are used while constructing new, complex events by the rule head in Line 2.

Consider a more involved StreamLog rule in Listing 5.2.

Listing 5.2: StreamLog rule identifying uninteresting items in each auction

```
1  auction(auctionID(A)):
2    uninterestingItems(auctionID(A), itemID(I)) ←
3      itemDescription(auctionID(A), itemID(I), description(D), value(S)) ∧
4      ( ¬ sell(auctionID(A), itemID(I), bidderID(B), value(E)) ∨
5        ( sell(auctionID(A), itemID(I), bidderID(B), value(E)) ∧
6          bidNumber(auctionID(A), itemID(I), number(N)) ∧
7          N ≤ 3 ∧
8          E < S + S/100
9        )
10     )
```

The rule identifies uninteresting items in each auction. More exactly, either unsold items (Line 4) or items for which there were not more than 3 bids and the end price did not raise higher than one percent of the start price (Lines 5–9).

Consider a StreamLog rule in Listing 5.3. Its evaluation time is specified by means of a sliding window of size one hour. The granularity of the window is half an hour.

Listing 5.3: StreamLog rule computing the average temperature in an area during each hour

```
1  Range 1h Slide 30min:
2    averageTemp(area(A), value(avg(V))) ←
3      temp(area(A), value(V))
```

The rule computes the average temperature in each area during each hour and reports the result every half an hour by constructing a new complex event *averageTemp* carrying the identifier of the area and the computed average temperature.

The grammar of StreamLog is defined as follows:

$$
\begin{array}{lcl}
\mathit{ModuleSet} & ::= & \mathit{Module}* \\
\mathit{Module} & ::= & \mathit{EvalTime}\ \text{" : "}\ \mathit{Rule}+ \\
\mathit{Rule} & ::= & \mathit{Atom}\ \text{" } \leftarrow \text{ "}\ \mathit{Query} \\
\mathit{Query} & ::= & \mathit{AtomicQuery} \\
 & & |\ (\ \text{"("}?\ \mathit{Query}\ \text{" } \wedge \text{ "}\ \mathit{Query}\ \text{")"}?\ ) \\
 & & |\ (\ \text{"("}?\ \mathit{Query}\ \text{" } \vee \text{ "}\ \mathit{Query}\ \text{")"}?\ ) \\
\mathit{AtomicQuery} & ::= & \mathit{Literal} \mid \mathit{RelTimer} \mid \mathit{Condition} \\
 & & \\
\mathit{Literal} & ::= & (\mathit{EvalTime}\ \text{" : "})?\ \text{"}\neg\text{"}?\ (\mathit{Identifier}\ \text{" : "})?\ \mathit{Atom} \\
\mathit{EvalTime} & ::= & \mathit{Identifier} \mid (\ (\mathit{Identifier}\ \text{" : "})?\ \mathit{Atom}\ ) \mid \mathit{Window} \\
\mathit{Window} & ::= & \mathit{Unbounded} \mid \mathit{Now} \mid \mathit{Sliding} \mid \mathit{Tumbling} \mid \mathit{Other} \\
\mathit{Unbounded} & ::= & \text{"}\mathit{unbounded}\text{"} \\
\mathit{Now} & ::= & \text{"}\mathit{now}\text{"} \\
\mathit{Sliding} & ::= & \text{"}\mathit{Range}\text{"}\ \mathit{Duration}\ \text{"}\mathit{Slide}\text{"}\ \mathit{Duration} \\
\mathit{Tumbling} & ::= & \text{"}\mathit{Range}\text{"}\ \mathit{Duration} \\
\mathit{Other} & ::= & \text{"["}\ \mathit{String}\ \text{","}\ \mathit{String}\ \text{"]"} \\
 & & \\
\mathit{RelTimer} & ::= & (\mathit{Identifier}\ \text{" : "})?\ \mathit{RelTimerSpec}\ \text{"["}\ \mathit{Identifier}\ \text{","}\ \mathit{Duration}\ \text{"]"} \\
\mathit{RelTimerSpec} & ::= & \text{"}\mathit{extend}\text{"} \mid \text{"}\mathit{shorten}\text{"} \mid \text{"}\mathit{from\text{-}end}\text{"} \mid \text{"}\mathit{from\text{-}start}\text{"} \\
 & & |\ \text{"}\mathit{extend\text{-}begin}\text{"} \mid \text{"}\mathit{shorten\text{-}begin}\text{"} \\
 & & |\ \text{"}\mathit{shift\text{-}forward}\text{"} \mid \text{"}\mathit{shift\text{-}backward}\text{"} \\
 & & |\ \text{"}\mathit{from\text{-}end\text{-}backward}\text{"} \mid \text{"}\mathit{from\text{-}start\text{-}backward}\text{"} \\
 & & \\
\mathit{Condition} & ::= & \mathit{Exp}\ \mathit{CompOp}\ \mathit{Exp} \\
\mathit{Exp} & ::= & \mathit{Number} \mid \text{"`"}\mathit{String}\text{"'"} \mid \mathit{Duration} \\
 & & |\ \mathit{Variable} \mid (\ \text{"}\mathit{b}(\text{"}\mathit{Identifier}\text{")"}\ ) \mid (\ \text{"}\mathit{e}(\text{"}\mathit{Identifier}\text{")"}\ ) \\
 & & |\ (\ \mathit{Exp}\ \mathit{ArithOp}\ \mathit{Exp}\ ) \mid (\ \text{"("}\mathit{Exp}\text{")"}\ ) \\
\mathit{CompOp} & ::= & \text{" = "} \mid \text{" } \neq \text{ "} \mid \text{" < "} \mid \text{" } \leq \text{ "} \mid \text{" > "} \mid \text{" } \geq \text{ "} \\
\mathit{ArithOp} & ::= & \text{" * "} \mid \text{"/"} \mid \text{" - "} \mid \text{" + "} \mid \text{"}\mathit{mod}\text{"} \\
\mathit{Duration} & ::= & (\mathit{Number}\ (\text{"}\mathit{month}\text{"}\mid\text{"}\mathit{months}\text{"}))? \\
 & & (\mathit{Number}\ (\text{"}\mathit{week}\text{"}\mid\text{"}\mathit{weeks}\text{"}))? \\
 & & (\mathit{Number}\ (\text{"}\mathit{day}\text{"}\mid\text{"}\mathit{days}\text{"}))?\ (\mathit{Number}\ (\text{"}\mathit{hour}\text{"}\mid\text{"}\mathit{hours}\text{"}))? \\
 & & (\mathit{Number}\ \text{"}\mathit{min}\text{"})?\ (\mathit{Number}\ \text{"}\mathit{s}\text{"})?\ (\mathit{Number}\ \text{"}\mathit{ms}\text{"})?
\end{array}
$$

An atom appearing in the body of a StramLog rule represents an event or a state being queried. An atom appearing in the head of a StramLog rule represents an event or a state being derived. If an atom appears in a rule head it may contain aggregation functions and grouping.

A StreamLog identifier represents an event and a state identifier. All identifiers used in a rule must be declared in the rule (grammar rules *Literal* and *EvalTime*).

If i and j are identifiers, temporal conditions such as i before j can be expressed in StreamLog by means of the functions $b()$ and $e()$ returning the beginning and the end of a time interval (compare Section 4.1) and comparison operators, i.e. i before j is equivalent to $e(i) < b(j)$.

All variables used in the conditions and the head of a rule must appear in a positive literal in the body of the rule.

As motivated in Section 4.1, time is represented by a linearly ordered set of time points $(\mathbb{T}, \leq)$, $\mathbb{T} \in \mathbb{Q}^+$. The time interval of one second is denoted as $1s$. The time interval of one millisecond is therefore $\frac{1}{1000}s$, we write $1ms$ as a shortcut. The time interval of one minute is $60s$, we write $1min$ as a shortcut. All other possibilities to specify the duration of a time interval are computed analogously.

Evaluation time must be specified for each rule. Rules with the same evaluation time belong to the same module. A module is suspended if its evaluation time has not begun yet or is already over. Evaluation time can be also specified for a literal. The evaluation time of a rule must cover all evaluation time intervals of its literals.

Evaluation time of a rule or a literal can be a (relative timer) event or state (first two alternatives of the grammar rule *EvalTime*),[1] or a repeating or non-repeating window (the last alternative). Compare the description of the different kinds of windows in Section 4.1 with the respective grammar rules.

The declarative and the operational semantics of StreamLog are defined in [52]. They are skipped in this report for the sake of brevity.

[1] The idea of relative timer events and its specification is adopted from [50].

Chapter 6

Instantiating Hierarchical Timed Automata

This chapter is devoted to the Instantiating Hierarchical Timed Automata (short IHTA). IHTA fulfill the requirements to a formalism capturing the semantics of CEP applications (Section 2.3) and will be used as metadata for the semantic optimization of CEP. This kind of metadata is inspired by the event-driven applications with complex workflow (like the one presented in Section 2.1.1). To the best of our knowledge, automata have never been used in this way. However, IHTA are not restricted to this purpose and can also be used, for example, for stream verification.

Section 6.1 briefly describes the main features of IHTA. Section 6.2 and Section 6.3 are devoted to the syntax and the semantics of IHTA respectively.

6.1 Main Features

Consider the auction use case described in Section 2.1.1. The graphical representation of its workflow is shown in Figure 6.1. The model is surely very simplified in two respects. First, event-based systems usually work on a much larger set of events. Second, events usually carry much more data. But for the purpose of the illustration of IHTA, this simplified example is suitable.

The main features of IHTA are the following:

1. IHTA are *stateful*. State changes are triggered by events and/or delays.
2. IHTA are *independent from the language specifying its transition labels*. In Chapter 5, StreamLog is definied as an example of such a language used in this work. The

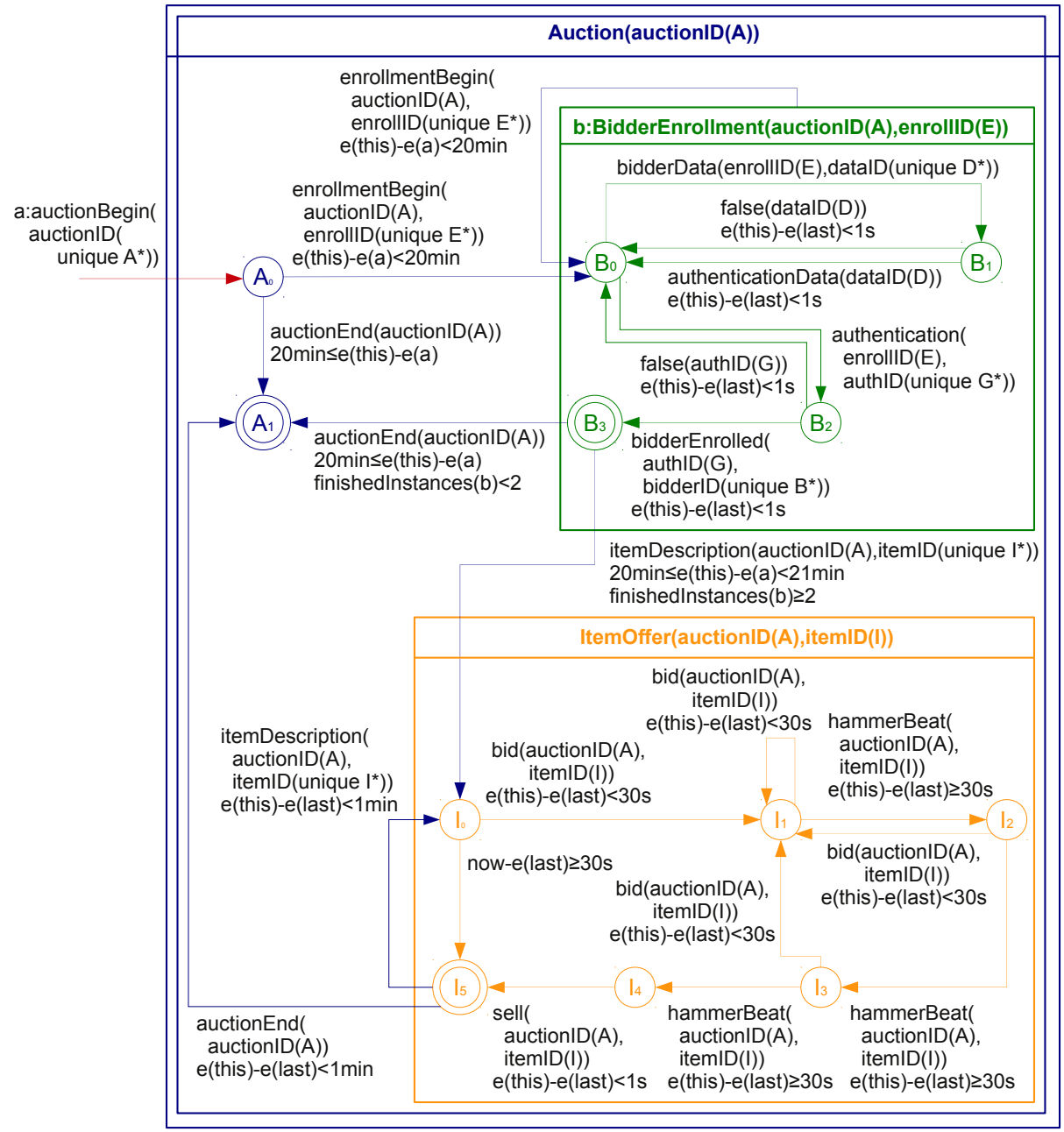

Figure 6.1: Auction modeled as IHTA.

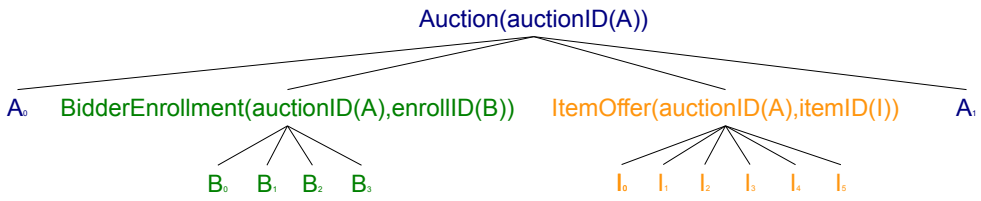

Figure 6.2: Tree of States of the IHTA in Figure 6.1

event queries in StreamLog allow access to event data. The temporal constraints are expressed on the occurrence time of matched events and/or on the current time. Other kinds of constraints are not considered in this work to keep transition labels of IHTA readable.

3. IHTA are *expressive*. They capture temporal, causal, cardinality, and data dependencies between events and states in a readable way. The expressiveness of IHTA and the kinds of constraints captured by the automata depend on the language specifying their transition labels.

4. IHTA are *non-deterministic* and *able to work with events with the same occurrence time*.

5. IHTA are *modular* which implies abstraction, readability, (ex-) changeability, and reuse of the modules.

6. IHTA can *represent an arbitrary number of concurrent processes* by instantiation of module specifications.

6.2 Syntax

This section is devoted to the syntax of IHTA. IHTA consist of states which may contain IHTA and transitions between states. Section 6.2.1 describes states, Section 6.2.2 is devoted to transitions. Section 6.2.3 presents the language StreamLog specifying the transition labels in the examples used in this work. But IHTA are language independent in general.

6.2.1 States

There are four kinds of states namely atomic, non-atomic, start, and end states. They are motivated and explained in this section.

Modularization, encapsulation and the need to model an arbitrary number of concurrent processes which are motivated in Section 2.3.3 are attained by introducing hierarchy of states, i.e. the possibility to represent IHTA as a singe state of another IHTA. A state s containing IHTA $\mathcal{I}$ is an abstraction of $\mathcal{I}$ which runs while s is active. The states of the IHTA in Figure 6.1 build a tree displayed in Figure 6.2.

States containing IHTA are called **non-atomic states**. They are represented as rectangles. States which do not contain IHTA are called **atomic states** and corre-

spond to states in classical Finite State Automata [108]. Atomic states are depicted as cycles. Each state is either atomic or non-atomic. In Figure 6.1, the non-atomic state *Auction(auctionID(A))* is a **superstate** of the non-atomic state *BidderEnrollment(auctionID(A),enrollID(E))*. The atomic state B_0 is a **substate** of the non-atomic state *BidderEnrollment(auctionID(A),enrollID(E))* which is a substate of the non-atomic state *Auction(auctionID(A))*. Trivially, a non-atomic state can be a substate and a superstate. Atomic states can only be substates.

Both atomic and non-atomic states can be either **start**, or **end**, or neither start nor end states. A start state is a state of IHTA which is entered first. For example, A_0, B_0, and I_0 are the start states of their respective IHTA in Figure 6.1. Each IHTA have at least one start state (IHTA are not deterministic). An end state is a last state of IHTA. *Auction(auctionID(A))*, A_1, B_3, and I_5 are the end states of their respective IHTA. Each IHTA have at least one end state. An end state is depicted as a double circle if it is an atomic state or as a double rectangle if it is a non-atomic state. All the other states in Figure 6.1 are neither start nor end states.

Section 2.3.3 motivates the requirements of concurrent processes and processes serving as abstraction of subprocesses. Atomic and non-atomic states of IHTA meet these requirements. Atomic states represent the state of elementary processes, while non-atomic states represent the state of higher processes containing elementary processes. Non-atomic states allow multiple substates to be active concurrently. The information represented by an atomic state is very basic. With an increasing number and level of concurrent substates, a non-atomic state represents information which is increasingly complex and abstract.

Statecharts [71] and Hierarchical Timed Automata (HTA) [41] support the notion of concurrent processes. However, only a fixed number of concurrent processes can be modeled by them. In the online auction use case the total number of concurrent bidder enrollment processes is, in contrast, neither known beforehand nor bound. This motivates one of IHTA's extensions (compared to Statecharts and HTA) called instantiation.

Instantiation draws a distinction between the *specification* of a non-atomic state and its *instances*. An instance i of a non-atomic state s has exactly one active state which is one of the direct substates of s. There might be zero or several instances of a non-atomic state at the same time. This resembles the relation in object oriented programming: Objects are instances of classes and there might be any number of objects instantiating a single class. In the example in Figure 6.1, each instance of the non-atomic state *BidderEnrollment(auctionID(A),enrollID(E))* has an active state which is B_0, B_1, B_2, or B_3.

Instances form a tree similarly to the tree of states. This gives rise to the notion of **subinstance** and **superinstance**. There is a connection between the tree of states and

the tree of their instances: If b is an instance of the non-atomic state B, a is an instance of the non-atomic state A, then b is a subinstance of a if and only if B is a substate of A.

6.2.2 Transitions

The online auction use case is a system which changes its state in reaction to events. For example, three consecutive *hammerBeat* events bring the item sale into a state in which no further bids are allowed. Transitions specify the triggering conditions and the effect of these reactions.

In classical automata models such as Finite Automata [108], transitions map **source states** to **target states**. Transitions can fire when their label matches a symbol from the input word, after which the automaton is in the target state.

The classical concepts were extended considerably. For example, Timed Automata [5] use timing conditions on clocks as additional firing conditions and allow timers to be reset as reaction to the firing of a transition. Hierarchical Timed Automata (HTA) [41] introduce additional types of transitions (called pseudo transitions) which are needed for managing the state hierarchy.

IHTA transitions extend the expressiveness of transitions of Timed Automata and HTA. Transition labels of IHTA allow matching incoming events by event queries accessing the data of events, temporal constraints on the beginning and end of the occurrence time of matched events, and constraints on the number of instances.

Transition Role

According to their role, transitions are classified into instantiating and terminating.

An **instantiating transition** t creates an instance of each non-atomic state t goes into. Consider Figure 6.1. Transitions between states A_0 and B_0, *BidderEnrollment(auctionID(A),enrollID(E))* and B_0 create instances of the non-atomic state *BidderEnrollment(auctionID(A),enrollID(E))*. The target state of an instantiating transition is a start state.

A **terminating transition** t terminates all instances of all non-atomic states t goes out of. Consider Figure 6.1. If the transition between states B_3 and A_1 fires all instances of the state *BidderEnrollment(auctionID(A),enrollID(E))* are terminated. The source state of a terminating transition is an end state.

A transition can be either only instantiating (between *BidderEnroll-*

ment(auctionID(A),enrollID(E)) and B_0), or only terminating (between I_5 and A_1), or both instantiating and terminating (between B_3 and I_0), or neither instantiating nor terminating (between I_1 and I_2).

Transition Trigger

According to their trigger, transitions are classified into event and delay transitions.

Transitions triggered by events are called **event transitions**. An event transition is labeled with a non-optional positive or negative atomic *event query* and optional sets of *temporal* and *termination constraints*. The language StreamLog specifying transition labels is described below.

Event queries are used for matching events. Event queries may contain *variables*. A variable X is local for the path X is declared within until the next declaration of X.

In order to differentiate between a variable declaration and a reference to the recent variable binding, the former are flagged with *. For example, X^* is a declaration of the variable X and X (without *) is a reference to the recent binding of X. The set of flagged variables and the set of unflagged variables of an event query must be disjoined.

Each variable must be declared in IHTA. A variable can be referenced in the instance i it is declared within and in the subinstances of i but not in the superinstances of i.

Each instance of a non-atomic state saves its current variable bindings. Let i be an instance with the variable bindings $VarBindings$. An event transition t fires in i if the event query q of t matches some event e of the stream with respect to $VarBindings$, i.e. $q \cdot VarBindings \cdot \sigma = e$ where σ is the set of mappings of the flagged variables of q to the respective values of e. Negated event queries may not contain variable declarations. See Definition 12 for the complete specification of the transition firing conditions.

Example: Consider the cycle involving states B_0 and B_1 in Figure 6.1. The event query *bidderData(enrollID(E),dataID(unique D*))* is used for matching events e containing the attribute *enrollID* the value of which is equal to the recent binding of the variable E. (E has been bound while firing the transition between A_0 and B_0 or the transition between *BidderEnrollment(auctionID(A),enrollID(E))* and B_0.) e must also contain the attribute $dataID$, and the variable D is bound to the respective value of e. The event queries *authentificationData(dataID(D))* and *false(dataID(D))* reference this binding of D.

An event query can be prefixed by an *event identifier* which references the event matched by the query. If q is a positive event query and j is an event identifier, $j : q$ is the declaration of j. An event identifier j is local for the path j is declared within until

the next declaration of j. Each instance saves its current event identifier bindings.

The functions $b(j)$ and $e(j)$ return the beginning and the end of the occurrence time of the recent event referenced by j (compare Chapter 4). These functions are used in the temporal constraints of IHTA. Each event identifier must be declared in an instance i before it is used in the temporal constraints of i or subinstances of i.

Since the event identifier bindings of an instance are updated *after* a transition t has fired in the instance, there would be no way for the temporal constraint c of t to identify the event which is being matched by the event query q of t. Therefore the auxiliary event identifier *this* is introduced. It references the currently matched event.

Modularization by hierarchical states, is an important design feature of IHTA. Therefore, the identifier environment's scope is local to an instance and its subinstances. In other words, an instance i cannot use event identifiers which are declared in the subinstances of i but not declared in i or a superinstance of i. To allow limited temporal constraints in transitions crossing hierarchy levels upwards, the auxiliary event identifier *last* is introduced. *last* identifies the event which was matched by the last fired event transition.

Besides, *this* and *last* are useful as "syntactic sugar", increasing readability. Thanks to *this* and *last* fewer event identifiers have to be created manually. The importance of *last* becomes especially clear when considering states with many ingoing transitions, like the state I_1 of Figure 6.1. The event transition between states I_1 and I_2 does not have to care about the various transitions which lead to the state I_1 and events matched by them. There are numerous temporal constraints in Figure 6.1 using *this* and *last*, e.g. $e(this) - e(last) < 1\text{s}$.

Please note that *this* and *last* are not defined if the current transition or the last fired transition has no event query or has a conjunction or a disjunction of multiple event queries.

Event identification in not a new feature of the event queries of IHTA. Some event query languages, e.g. XChange$^{\text{EQ}}$ [26, 50, 52], support this feature to facilitate short but expressive temporal conditions on events as needed in IHTA.

Analogously to event queries which can be prefixed by event identifiers, a state specification can be prefixed by a *state identifier*, for example *BidderEnrollment(auctionID(A),enrollID(E))* is prefixed by the identifier b. This is the declaration of the state identifier b.

State identifiers are used in termination constraints of terminating transitions. Termination constraints restrict the number of finished instances (i.e. instances in a particular

end state) of a particular non-atomic state.

Example: The terminating transition between states B_3 and I_0 is labeled by the termination constraint $finishedInstances(b) \geq 2$. This means that the number of instances of the non-atomic state referenced by b (i.e. of *BidderEnrollment(auctionID(A),enrollID(E))*) which are in the state B_3 must be at least 2 expressing that items are presented in an auction for which at least two bidders are enrolled. Otherwise the auction ends, consider the transition between states B_3 and A_1 in Figure 6.1.

Since state identifiers are used in terminating transitions crossing multiple levels of state hierarchy upwards, they are global within the run (tree of instances) they are defined in. In other words, state identifiers which are declared in an instance i can be used in i and in all sub- and superinstances of i (in contrast to variables and event identifiers which cannot be used in the superinstances of i).

Delay transitions are different from event transitions in that they are triggered by a timeout and not by an event. Therefore, delay transitions are labeled by temporal and/or termination constraints and not by an event query. *now* denotes the current time.

Example: In Figure 6.1, there is a delay-transition between the states I_0 and I_5. It expresses that the current item sale is canceled when there are no bids within 30 seconds after the item description.

6.2.3 Changes of StreamLog

IHTA are defined to be fully language independent in the sense that any (complex) event query language can be used to specify the transition labels of IHTA. In this work the complex event query language StreamLog presented in Chapter 5 is used for this purpose. This section is devoted to the changes of the language which are needed to specify transition labels. These changes are:

1. No conjunction or disjunction of multiple literals in a transition label
2. $*$-*notation for variables* to differentiate between a declaration of a variable (with $*$) and a reference to its recent binding (without $*$)
3. *Key word unique for variables* to denote that this variable can be bound to a value only once
4. *Variable now* denoting the current time point
5. *Event identifiers this and last* referencing the currently or recently matched event respectively

6. *Termination constraints* to restrict the number of instances in an end state of a particular nonatomic state

These changes are explained in the following.

Transitions of IHTA are labeled with StreamLog queries. If a StreamLog query matches its respective transition fires. A StreamLog query is a conjunction of (1) a positive or negative atomic event query, (2) temporal constraints, and (3) termination constraints. Consider the examples in the previous section.

Atomic event queries describe the types of matched events, their attributes and attribute values. An event query can contain variable declarations and references to the recent variable bindings. A variable flagged with * is the declaration of the variable, i.e. this variable is bound while matching events. A variable without * is the reference to the recent binding of the variable. A variable prefixed with the keyword *unique* will never be bound more than once to the same value. A positive event query prefixed with an event identifier j is the declaration of the event identifier j. j references the event matched by the query.

Temporal constraints are constraints on the occurrence time of matched events or/and on the time at which the current transition can be fired. Temporal constraints can contain *now* which refers to the current time point, 1-ary functions $b(j)$ and $e(j)$ returning the beginning and the end of the occurrence time of the event referenced by j. Two special event identifiers *this* and *last* can be used. *this* references the currently matched event, *last* references the event matched by the last fired event transition in the considered instance. The value of *now* and the values returned by the functions are constrained by means of the binary operators $=, \neq, <, \leq, >, \geq$.

Termination constraints restrict the number of finished instances of a non-atomic state of IHTA. Termination constraints contain 1-ary functions $allInstances(s)$ and $finishedInstances(s)$ returning the number of all instances and the number of finished instances (i.e. instances in the end state which is the source state of the current transition) of the non-atomic state referenced by s. The values returned by these functions are constrained by means of the binary operators $=, \neq, <, \leq, >, \geq$.

Let l be a StreamLog query and t be the transition labeled by l. As explained above, if l contains an event query, t is an event transition. If l contains temporal and/or termination constraints and no event query, t is a delay transition. If l contains termination constraints, t is a terminating transition.

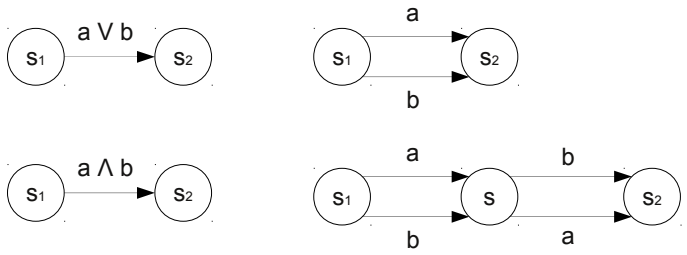

Figure 6.3: Simulation of disjunction and conjunction of two elements a and b of a transition label of IHTA.

The grammar of StreamLog (defined Chapter 5) is extended by the following rules:

$$
\begin{aligned}
TransitionLabel &::= Literal?\ Condition*\ TermConstraint* \\
Exp &::= Number \mid \text{"`"}String\text{"'"} \mid Duration \mid \text{"}now\text{"} \\
&\quad \mid Variable \mid \text{"}b(\text{"}Identifier\text{"})\text{"} \mid \text{"}e(\text{"}Identifier\text{"})\text{"} \\
&\quad \mid Exp\ ArithOp\ Exp \mid \text{"}(\text{"}Exp\text{"})\text{"} \\
EventIdentifier &::= Identifier \mid \text{"}this\text{"} \mid \text{"}last\text{"} \\
StateIdentifier &::= Identifier \\
TermConstraint &::= (\text{"}finishedInstances\text{"} \mid \text{"}allInstances\text{"}) \\
&\quad \text{"}(\text{"}\ StateIdentifier\ \text{"})\text{"}\ CompOp\ Number
\end{aligned}
$$

Note that a transition label in StreamLog allows neither disjunction nor conjunction of multiple elements (e.g., literals). Both can be simulated as shown in Figure 6.3. A disjunction of n elements leads to n additional transitions in IHTA. A conjunction of n elements causes $n!$ additional transitions and $n-1$ auxiliary states in IHTA. Nevertheless, we decided not to allow arbitrary disjunction and conjunction in transition labels for the following reasons: (1) Arbitrary disjunction and conjunction can be simulated as described above. (2) IHTA are similar to the classical Nondeterministic Finite Automata [108] allowing only one symbol of the alphabet per label. (3) The event identifiers *this* and *last* are not defined if the label of the current or last transition contains a disjunction or a conjunction of multiple atomic event queries. Note that negation of an atomic event query cannot be simulated. Therefore this feature is supported by the language.

Of course, multiple extensions of the language are thinkable and preferable for many applications. For example, introduction of other elements into a transition label such as conditions on event data. But for this work we limit the language to the properties described above. Nevertheless the language is quite expressive and keeps IHTA readable.

6.3 Semantics

In this section, the semantics of IHTA is formally defined. Each formal definition is preceded by an informal description and examples. Section 6.3.1 defines IHTA formally. Section 6.3.2 presents the automaton configuration and the rules for its modification.

6.3.1 Definition of IHTA

Let *Literals* be the set of positive or negative atomic event queries, *Conditions* be a set of temporal constraints, and *TermConstraints* be a set of termination constraints. The elements of these three sets are expressed in StreamLog as defined in the previous section.

Definition 3 (Instantiating Hierarchical Timed Automaton (IHTA)). IHTA $\mathcal{I}$ is a tuple $(S, Start, End, children, T)$ where

- S is a finite set of states.
- $Start \subseteq S$ is a set of start states.
- $End \subseteq S$ is a set of end states.
- $children : S \to 2^S$ maps each state $s \in S$ to the set of its direct substates (which may be empty).[1] The function gives rise to a tree of states with root which is the root state of IHTA (Definition 5).
- $T \subseteq S \times (Literals \times Conditions \times TermConstraints) \times S$ is the set of transitions. A transition connects two states s and s', has an optional positive or negative atomic event query q, an optional set of temporal constraints c, and an optional set of termination constraints f. We use the notation $t : s \xrightarrow{q,c,f} s' \in T$, where q, c, f can be omitted to express that they are necessarily absent. However transitions with empty labels are not allowed. A transition t without source state and with a target state $s' \in Start$ is called *enter transition*, denoted $t : \emptyset \xrightarrow{q,c} s' \in T$.

Example: The *Item Offer* represented graphically in Figure 6.1 is specified formally by the IHTA $\mathcal{I}_{ItemOffer} = (S, Start, End, children, T)$ where

[1] For a set X, the notation 2^X represents the power set of X, i.e. the set of all subsets of X, formally: $2^X \stackrel{\text{def}}{=} \{x | x \subseteq X\}$.

$$
\begin{aligned}
S :=\ & \{ItemOffer(auctionID(A), itemID(I)), I_0, I_1, I_2, I_3, I_4, I_5\} \\
Start :=\ & \{I_0\} \\
End :=\ & \{I_5\} \\
children :=\ & \{ItemOffer(auctionID(A), itemID(I)) \mapsto \{I_0, I_1, I_2, I_3, I_4, I_5\}\} \cup \{x \mapsto \emptyset | x \in \{I_0, I_1, I_2, I_3, I_4, I_5\}\} \\
T :=\ & (I_0, (bid(auctionID(A), itemID(I)), e(this) - e(last) < 30s, \emptyset), I_1), \\
& (I_0, (\emptyset, now - e(last) \geq 30s, \emptyset), I_5), \\
& (I_1, (bid(auctionID(A), itemID(I)), e(this) - e(last) < 30s, \emptyset), I_1), \\
& (I_1, (hammerBeat(auctionID(A), itemID(I)), e(this) - e(last) \geq 30s, \emptyset), I_2), \\
& (I_2, (bid(auctionID(A), itemID(I)), e(this) - e(last) < 30s, \emptyset), I_1), \\
& (I_2, (hammerBeat(auctionID(A), itemID(I)), e(this) - e(last) \geq 30s, \emptyset), I_3), \\
& (I_3, (bid(auctionID(A), itemID(I)), e(this) - e(last) < 30s, \emptyset), I_1), \\
& (I_3, (hammerBeat(auctionID(A), itemID(I)), e(this) - e(last) \geq 30s, \emptyset), I_4), \\
& (I_4, (sell(auctionID(A), itemID(I)), e(this) - e(last) < 1s, \emptyset), I_5)
\end{aligned}
$$

IHTA usually specify a substream of events, not the whole event stream E. Schema of IHTA describes this substream of E. Schema of IHTA is a set of atomic event queries as defined in the previous section. Only those events which match at least one of the atomic events of the schema are relevant for the respective IHTA. All other events are ignored by the IHTA.

Definition 4 (Schema of IHTA, Event Relevant for IHTA). Let *AtomicEventQueries* be the set of atomic event queries. Let $\mathcal{I}$ be IHTA. The *schema of* $\mathcal{I}$ is a set of atomic event queries, denoted $Schema(\mathcal{I}) \subseteq AtomicEventQueries$. The *default schema of* $\mathcal{I}$ is the set of atomic event queries appearing positively or negatively in at least one transition label of $\mathcal{I}$. Note that a schema of $\mathcal{I}$ always contains the default schema of $\mathcal{I}$. An event matching at least one atomic event query $q \in Schema(\mathcal{I})$ is relevant for $\mathcal{I}$. All other events are irrelevant for $\mathcal{I}$.

Example: The *Item Offer* IHTA described above have the following default schema: $Schema(\mathcal{I}_{ItemOffer})$ = { *bid(auctionID(A), itemID(I)), hammerBeat(auctionID(A), itemID(I)), sell(auctionID(A), itemID(I))* }.

Definition 5 (Atomic State, Non-atomic State, Child State, Descendant State, Root State, Parent State, Ancestor State, Substate, Superstate). A state $s \in S$ is

- *atomic*, if s does not contain IHTA, i.e. $children(s) = \emptyset$. s is depicted as a circle.
- *non-atomic*, if s contains IHTA itself, i.e. $children(s) \neq \emptyset$. s is depicted as a rectangle. All states $s' \in children(s)$ are *children* of s and s is the *parent* of all s', denoted $parent(s')$. *Descendants* of s are all states which are within s, formally: $descendants : S \to 2^S$

$$descendants(s) = \begin{cases} \emptyset & \text{if } s \text{ is atomic} \\ children(s) \cup \bigcup_{s' \in children(s)} descendants(s') & \text{otherwise} \end{cases}$$

Descendants of s are *substates* of s, children of s are *direct substates* of s.

A state $r \in S$ containing all other states is the *root state* of the IHTA, i.e. $r \cup descendants(r) = S$. Each IHTA has exactly one root state.

Ancestors of s' are all states s' appears within, formally:

$ancestors : S \to 2^S$

$$ancestors(s') = \begin{cases} \emptyset & \text{if } s' \text{ is the root state} \\ parent(s') \cup ancestors(parent(s')) & \text{otherwise} \end{cases}$$

Ancestors of s' are *superstates* of s', parent of s' is the *direct superstate* of s'. The root state has no parent. All other states have exactly one parent. The root state is an ancestor of all other states.

Each state of IHTA is either atomic or non-atomic. Each (atomic or non-atomic) state of IHTA is either start or end or neither start nor end state. An end state s is depicted as a double circle if s is atomic or as a double rectangle if s is non-atomic. Consider examples in Section 6.2.

Definition 6 (**Event Transition, Delay Transition**)**.** A transition $t : s \xrightarrow{q,c,f} s' \in T$ is

- *event* if $q \neq \emptyset$.
- *delay* if $q = \emptyset$.

Each transition is either event or delay. Consider examples in Section 6.2.

Definition 7 (**Instantiating Transition, Terminating Transition**)**.** A transition $t : s \xrightarrow{q,c,f} s' \in T$ is

- *instantiating* if t goes into a (set of nested) non-atomic state(s) and $s' \in Start$.
- *terminating* if t goes out of a (set of nested) non-atomic state(s) and $s \in End$.

All properties of transitions are independent from each other and can be freely combined in the sense that an event or a delay transition can be (a) neither instantiating nor terminating, (b) instantiating but not terminating, (c) terminating but not instantiating or (d) both instantiating and terminating. Consider examples in Section 6.2 and Figure 6.6.

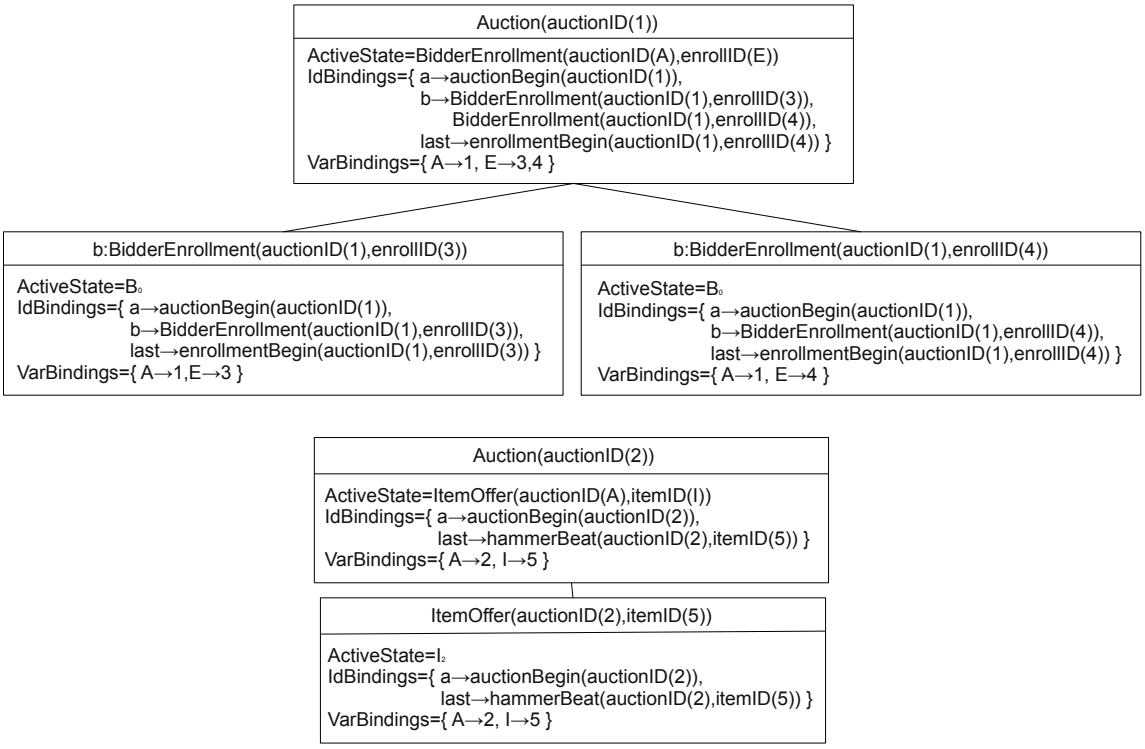

Figure 6.4: Graphical representation of two runs of IHTA in Figure 6.1. An instance is represented by a box. The tree of instances is called run.

6.3.2 Automaton Configuration

Having specified the basic notions IHTA are comprised of, we will now present the automaton configuration, a structure which represents the entire state during the runs of an automaton. It includes all sets of nondeterministic runs, all instances of non-atomic states in each run and, for each instance, its active state, variable bindings, and event and state identifier bindings.

Instance of a Non-Atomic State

Instantiation allows to represent an arbitrary number of concurrent processes. Consider Figure 6.4. *Auction(auctionID(1))* and *Auction(auctionID(2))* are instances of the state *Auction(auctionID(A))* (Definition 8). These instances represent two auctions taking place at the same time independently from each other. Both auctions follow the predefined workflow specified by the IHTA within the state *Auction(auctionID(A))*. The active states of the instances are *BidderEnrollment(auctionID(1),enrollID(E))* and *ItemOffer(auctionID(2),itemID(I))* respectively (Definition 10). Each of the instances has its own variable bindings and identifier bindings. Each of the concurrent processes can give rise to new concurrent processes. So for example, during the auction with iden-

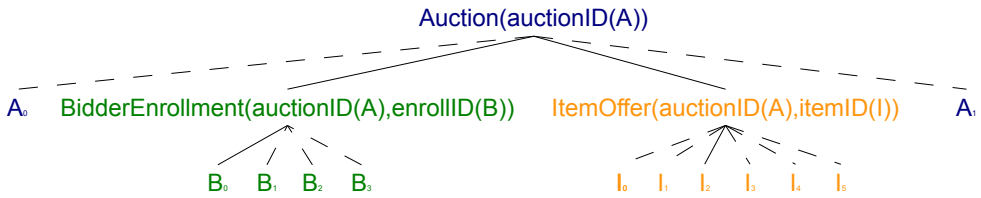

Figure 6.5: The run modeling approach of Hierarchical Timed Automata [41].

tifier 1 two bidder enrollment processes take place at the same time, i.e. the instance *Auction(auctionID(1))* has two subinstances instantiating the nonatomic state *BidderEnrollment(auctioID(A),enrollID(E))*. Instances of non-atomic states form a tree called run (Definition 14). Figure 6.4 shows two runs of the IHTA in Figure 6.1. A run cannot be modeled as a partial tree of the state tree because there can be more than one instances with the same active state, e.g. *BidderEnrollment(auctionID(1),enrollID(3)* and *BidderEnrollment(auctionID(1),enrollID(4)*.

Hierarchical Timed Automata (HTA) [41] do not have the notion of instantiation. A run of HTA is presented as a partial tree of the state tree. Figure 6.5 shows an example of this approach. Edges which are bold represent a tree of states which are currently active. The approach cannot represent a state being active more than once in a run. Therefore, only a fixed number of concurrent processes can be modeled. This does not meet our requirements of modeling an arbitrary number of concurrent processes. IHTA handle this by *instantiation*. A run is built top-down during run-time in the following way. When an enabled transition t (Definition 12) enters a non-atomic state n, an instance i of n is created. If n is the root state, i has no parent. Otherwise, the parent of i is the instance firing t. i inherits the variable bindings and the identifier bindings from its parent, but only those which were not yielded during creating the siblings of i.

Every instance has a set of local variable bindings and identifier bindings. These are updated when a transition fires in the instance. If a transition fires in an instance, the active state of the instance changes. This procedure is called *transformation*.

Sibling instances represent concurrent processes. Subinstances of an instance can *terminate*.

These three steps, i.e. instantiation, transformation, and termination, are executed every time a transition fires (Definition 18).

Now, let us formally define the notions which were informally explained above. Let $\mathcal{I}$ be IHTA. Let $IdentifierBindings$ be the set of event (or state) identifier bindings and $VariableBindings$ be the set of variable bindings.

Definition 8 (Instance of a Non-Atomic State). An instance i of a non-atomic state

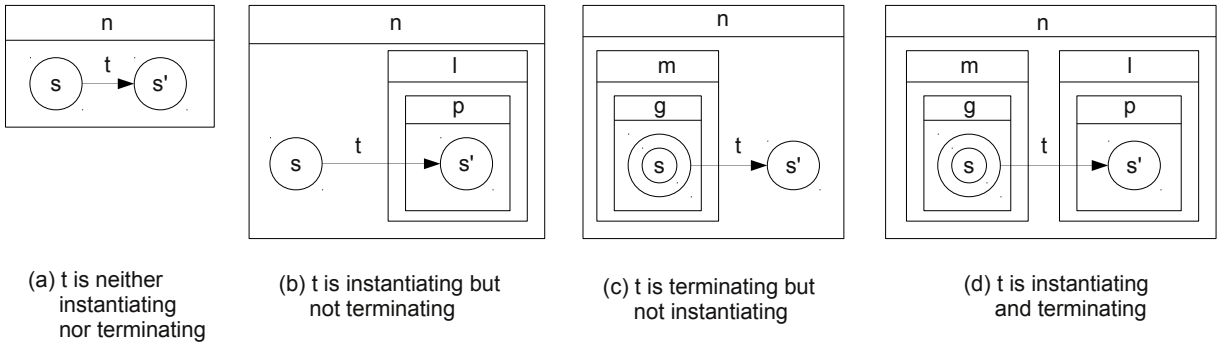

Figure 6.6: Four kinds of transitions illustrated by the (incomplete) IHTA

$n \in S$ of $\mathcal{I}$ is a tuple $(n \cdot VarBindings, ActiveState, IdBindings, VarBindings)$ where

- $n \cdot VarBindings$ is the name of i which is the result of the application of $VarBindings$ (as defined below) to n.
- $ActiveState \in children(n)$ is the active state of i (Definition 10).
- $IdBindings \in IdentifierBindings$ is the set of identifier bindings of i.
- $VarBindings \in VariableBindings$ is the set of variable bindings of i.

Definition 9 (**Schema of an Instance, Event Relevant for an Instance**). Let $n \in S$ be a nonatomic state of IHTA with the schema $Schema(n)$. Let i be an instance of n with the variable bindings $VarBindings$. The schema of i is the schema of n in which the variable bindings of i are propagated, formally $Schema(i) = \{q \cdot VarBindings \mid q \in Schema(n)\}$. An event is relevant for i if it matches at least one atomic event query of the schema of i. Otherwise, an event is irrelevant for i.

Example: If the schema of the IHTA *Item Offer* $\mathcal{I}_{ItemOffer}$ in Figure 6.1 is *Schema($\mathcal{I}_{ItemOffer}$) = { bid(auctionID(A), itemID(I)), hammerBeat(auctionID(A), itemID(I)), sell(auctionID(A), itemID(I)) }* then the schema of its instance $i_{ItemOffer}$ in Figure 6.4 is *Schema($i_{ItemOffer}$) = { bid(auctionID(2), itemID(5)), hammerBeat(auctionID(2), itemID(5)), sell(auctionID(2), itemID(5)) }*.

Consider Figure 6.6.

Definition 10 (**Active State of an Instance**). Let i be an instance of a non-atomic state $n \in S$. If $t : s \xrightarrow{q,c,f} s' \in T$ is an enabled transition in i (Definition 12) then the active state of i, denoted $activeState(i)$, is defined as follows:

$$activeState(i) = \begin{cases} l \mid l \in children(n) \wedge l \in ancestors(s') & \text{if } t \text{ is instantiating} \\ s' & \text{else} \end{cases}$$

Every time an enter transition of IHTA $\mathcal{I}$ fires a new run of $\mathcal{I}$ is created. An enter transition $t : \emptyset \xrightarrow{q,c} s' \in T$ of $\mathcal{I}$ is performed if the following condition is satisfied: If the event query q of t is positive, there must be an event in the stream which is matched by q such that the optional temporal constraints c of t are satisfied. If q is negative, i.e. $q = \neg q'$, q matches the event stream if the stream does not contain an event matched by q' such that the non-optional temporal constraints c of t are satisfied.

Definition 11 (**Enabled Enter Transition**)**.** Let E be an event stream. Let $\mathcal{I}$ be IHTA. An enter transition $t : \emptyset \xrightarrow{q,c} s' \in T$ of $\mathcal{I}$ is enabled if:

- if $c = \emptyset$ then q is positive and $\exists a^d \in E, \exists \sigma$ such that $q\sigma = a$.
- if $c \neq \emptyset$ then let $d' \in \mathbb{TI}$ be the time interval during which c is satisfied
  - if q is positive then $\exists a^d \in E, \exists \sigma$ such that $q\sigma = a$ and $d \sqsubseteq d'$[2] .
  - if q is negative, i.e. $q = \neg q'$, then $\nexists a^d \in E$ for which $\exists \sigma$ such that $q\sigma = a$ and $d \sqsubseteq d'$.

An instance i of a non-atomic state $n \in S$ executes a transition $t : s \xrightarrow{q,c,f} s' \in T$ if the following conditions are satisfied:

- The instance i is in the right state which depends on the kind of the transition t. If t is:
  - neither instantiating nor terminating (Case (a) in Figure 6.6), $s, s' \in children(n)$ and s is the active state of i.
  - instantiating but not terminating (Case (b) in Figure 6.6), $s \in children(n), s' \in descendants(n)$ and s is the active state of i.
  - terminating but not instantiating (Case (c) in Figure 6.6), $s \in descendants(n), s' \in children(n)$ and m is the active state of i where $m \in children(n), m \in ancestors(s)$.
  - instantiating and terminating (Case (d) in Figure 6.6), $s, s' \in descendants(n)$ and m is the active state of i where $m \in children(n), m \in ancestors(s)$ and there is no state k such that $s, s' \in descendants(k), k \in descendants(n)$ (otherwise an instance of k performs t).

[2] $d \sqsubseteq d'$ is the shortcut for $b(d') \leq b(d)$ and $e(d) \leq e(d')$. In other words the time interval d' comprises the time interval d.

- The components of the label of t, namely the event query q, the temporal constraints c, and the termination constraints f are satisfied.
  - If t is an event transition and the event query q of t is positive, there must be an event in the stream which is matched by q such that the optional temporal constraints c and the optional termination constraints f of t are satisfied with respect to the variable bindings and identifier bindings of i.

    If q is negative, i.e. $q = \neg q'$, q matches the event stream if the stream does not contain an event matched by q' such that the temporal constraints c and the termination constraints f of t are satisfied with respect to the variable bindings and identifier bindings of i. Please note that in this case either temporal or termination constraints are non-optional.
  - The temporal constraints c of t are satisfied with respect to the identifier bindings of i.
  - If t is terminating, the termination constraints f of t are satisfied with respect to the identifier bindings of i.

Definition 12 (Transition Enabled in an Instance). Let i be an instance of a non-atomic state $n \in S$, let $VarBindings$ be the variable bindings of i and $IdBindings$ be the identifier bindings of i. Let E be an event stream. A transition $t : s \xrightarrow{q,c,f} s' \in T$ is enabled in i if:

1. $activeState(i) = \begin{cases} m \mid m \in children(n) \wedge m \in ancestors(s) \wedge \\ \quad \nexists k \mid s, s' \in descendants(k) \wedge \\ \quad\quad k \in descendants(n) & \text{if } t \text{ is terminating} \\ s & \text{else} \end{cases}$
2. if t is an event transition then
   - if $c = \emptyset$ and $f = \emptyset$ then q is positive and $\exists a^d \in E, \exists \sigma$ such that $q \cdot VarBindings \cdot \sigma = a$.
   - if $c \neq \emptyset$ or $f \neq \emptyset$ then let $d' \in \mathbb{TI}$ be the time interval during which $c \cdot IdBindings$ and $f \cdot IdBindings$ are satisfied
     - if q is positive then $\exists a^d \in E, \exists \sigma$ such that $q \cdot VarBindings \cdot \sigma = a$ and $d \sqsubseteq d'$.
     - if q is negative, i.e. $q = \neg q'$, then $\nexists a^d \in E$ for which $\exists \sigma$ such that $q' \cdot VarBindings \cdot \sigma = a$ and $d \sqsubseteq d'$.
3. if $c \neq \emptyset$ then $c \cdot IdBindings$ is satisfied.

4. if t is terminating then $f \cdot IdBindings$ is satisfied.

Just analogously to Definition 5 for states, the following sets are defined for an instance:

Definition 13 (**Child Instance, Descendant Instance, Parent Instance, Ancestor Instance, Subinstance, Superinstance**). Let $n, n' \in S$ such that $n' \in children(n)$. Let i be an instance of n and i' be an instance of n' such that i' was created by an instantiating transition in i or both i and i' were created by an instantiating transition in an ancestor instance of i. Then i' is a *child* of i, denoted $i' \in children(i)$, and i is *parent* of i', denoted $parent(i') = i$. Let I be the set of instances. *Descendants* of i are the following:

$$descendants : I \to 2^I$$

$$descendants(i) = \begin{cases} \emptyset & \text{if the active state of } i \text{ is atomic} \\ children(i) \cup \bigcup_{i' \in children(i)} descendants(i') & \text{otherwise} \end{cases}$$

Descendants of i are *subinstances* of i, children of i are *direct subinstances* of i.

Ancestors of i' are the following:

$$ancestors : I \to 2^I$$

$$ancestors(i') = \begin{cases} \emptyset & \text{if } i' \text{ is an instance of the root state} \\ parent(i') \cup ancestors(parent(i')) & \text{otherwise} \end{cases}$$

Ancestors of i' are *superinstances* of i', parent of i' is the *direct superinstance* of i'. An instance of the root state has no parent. All other instances have exactly one parent. An instance of the root state is an ancestor of all other instances.

Run

A run r saves all current instances in r as well as their name, active state, identifier bindings and variable bindings.

Definition 14 (**Run**). Let $\mathcal{I}$ be IHTA. Let I be the set of instances and $Names$ be the set of instance names. A run r of $\mathcal{I}$ is a tuple ($children$, $name$, $activeState$, $identifierBindings$, $variableBindings$) where

- $children : I \to 2^I$ maps an instance $i \in I$ to its children. $children(i)$ gives rise to a tree of instances the root of which is an instance of the root state.
- $name : I \to Names$ maps an instance $i \in I$ to its name which is the result of the application of variable bindings of i to the state $n \in S$ instantiated by i, i.e. $name(i) = n \cdot variableBindings(i)$.
- $activeState : I \to S$ maps an instance to its active state.

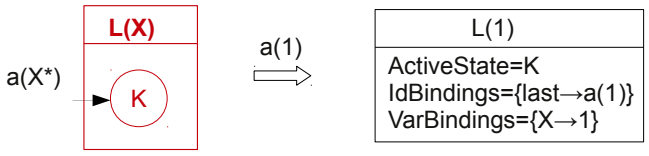

Figure 6.7: The (incomplete) IHTA and its initial run. The enter transition instantiates only the root state.

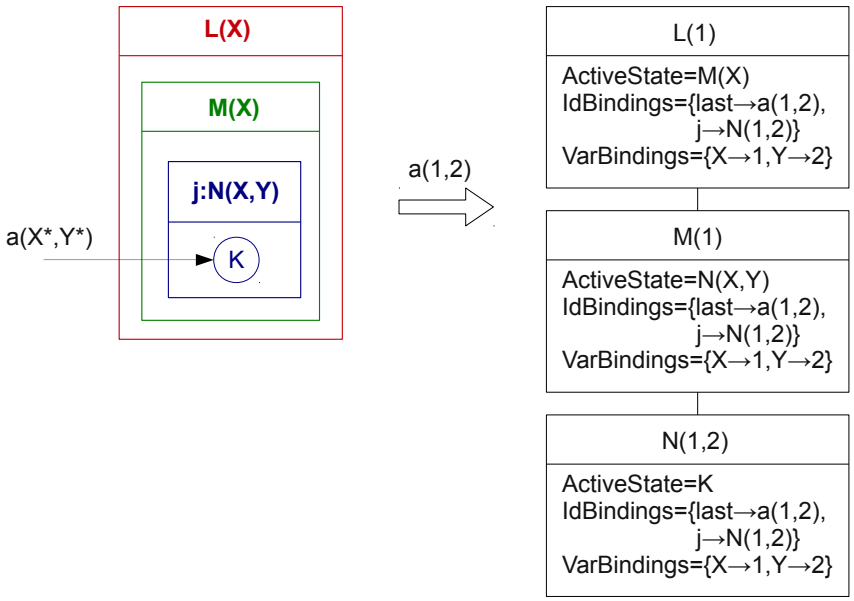

Figure 6.8: The (incomplete) IHTA and its initial run. The enter transition instantiates the root state and its two substates.

- $identifierBindings : I \rightarrow IdentifierBindings$ maps an instance to its identifier bindings.
- $variableBindings : I \rightarrow VariableBindings$ maps an instance to its variable bindings.

Definition 15 (Event Relevant for a Run). Let r be a run of IHTA and I_r be the set of instances in r. An event is relevant for r if it is relevant for at least one instance in I_r. Otherwise, an event is not relevant for r.

Definition 16 (Initial Run). Let $\mathcal{I}$ be IHTA. Let $t : \emptyset \xrightarrow{q,c} s' \in T$ be an enabled enter transition of $\mathcal{I}$. Let $root \in S$ be the root state of $\mathcal{I}$. Initially there is one run r. It contains an instance i of $root$ without subinstances if t instantiates only $root$. If t instantiates $root$ and substates l of $root$, r contains an instance i of $root$ and an instance i' of each l which

are subinstances of i. $r := Initialize_t()$ which is defined as follows:

$$
\begin{aligned}
& i \text{ is a new instance of } root \\
children &:= i \mapsto \emptyset \\
activeState &:= \begin{cases} i \mapsto s' & \text{if } t \text{ instantiates only } root \\ i \mapsto l \mid l \in children(root) \wedge & \text{if } t \text{ instantiates } root \text{ and} \\ \qquad l \in ancestors(s') & \text{substates of } root \end{cases} \\
identifierBindings &:= i \mapsto \{last \mapsto e \cup id \mapsto e\} \\
variableBindings &:= i \mapsto \sigma \\
name &:= i \mapsto root \cdot \sigma \\
\text{return } &Inst(i, s', children, name, activeState, identifierBindings, variableBindings)
\end{aligned}
$$

where
id is the event identifier, i.e. $q = id : q'$,
e is the event matched by q, and
σ is the unifier of q and e, i.e. $q\sigma = e$.
In this case we say that **t is enabled by e**.
The function *Inst(i, s', children, name, activeState, identifierBindings, variableBindings)* is given in Definition 18.

Consider the (incomplete) IHTA and their respective initial runs in Figure 6.7 and Figure 6.8.

Run Modification

Let r be a run of IHTA $\mathcal{I}$ and I_r be the set of instances of r. Let $t : s \xrightarrow{q,c,f} s' \in T$ be a transition enabled in an instance $i \in I_r$ of a nonatomic state $n \in S$. When t fires in i, r is modified as follows:

1. Transformation: If s' is a child of n then s' becomes the active state of i. Otherwise, the active state of i is the ancestor of s' which is a child of n. (Compare Definition 10.) If t is an event transition, the variable bindings and identifier bindings of i are updated. The variable bindings of i are updated using the unifier of the event query q of t and the current event e. If q is labeled by an event identifier id, the identifier bindings of i are updated such that id is mapped to e. The identifier bindings of all ancestors of i are also updated: The event identifier *last* references e.

Example: Consider Figure 6.9. Assume the event $a(1, 2)$ arrives. The transition $t : G \xrightarrow{a(X^*,Y^*)} K$ is enabled in the instance i of L (compare Definition 12). $K \notin children(L)$

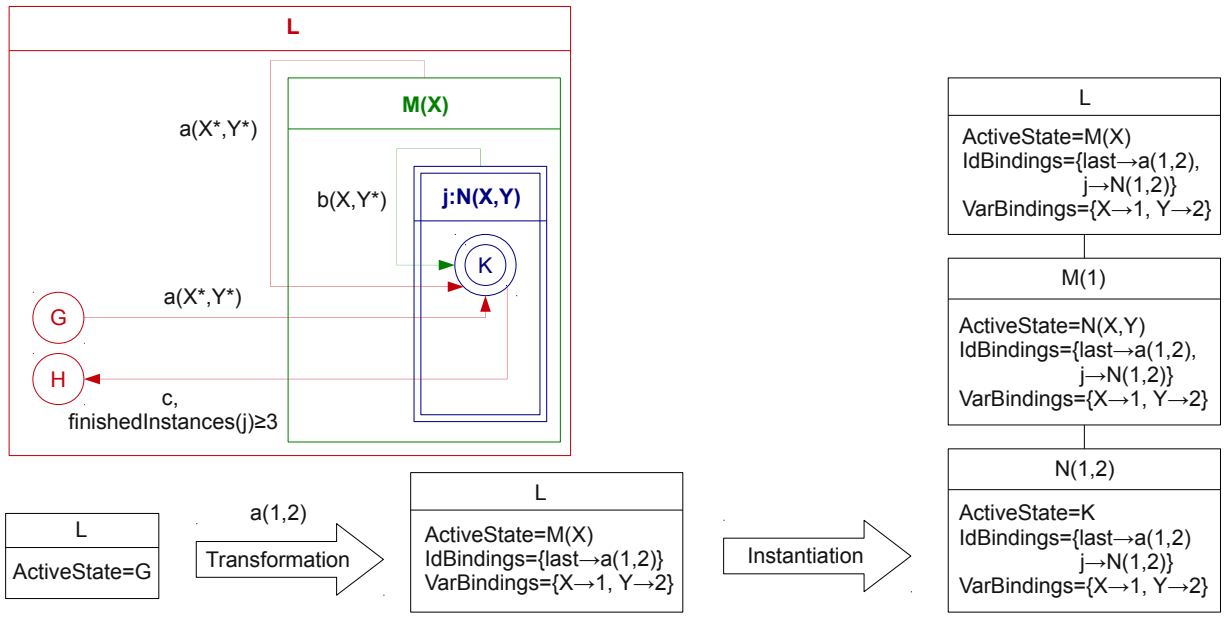

Figure 6.9: The (incomplete) IHTA and its run modification involving transformation and instantiation.

and the active state of i is $M(X)$ because $M(X) \in \mathit{ancestors}(K)$ and $M(X) \in \mathit{children}(L)$. The variable bindings and the identifier bindings of i are updated as shown in Figure 6.9.

2. Instantiation: If t is instantiating, then for each state t goes into (more exactly, for each ancestor of s' which is a descendant of n) a new instance i' is created. All these instances are in a parent-child relation as defined in Definition 13. Each new instance i' inherits variable bindings and identifier bindings from its parent but only those which were not yielded while creating siblings of i'. The active state of each new instance i' is either s' or the state which is instantiated by the children of i'. If a state k instantiated by t carries a state identifier id' the identifier bindings of the instance i' of k are updated so that id' is mapped to the name of i'. This state identifier binding is provided to all ancestors of i'.

Example: Consider Figure 6.9 again. The enabled transition $t : G \xrightarrow{a(X^*,Y^*)} K$ in i is instantiating. An instance for each state $M(X)$ and $N(X,Y)$ is created because $M(X), N(X,Y) \in \mathit{ancestors}(K)$ and $M(X), N(X,Y) \in \mathit{descendants}(L)$. $M(1)$ is the instance of $M(X)$, $M(1) \in \mathit{children}(i)$, and $N(1,2)$ is the instance of $N(X,Y)$, $N(1,2) \in \mathit{children}(M(1))$. The active state of $M(1)$ is $N(X,Y)$ and the active state of $N(1,2)$ is K. Each instance inherits the variable bindings and the identifier bindings from its parent instance.

Modeling Concurrent Processes

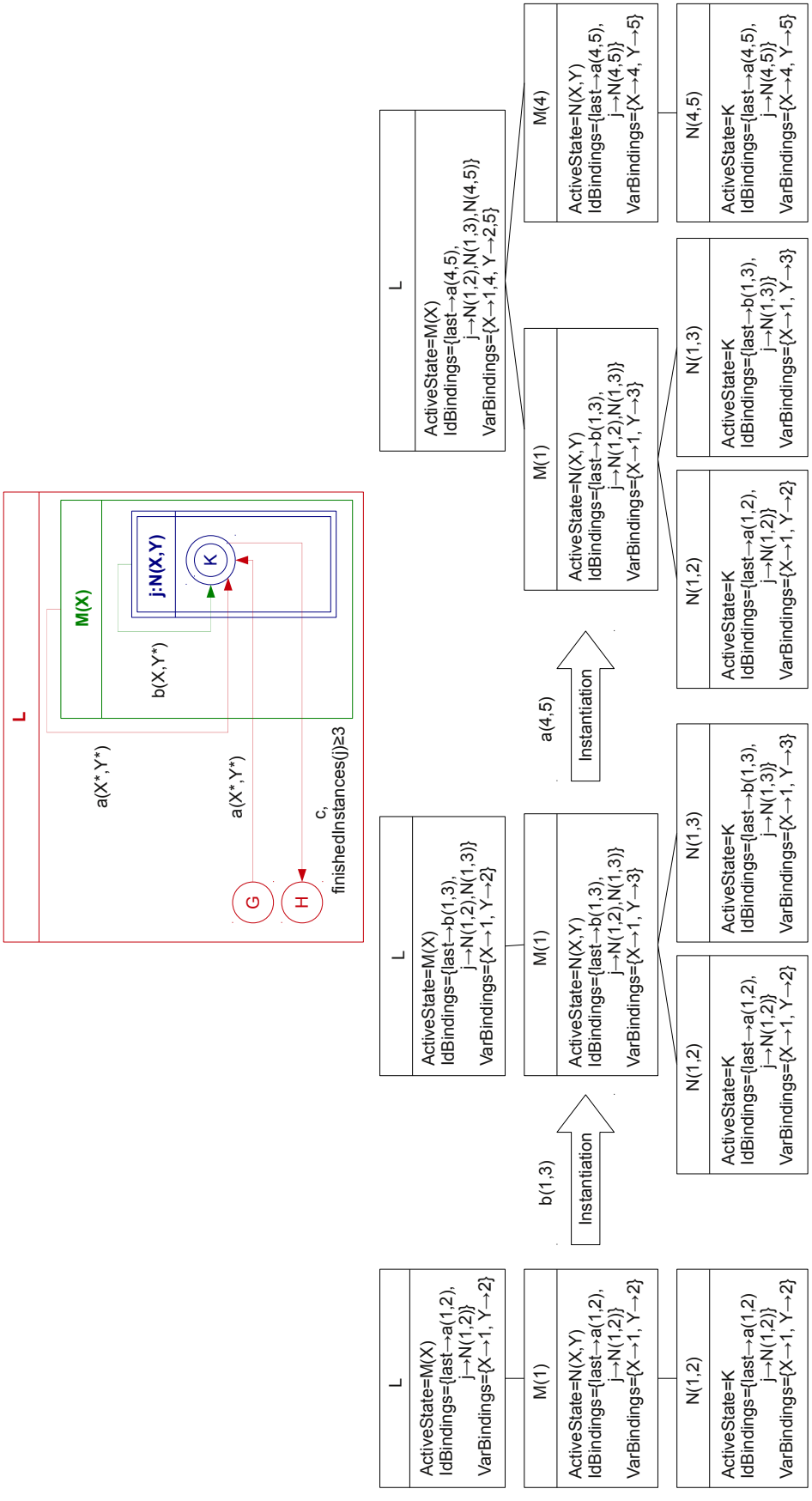

Figure 6.10: Modeling concurrent processes

A special form of instantiating transitions are transitions with a non-atomic source state s and a start target state s' where s' is a descendant of s. Figure 6.10 shows the result of the execution of two instantiating transitions of this form. In contrast to the instantiation shown in Figure 6.9, newly created subinstances are siblings of already existing subinstances. Since each of these subinstances processes transitions independently from other subinstances this models multiple concurrent processes.

To relate an event to an instance, event queries refer to the variable bindings of an instance. Variables used for this relation are usually declared in the event queries of instantiating transitions (see Section 6.2). This requires special care from the user in the following two respects: (1) Variables used for *uniquely* relating events to instances must be keys. (2) It is not prevented that variables are accidentally shadowed during the lifetime of an instance. For example, the variable Y in the instance $M(1)$ is shadowed after the first instantiation in Figure 6.10. But its subinstance $N(1, 2)$ still has the old binding of Y.

3. Termination: If t is terminating, all finished subinstances i' of the instance i with respect to the end state s are terminated. As defined in Definition 17, an instance i' is finished with respect to the end state s if either:

- the active state of i' is s or
- the active state of i' is a non-atomic state n (i.e. i' has subinstances i'' instantiating n) and all instances i'' are finished with respect to s.

Example: Consider the resulting run in Figure 6.10. All instances of the non-atomic state $N(X, Y)$ are finished with respect to the end state K according to the first condition above. Both instances of the non-atomic state $M(X)$ are finished with respect to the end state K according to the second condition above.

Definition 17 (Finished Instance with respect to an End State). Let i be an instance of a non-atomic state $n \in S$, let $s \in \mathit{descendants}(n)$ be an atomic end state. i is called finished with respect s if either

- $\mathit{activeState}(i) = s$
  or
- $\mathit{activeState}(i)$ is non-atomic and all instances of $\mathit{activeState}(i)$ which are subinstances of i are finished with respect to s.

Termination Constraints

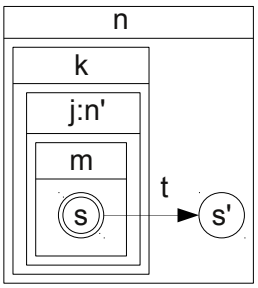

Figure 6.11: State identifier j in termination constraints of t illustrated by the (incomplete) IHTA

In the auction use case, for an item to be presented in an auction there must be at least two bidder enrollments. Therefore, the event transition between states B_3 and I_0 in Figure 6.1 is only executed if at least two instances of the non-atomic state *b:BidderEnrollment(auctionID(A),enrollID(E))* are in the state B_3. Such conditions on the number of instances are expressed by termination constraints of terminating transitions. Termination constraints are formulas of arithmetic expressions comparing the value returned by the functions $allInstances$ or $finishedInstances$ (defined below) with a natural number by means of the binary operators $=, \neq, <, \leq, >, \geq$.

Let $t : s \xrightarrow{q,c,f} s' \in T$ be a terminating transition in an instance i of a non-atomic state $n \in S$ in a run r. Let $j : n' \in S$ be a non-atomic state referenced by the state identifier j such that $j : n' \in descendants(n)$ and $j : n' \in ancestors(s)$. Consider Figure 6.11. Two functions $allInstances$ and $finishedInstances$ can be used in the termination constraint f of t. They are defined as follows:

- The function $allInstances : Identifiers \rightarrow \mathbb{N}_0$ maps j to the number of instances of n' regardless their active states in r.
- The function $finishedInstances : Identifiers \rightarrow \mathbb{N}_0$ maps j to the number of finished instances of n' with respect to s in r.

If t does not have an explicit termination constraint then it carries a **default termination constraint** $allInstances(l) = finishedInstances(l)$ where l references all states x for which $x \in descendants(n)$ and $x \in ancestors(s)$ holds (these are k, n' and m in Figure 6.11). Default termination constraint expresses that all instances of all substates of n in r must be finished with respect to s. If a terminating transition is labeled by an explicit termination constraint f, the default termination constraint is replaced by f.

Example: The termination constraint of the transition between states B_3 and I_0 in Figure 6.1 is $finishedInstances(b) \geq 2$. Therefore the default termination constraint

$allInstances(b) = finishedInstances(b)$ is replaced by the custom termination constraint, and the transition can fire even if not all instances of the non-atomic state *b:BidderEnrollment(auctionID(A),enrollID(E))* are finished with respect to B_3 but at least two.

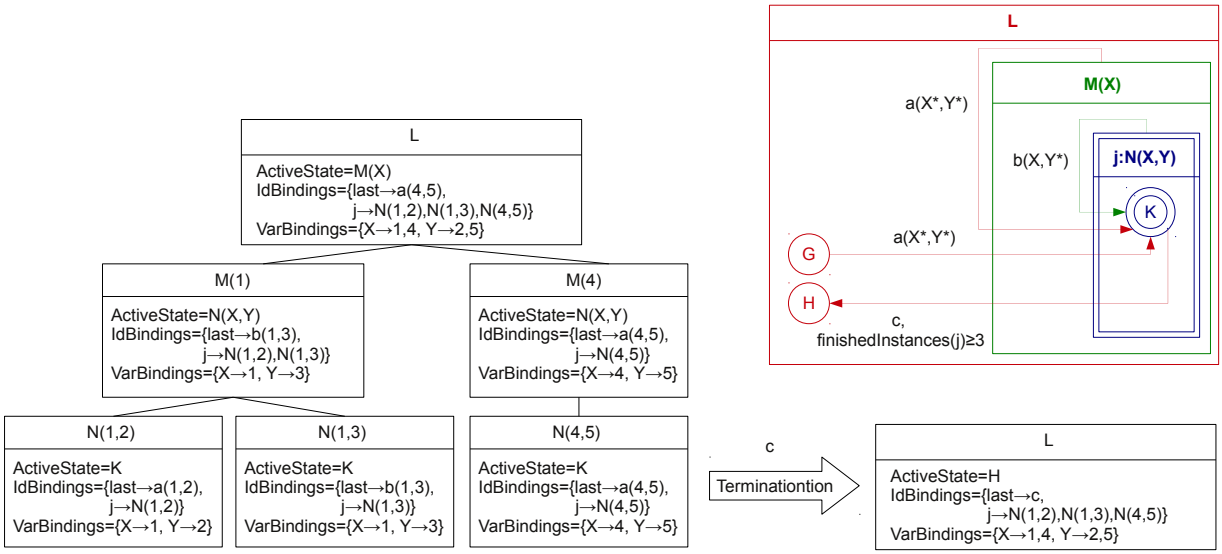

Figure 6.12: The (incomplete) IHTA and its run modification involving termination

Example: Assume the event c arrives. Then the terminating transition between states K and H in Figure 6.12 is enabled, compare Definition 12. (If j referenced the non-atomic state $M(X)$ the termination constraint would not be satisfied and the transition would not be enabled.) All subinstances of the instance i of L which are finished with respect to K terminate. The active state of i is now H.

Definition 18 formally specifies the run modification including tree steps namely termination, transformation and instantiation which were informally described and illustrated by examples above.

Definition 18 (Deterministic Run Modification). Let $r = (children, name, activeState, identifierBindings, variableBindings)$ be a run of IHTA. Let I_r be the set of instances in r and $i \in I_r$ instantiating $n \in S$. Let $t : s \xrightarrow{q,c,f} s' \in T$ be the only enabled transition in i. Performing t in i is executed by the deterministic modification function

$$\mathcal{D}_{i,t}(children, name, activeState, identifierBindings, variableBindings) \stackrel{def}{=} (children_3, name_3, activeState_3, identifierBindings_3, variableBindings_3)$$

which takes r as an input and returns a modified run ($children_3$, $name_3$, $activeState_3$, $identifierBindings_3$, $variableBindings_3$) as an output. The execution of $\mathcal{D}_{i,t}$ involves

three steps:

1. **Termination**
   If t is terminating the subinstances of i are terminated. For this effect, an intermediate run ($children_1$, $name_1$, $activeState_1$, $identifierBindings_1$, $variableBindings_1$) is created:

$$children_1 := \begin{cases} children[i \mapsto \emptyset] & \text{if } t \text{ is terminating} \\ children & \text{else} \end{cases}$$
$$name_1 := name$$
$$activeState_1 := activeState$$
$$identifierBindings_1 := identifierBindings$$
$$variableBindings_1 := variableBindings$$

2. **Transformation**
   The active state of i is updated. If t is an event transitions, the event identifiers and variable bindings of i are updated. An intermediate run ($children_2$, $name_2$,

$activeState_2$, $identifierBindings_2$, $variableBindings_2$) is created:

$$children_2 := children_1$$

$$name_2 := name_1$$

$$activeState_2 := \begin{cases} activeState_1[i \mapsto l \mid l \in children(n) \wedge \\ \quad l \in ancestors(s')] & \text{if } t \text{ is instantiating} \\ activeState_1[i \mapsto s'] & \text{else} \end{cases}$$

$$identifierBindings_2 := \begin{cases} identifierBindings_1[i \mapsto \\ \quad \{identifierBindings_1(i) \cup \\ \quad last \mapsto e \ \cup \ id \mapsto e\}, \\ \quad \forall a \in ancestors(i) : a \mapsto \\ \quad \{identifierBindings_1(a) \cup \\ \quad last \mapsto e\}] & \text{if } t \text{ is an event transition} \\ identifierBindings_1 & \text{else} \end{cases}$$

$$variableBindings_2 := \begin{cases} variableBindings_1[i \mapsto \\ \quad \{variableBindings_1(i) \cup \sigma\}] & \text{if } t \text{ is an event transition} \\ variableBindings_1 & \text{else} \end{cases}$$

where
id is the event identifier, i.e. $q = id : q'$,
e is the event matched by $q \cdot variableBindings_1(i)$, and
σ is the unifier of $q \cdot variableBindings_1(i)$ and e, i.e. $q \cdot variableBindings_1(i) \cdot \sigma = e$.
In this case we say that **t is enabled by e**.

3. **Instantiation**
   If t is instantiating, then for each state t goes into a new instance is created. The resulting run is ($children_3$, $name_3$, $activeState_3$, $identifierBindings_3$, $variableBindings_3$) := $Inst(i, s', children_2, name_2, activeState_2, identifierBindings_2, variableBindings_2)$ which is defined as follows:

$$Inst(i, s', children, name, activeState, identifierBindings, variableBindings) \stackrel{\text{def}}{=}$$

$$\begin{cases} Inst(i', s', children', name', activeState', \\ \quad identifierBindings', variableBindings'), \text{ where} \\ \quad i' \text{ is a new instance of } activeState(i) \\ \quad children' := children[i \mapsto \{children(i) \cup i'\}] \\ \quad activeState' := activeState[i' \mapsto l \mid \\ \quad\quad l \in children(activeState(i)) \wedge \\ \quad\quad (l \in ancestors(s') \vee l = s')] \\ \quad variableBindings' := variableBindings[i' \mapsto \\ \quad\quad \{variableBindings(i) - varBindings(i)\}] \\ \quad name' := name[i' \mapsto activeState(i) \cdot variableBindings'(i')] \\ \quad identifierBindings' := identifierBindings[i' \mapsto \\ \quad\quad \{(identifierBindings(i) \cup id' \mapsto name') \\ \quad\quad -idBindings(i)\}, \\ \quad\quad \forall a \in ancestors(i) : a \mapsto \\ \quad\quad \{identifierBindings(a) \cup id' \mapsto name'\}] & \text{if } activeState(i) \neq s' \\ \\ (children, name, activeState, identifierBindings, \\ \quad\quad variableBindings) & \text{else} \end{cases}$$

where
id' is the state identifier of the active state of i instantiated by i',
$idBindings(i)$ and $varBindings(i)$ be the identifier bindings and the variable bindings of i yielded while creating siblings of i'. These bindings are irrelevant for i'.

Set of Nondeterministic Runs

Definition 18 specifies run modification if there is only one enabled transition in an instance, i.e. deterministic run modification. However, IHTA are non-deterministic in general, i.e. there can be multiple transitions enabled in an instance at the same time. It is not known beforehand which of these transitions leads to the end state. Therefore, IHTA support don't know (or disjunctive) nondeterminism (consider Section 2.3.5). Our understanding of nondeterminism is based on Nondeterministic Finite Automata NFA [108].

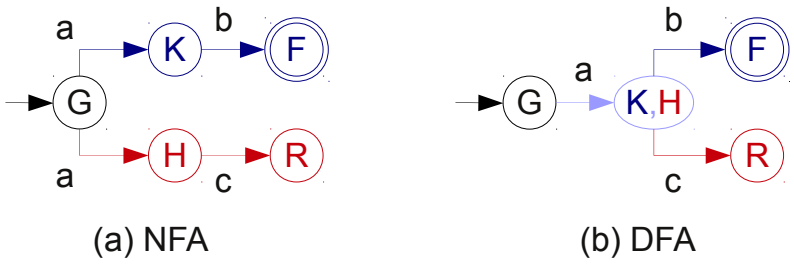

Figure 6.13: Powerset construction

When multiple transitions of an NFA can be executed, *any* of them is chosen. As long as there is *at least one* run of the NFA which leads to an end state, the input word is accepted.

Powerset construction method [108], i.e. translation of NFA into Deterministic Finite Automata (DFA), is a solution to the ambiguous choice between multiple enabled transitions. As motivated in Section 2.3.5, powerset construction allows event stream verification on-the-fly, i.e. there is no need to save past events in order to test other choices.

Example: Figure 6.13 shows an exemplary NFA and its respective DFA without the states which are unreachable from the start state. Assume the active state is G and the event a arrives. Two transitions of the NFA are enabled, the one between states G and K and the one between states G and H. Both runs are followed. They are depicted in blue and red in Figure 6.13. The active state of the blue run is K and the active state of the red run is H. This behavior is simulated by merging the states K and H in the DFA. Assume the event b arrives. The red run of the NFA cannot accept it and is therefore dropped. The blue run accepts it and is therefore stil valid. (If the blue run did not accept b, the word would not be in the language specified by the NFA.) The active state is F and the word is accepted. Note that the same behavior is simulated by the DFA.

As the example shows, powerset construction method relies on the fact that the automaton is finite, i.e. has a finite number of states. There is no way of using it for IHTA because the number of instances of states is unbounded in each run. However, powerset construction method can be reproduced by keeping track of all possible nondeterministic runs of IHTA (which are called *a set of nondeterministic runs* in the following). When an instance in a run r is in a state with k enabled transitions, r is branched into k runs $r_1, \ldots, r_k$. Each of $r_1, \ldots, r_k$ performs one enabled transition. $r_1, \ldots, r_k$ belong to the same set of nondeterministic runs as r. All sets of nondeterministic runs are accumulated by *automaton configuration* and modified dynamically. When an event e occurs, runs of the respective set of nondeterministic runs which do not have an instance with event transitions which are triggered by e are deleted. If a set of nondeterministic runs becomes

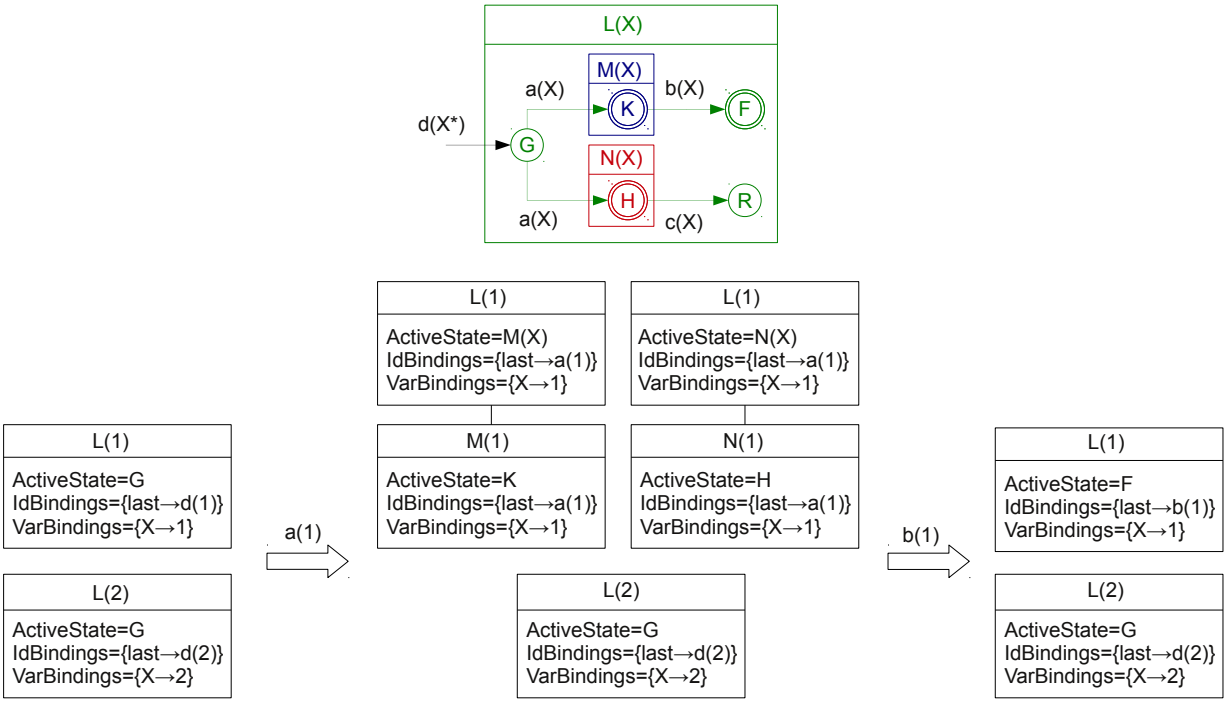

Figure 6.14: Set of nondeterministic runs

empty, i.e. no run was able to accept an event, the stream violates the IHTA. (Listing 6.1 is devoted to the stream verification algorithm which is based on this idea.) The reason of run branching (instead of state merging) is that each of nondeterministic runs can create new instances.

Example: Consider Figure 6.14. Initially there are two runs of the IHTA. Assume $a(1)$ arrives. It is relevant for the first run. The run is branched into two nondeterministic runs. The (same!) instance $L(1)$ has different active states and different subinstances in different runs. If at least one of the nondeterministic runs is successful, the stream does not violate the IHTA.

The second difference between classical nondeterminism [108] and nondeterminism of IHTA is that the elements of an input word (i.e. events of a stream) are not necessarily ordered, i.e. they can have the same occurrence time since this is an important feature in many practical applications. The number of events with the same occurrence time must be finite, otherwise there would be no time progress. This is no restriction of practical applications because the number of events with the same occurrence time is unbounded. The order in which such events are matched by transitions of an instance is ambiguous, and a run evolves differently depending on a particular order.

IHTA handle this by keeping track of all possible choices of event order. For a set of n events with equal occurrence time there are $n!$ (the factorial of n) possible permu-

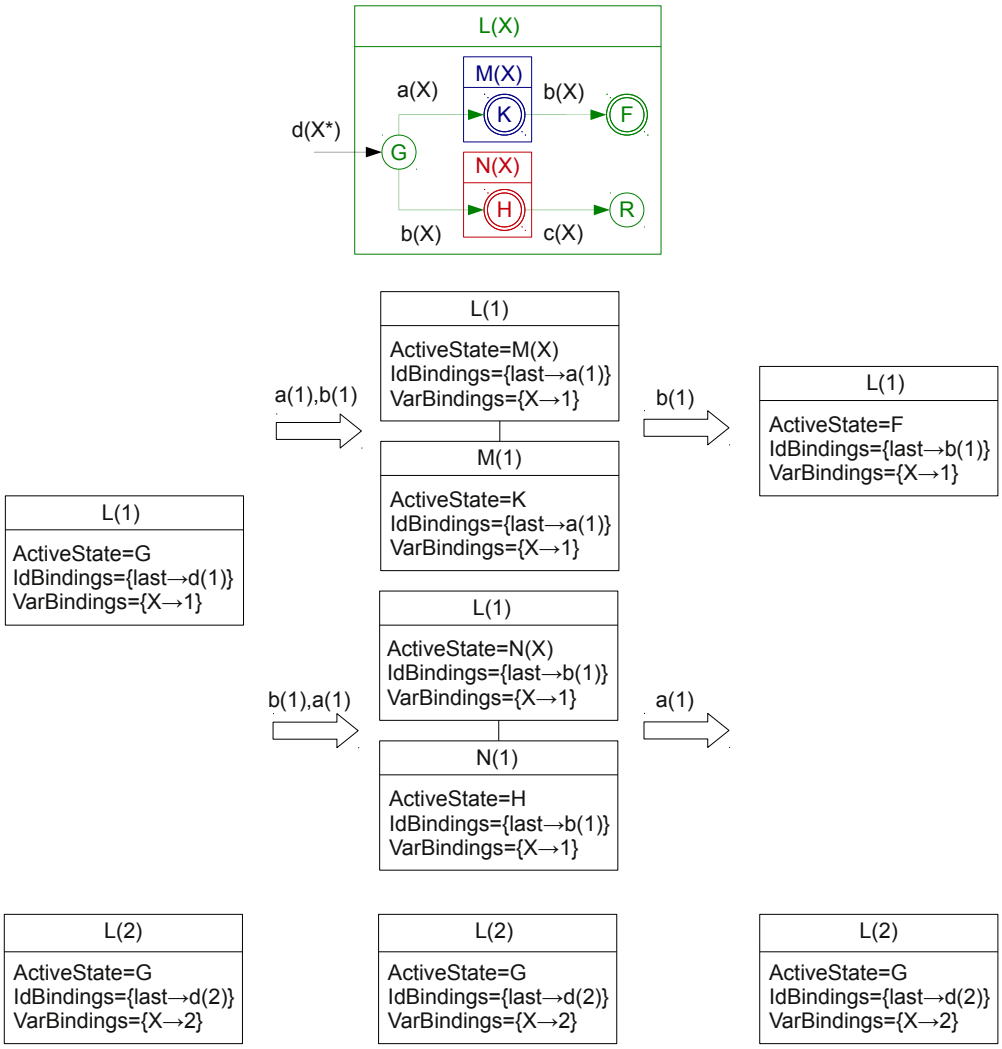

Figure 6.15: Set of nondeterministic runs

tations. Therefore, when such a set is detected, every existing run for which the set is relevant is branched into $n!$ runs. Each of these runs separately handles one possible event permutation and is dropped when it cannot match an event.

Example: Consider Figure 6.15. Initially there are two runs of the IHTA. Assume two events $a(1)$ and $b(1)$ arrive at the same time. They are relevant for the first run. The run is split into two nondeterministic runs. The (same!) instance $L(1)$ has different active states and different subinstances in different runs. The first one of the new runs works on $a(1), b(1)$, the second on $b(1), a(1)$. The second run is not able to accept $a(1)$ and is deleted. The first run accepts both events.

All possible permutations of events with the same end time point of their occurrence times must be considered if these events are relevant for the same run (not only for the same instance!). The order of execution of events which are relevant for different runs

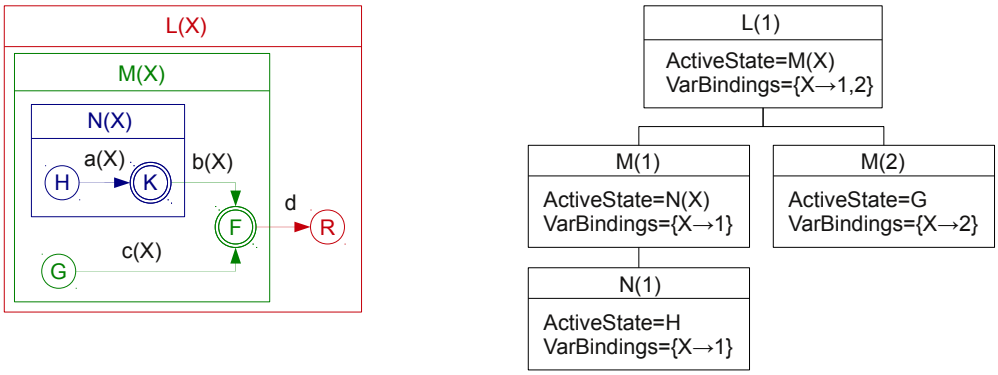

Figure 6.16: The (incomplete) IHTA and its run.

does not matter.

Example: Consider Figure 6.16. Assume events $a(1), b(1), c(2), d$ arrive at the same time. They are relevant for the same run but for four different instances in the run. These events must be processed in the above or alternatively in the following orders: $a(1), c(2), b(1), d$ or $c(2), a(1), b(1), d$. The evaluations of these permutations return the same run. A set of nondeterministic runs (Definition 19) saves only different runs. All other permutations of the events cannot be accepted. If only one permutation p of the events were considered and if p were not the one of the above permutations, the stream verification algorithm (Listing Listing 6.1) would stop the evaluation by returning false.

Definition 19 (Set of Nondeterministic Runs). Let r be a run of IHTA. Let I_r be the set of instances in r. Let $t_1, \ldots, t_k \in T$ be the set of all transitions enabled in any instance $i \in I_r$. Then r is replaced by the set of k non-deterministic runs $r_1, \ldots, r_k$ (which belong to the same set of nondeterministic runs as r) such that $r_n = \mathcal{D}_{i_n,t_n}(r)$ where $r_n \in \{r_1, \ldots, r_k\}$ and $t_n \in \{t_1, \ldots, t_k\}$ is a transition enabled in $i_n \in I_r$.

Definition 20 (Automaton Configuration). Let $\mathcal{I}$ be IHTA. Automaton configuration of $\mathcal{I}$ is the set of all sets of nondeterministic runs of $\mathcal{I}$.

Definition 21 (Event Accepted by an Instance). Let E be an event stream and $e \in E$. Let $\mathcal{I}$ be IHTA and i be an instance of some non-atomic state of $\mathcal{I}$. e is accepted by i if there is a transition $t \in T$ and such that t is enabled by e in i.

Definition 22 (Event Accepted by a Run). Let E be an event stream and $e \in E$. Let $\mathcal{I}$ be IHTA and r be a run of $\mathcal{I}$. Let I_r be the set of instances in r. e is accepted by r if e is accepted by all instances $i \in I_r$ for which e is relevant.

Definition 23 (Event Stream Accepted by IHTA). Let $E[\leq p]$ be an event stream arrived until a time point $p \in \mathbb{T}$. Let $\mathcal{I}$ be IHTA. If at least one run r of each set of

nondeterministic runs of $\mathcal{I}$ accepts all events of $E[\leq p]$ which are relevant for r and is in an atomic end state of $\mathcal{I}$, $E[\leq p]$ is accepted by $\mathcal{I}$.

Let $\mathcal{A}$ be automaton configuration. At the beginning $\mathcal{A}$ is empty (compare Line 1 in Listing 6.1). $\mathcal{A}$ is modified according to the automaton configuration modification algorithm in Listing 6.1. The pseudo code of the algorithm is followed by its detailed line-by-line explanation.

Listing 6.1: Automaton Configuration Modification Algorithm

```
1  A = ∅
2  forever
3    events ← get_next_events()
4    if events = ∅
5    then for each ndRuns ∈ A
6           new_ndRuns ← ∅
7           for each run ∈ ndRuns
8             new_ndRuns ← new_ndRuns ∪ DelayTransitions(run)
9           end
10          A ←(A − ndRuns) ∪ new_ndRuns
11        end
12   else for each event ∈ events
13          for each enter t ∈ T enabled by event
14            new_run ← Initialize_t()
15            for each i ∈ I_new_run
16              set_accepted(event,i)
17            end
18            new_ndRuns ← new_run
19            A ←A ∪ new_ndRuns
20          end
21        end
22        for each ndRuns ∈ A
23          new_ndRuns ← ∅
24          for each run ∈ ndRuns
25            for each permutation ∈ permutations(events)
26              new_ndRuns ← new_ndRuns ∪ EventTransitions(run,permutation)
27            end
28          end
29          if new_ndRuns = ∅
30          then return false
31          else A ←(A − ndRuns) ∪ new_ndRuns
32          end
33        end
34   end
35 end
36 function DelayTransitions(run)
37   new_runs ← ∅
38   for each i ∈ I_run
39     for each t ∈ T enabled in i
```

```
40          new_run ← D_{i,t}(run)
41          new_runs ← new_runs ∪ DelayTransitions(new_run)
42        end
43    end
44    if new_runs = ∅
45    then return run
46    else return new_runs
47    end
48  end
49  function EventTransitions(run,events)
50    new_runs ← ∅
51    relevant_events ← events
52    for each event ∈ relevant_events
53      if event is relevant for run
54      then for each i ∈ I_run
55              for each t ∈ T enabled in i
56                if t is enabled by event
57                then if ¬get_accepted(event,i)
58                     then set_accepted(event,i)
59                          new_run ← D_{i,t}(run)
60                          new_runs ← new_runs ∪ EventTransitions(new_run,events)
61                     end
62                else new_run ← D_{i,t}(run)
63                     new_runs ← new_runs ∪ EventTransitions(new_run,events)
64                end
65             end
66             if event is relevant for i and ¬get_accepted(event,i)
67             then new_runs ← ∅
68                  return new_runs
69             end
70           end
71      else relevant_events ← relevant_events − event
72      end
73    end
74    if relevant_events = ∅
75    then return run
76    else return new_runs
77    end
78  end
```

The algorithm in Listing 6.1 runs until the event stream violates the IHTA with the automaton configuration $\mathcal{A}$. Otherwise the algorithm does not terminate. An iteration of the main loop in lines 2–35 takes place every time when either there is at least one incoming event, i.e. $events \neq \emptyset$ (line 3), or at least one delay transition can be executed.

If there are no incoming events (line 4), only delay transitions can be processed (lines 5–11). For each set of nondeterministic runs *ndRuns* and for each *run* in it all enabled delay transitions are fired by calling the function $DelayTransitions(run)$ in line 8. Since delay transitions can be performed nondeterministically, i.e. when in one state more than one delay transitions are triggered at the same time, the *run* can be split into several nondeterministic runs (compare Definition 19). All of them are accumulated in the set of new nondeterministic runs new_ndRuns in line 8. Finally, the old set of nondeterministic runs *ndRuns* of the *run* is replaced by the new set of nondeterministic runs new_ndRuns in the automaton configuration $\mathcal{A}$ in line 10.

If there are incoming events, both delay and event transitions can be processed (lines 12–34). First in lines 12–21, all enter transitions enabled by the input events are triggered. These transitions create new runs as defined by Definition 16 in line 14. These new runs are added to the automaton configuration $\mathcal{A}$ in line 19. Since an event is processed by an instance only once, line 16 saves that each newly created instance already processed the event which had triggered the respective enter transition. Note that this first step is the only step of the algorithm which is possible when the main loop in lines 2–35 is called for the first time with $\mathcal{A} = \emptyset$. If at the end of this first step, $\mathcal{A}$ contains newly created runs, the second step described in the following becomes possible.

Second in lines 22–33, the algorithm checks for each run of each set of nondeterministic runs whether the run accepts each relevant event (Definition 22) of at least one permutation of the incoming events by calling the function $EventDelayTransitions(run, permutation)$ for each *run* and each *permutation* of events in line 26. If it is the case, this run is replaced by the resulting new runs, otherwise this run is deleted. This is explained in more detail in the following.

Let *events* be the set of incoming events with the same end time point of their occurrence times. Assume all *events* are relevant for the same run. As motivated above all possible permutations of *events* must be considered. There are $|events|!$ (the factorial of $|events|$) permutations where $|events|$ denotes the number of *events*.

For each set of nondeterministic runs *ndRuns* (line 22), for each *run* in it (line 24), and for each *permutation* of the incoming events (line 25), it is tested whether the *run* accepts all relevant events of the *permutation* (all respective enabled event or delay transitions are fired) in line 26. The set new_ndRuns accumulates all new runs resulting from the

runs which were able to accept all relevant events in at least one permutation. If no *run* is able to accept all the incoming events in any *permutation*, the set *new_ndRuns* is empty. Otherwise the set *new_ndRuns* contains new nondeterministic runs.

If the set *new_ndRuns* is not empty, the set of old nondeterministic runs *ndRuns* of the *run* is replaced by the set of new nondeterministic runs *new_ndRuns* in the automaton configuration $\mathcal{A}$ in line 31. Otherwise the input event stream violates the IHTA with the automaton configuration $\mathcal{A}$. Compare Definition 23. Therefore, the execution of the algorithm is stopped by returning *false* in line 30.

The main loop is based on two auxiliary functions $DelayTransitions(run)$ processing all enabled delay transitions in the *run* and $EventTransitions(run, events)$ processing all enabled event and delay transitions in the *run*. Separate treatment of event transitions is necessary because $EventTransitions(run, events)$ returns a (nonempty) set of new runs only if the *run* accepted all relevant *events*. Otherwise the returned set is empty. $DelayTransitions(run)$, in contrast, always returns a nonempty set of new runs. If no delay transition can fire in the *run*, the unchanged *run* is returned. Both functions are described in more detail in the following.

$DelayTransitions(run)$ takes a single *run* as its argument and returns the set *new_runs* after recursively performing all delay transitions which are possible at the current time point in the *run*.

More exactly: For each instance i in the set of instances I_{run} of the *run* (line 38) and for each delay transition t enabled in i (line 39), t is fired in i in line 40. The result is a *new_run*. Compare Definition 18. Other enabled delay transitions are recursively performed on the *new_run* in line 41. All these nondeterministic runs are accumulated by the set *new_runs* in line 41. If the set *new_runs* is not empty, it is returned in line 46. Otherwise the unchanged *run* is returned in line 45.

$EventTransitions(run, events)$ is similar to $DelayTransitions(run)$ but it additionally takes an ordered list of events, *events*, as an argument and processes all enabled event and delay transitions in the *run*. The function returns a nonempty set of new runs only if all relevant *events* are accepted by the *run*, otherwise the resulting set is empty.

More exactly: For each *event* of the input *events* (line 52) it is tested whether it is relevant for the *run* in line 53. If it is the case then for each instance i in the set of instances I_{run} of the *run* (line 54) and for each event or delay transition t enabled in i (line 55), t is fired in i in line 59 or line 62 depending on whether t is an event or a delay transition (see below). The result is a *new_run*.

If t is a delay transition, it is treated as described above (compare lines 40–41 with lines

62–63) with the only difference that the new_run must still process all $events$. Therefore, $EventTransitions(new\_run, events)$ is recursively called in line 63.

If t is an event transition (line 56) and the $event$ has not been accepted by the instance i yet (line 57), the $event$ is marked as accepted by i in line 58, t is triggered by the $event$, and the resulting new run is saved in new_run in line 59. The function is called recursively on the new_run in line 60.

If the $event$ is relevant for the instance i in the run but has not been accepted by i (line 66), then the run was not able to accept all events (compare Definition 22) and the resulting set new_runs is empty (line 67–68).

If some $event$ of the input $events$ is not relevant for the run then it is deleted from the list $relevant\_events$ in line 71.

If no event of the input $events$ is relevant for the run (line 74), the set $relevant\_events$ is empty and the unchanged run is returned in line 75. Otherwise in line 76, the set new_runs is returned.

Termination

Let $\mathcal{I}$ be IHTA.

The algorithm in Listing 6.1 does intentionally not terminate. However, each iteration of the main loop in lines 2–35 terminates under the following conditions:

1. For each time point t, the number of incoming events $|events|$ is finite.
2. For each time point, there must be no cycles of delay transitions in $\mathcal{I}$, i.e. the same delay transition may not be processed more than once by runs yielded from calls of $DelayTransitions(run)$.

The number of transitions $|T|$ is finite. The maximum number of instances j created by an instantiating transition is finite.

The number of sets of nondeterministic runs n_s in the automaton configuration of $\mathcal{I}$ is finite. The total number of runs n_r in all sets of nondeterministic runs is also finite. The total number of instances n_i in all runs is finite. $n_s \in \mathcal{O}(n_i)$ and $n_r \in \mathcal{O}(n_i)$ since in the worst each instance belongs to its own run which is the only run of a set of nondeterministic runs.

Let $|E[\leq p]|$ denote the number of events relevant for $\mathcal{I}$ and arrived on the stream E until the time point $p \in \mathbb{T}$. Then $n_i \in \mathcal{O}(|E[\leq p]| \cdot |E[\leq p]|! \cdot |T| \cdot j)$ because in the worst case each event arrived on the stream E until the time point $p \in \mathbb{T}$ in each permutation of events triggers an instantiating transition $t \in T$ creating at most j instances.

For these reasons the number of iterations of all nested loops of the algorithm is finite.

Complexity

$$
\begin{aligned}
&DelayTransitions(run) && \in \mathcal{O}(n_i \cdot |T|) \\
&EventTransitions(run, events) && \in \mathcal{O}(n_i \cdot |T| \cdot |events|) \\
&main(\mathcal{A}, events) && \in \mathcal{O}(n_i \cdot |events|! \cdot n_i \cdot |T| \cdot |events|) \\
& && \in \mathcal{O}((|E[\leq p]| \cdot |E[\leq p]|! \cdot |T| \cdot j)^2 \cdot |events|! \cdot |T| \cdot |events|) \\
& && \in \mathcal{O}(|E[\leq p]|^2 \cdot |E[\leq p]|!^2 \cdot |T|^3 \cdot j^2 \cdot |events|! \cdot |events|)
\end{aligned}
$$

Chapter 7

Event Stream Constraint Language

This chapter is devoted to the Event Stream Constraint Language ESCL. We start in Section 7.1 with the summary of the main features of the language. Section 7.2 illustrates these features by examples, explains the syntax and the informal semantics of the language. Section 7.3 presents the grammar of ESCL. Section 7.4 explains the normalization of ESCL constraints to facilitate their use for semantic optimization of CEP queries. And finally, Section 7.5 introduces the formal declarative semantics of the language.

7.1 Main Features

This section summarizes the main features of the Event Stream Constraint Language in a form of a list. Illustrating examples are given in Section 7.2. These features are the following (compare Section 2.3):

1. ESCL is a *declarative first-order logic language*. An ESCL constraint is a logic formula which is either a fact or a rule. A rule has a body and a head. If a constraint body matches a stream, the constraint head must be satisfied by the stream. A rule expresses causality very naturally. A fact is a rule with an empty body, i.e. the head of a fact holds unconditionally.

2. ESCL is *tailored to the peculiarities of CEP* described in Section 2.2. In particular, it formulates constraints within time intervals which can also be application states. For this reason, ESCL allows not only event but also *state queries*. As motivated in Section 2.3.1, states are very important (and even indispensable) for many event-based applications. Indeed, some constraints hold during particular states only. States cannot be replaced by time intervals because the beginning and the duration of states is usually unknown beforehand.

3. ESCL is *readable* and *easy to use*. Thanks to event and state identification simple but expressive cardinality, temporal, data, and spatial constraints are possible. Modules and local definitions help to keep ESCL constraints short.

4. ESCL is *expressive*. It supports the following kinds of constraints:

   - *Cardinality constraints* describe the number of events or states during time intervals. More exactly, the lower bound, the upper bound, the exact number of events or states and an arbitrary combination of these cardinality specifications by logic *and* ($\wedge$) and *or* ($\vee$) are allowed by the language. To express cardinality constraints, the existentional quantifier ($\exists$) extended by cardinality specifications and grouping is used. Grouping specifies the data variables of events or states according to the values of which these events or states are grouped together and counted.
   - *Data constraints* express functional dependencies and other relations between the attribute values carried by events and states. ESCL allows access to event and state data by means of variables.
   - *Temporal relations* express the dependencies between the occurrence time of events and the validity time of states. They are the most important group of constraints needed in all CEP applications. Consider, for example, the use cases in Section 2.1. ESCL allows access to the beginning and the end of the event occurrence time and the state validity time. As a consequence, ESCL temporal constraints cover Allens relations [3] but are not restricted to them. ESCL supports relative timer events [50] and states as well as windows defined in Section 4.1. Each ESCL constraint is formulated within a time interval called the validity time of the constraint. Constraints with the same validity time belong to the same module. Both constraints and queries can be modularized in this way so that (1) queries can be suspended if their evaluation time has not begun yet or is already over and (2) only those constraints are relevant for the semantic optimization of a query the validity time intervals of which overlap the query evaluation time.
   - *Spatial relations* describe the dependencies between the locations events are sent from. These constraints are needed only in some CEP application like, for example, emergency detection in a metro station described in Section 2.1.2.

5. ESCL allows for formulating constraints on both a stream and a conventional database, in particular *CEP and database queries can be combined within one ESCL constraint*. This is an indispensable feature since some CEP constraints hold only

under consideration of static knowledge saved in a database and not available on the stream.

6. ESCL has strong *formal foundations*. Consider its declarative semantics in Section 7.5. It is a kind of Herbrand interpretation [29] extended to deal with temporal relations and event and state identifiers. The model relation between an interpretation and an ESCL constraint is defined very similarly to the Tarski model relationship between an interpretation and a formula [29]. Since ESCL is developed for the semantic optimization of CEP, the operational semantics of the language is defined by the algorithm semantically rewriting StreamLog queries with respect to ESCL constraints. In Chapter 10, the algorithm is given and proven to terminate and be correct with respect to the declarative semantics of ESCL. However, ESCL is not restricted to this purpose. It could also be used, for example, for stream verification which is out of the scope of this report. For this purpose other operational semantics is required but the declarative semantics of ESCL remains the same.

7. ESCL is *modularly defined*, i.e. the definitions of the grammar and the declarative semantics of the language are divided into four modules: (1) The core of the language, (2) Data constraints, (3) Temporal constraints, and (4) Spatial constraints. Such modular definition of ESCL allows easy extension of the language by additional features or even modules, (ex-)change of a module, usage of a fragment of ESCL (for example, spatial constraints can be omitted for such a use cases as an online auction described in Section 2.1.1). Besides, the modularity of the formal definition of the language increases its readability and understandability.

7.2 Syntax and Informal Semantics

This section illustrates some of the features of ESCL briefly described in Section 7.1 by examples, explains the syntax of the language and its informal semantics.

Some ESCL constraints presented in this section can probably contribute nothing to the semantic optimization of a query (Chapter 10) but, as mentioned above, ESCL can also be used for other purposes, for example, for stream verification. Therefore, the language is deliberately designed to express most prevalent CEP constraints such as cardinality, causal, data, temporal, and spatial dependencies between events and states.

We start in Section 7.2.1 with the explanation why it is preferable to use minimal and complete sets of optimal constraints for semantic query optimization. Section 7.2.2 and Section 7.2.3 are devoted to the modularization of ESCL constraints and local definitions

within a module of ESCL constraints. Sections 7.2.4, 7.2.5, 7.2.6, and 7.2.7 give examples and explain the informal semantics of the ESCL cardinality, temporal, data, and spatial constraints respectively.

7.2.1 Minimal and Complete Sets of Optimal Constraints

For semantic query optimization, a constraint should be optimal and a set of constraints should be minimal and complete.

Definition 24 (**A subconstraint**). Let $C : h \leftarrow b$ be an ESCL constraint with the body b and the head h. b is a conjunction of atomic queries, h is a conjunction or a disjunction of atomic queries. $C' : h' \leftarrow b'$ is a subconstraint of C if each atomic query in h' can be found in h or each atomic query in b' can be found in b.

Definition 25 (**An optimal constraint**). A constraint is *optimal* if it is not implied by its subconstraint(s).

Definition 26 (**A minimal set of constraints**). A set of constraints is *minimal* if no constraint of the set is implied by other constraint(s) of the set.

A minimal set of optimal constraints contributes to the efficiency of the semantic query optimization method since less constraints have to be taken into account (because of the minimality of a constraint set) and only the relevant information is used for query optimization (because of the optimality of constraints). The user is responsible for optimal constraints. A minimal set of constraints can be computed automatically (which is out of the scope of this report).

Definition 27 (**A complete set of constraints**). A set of constraints is *complete* if the set expresses the entire application knowledge.

A complete set of constraints makes the semantic query optimization method more powerful. The more semantic information, the more possibilities to improve the evaluation of a query. The user is responsible for a complete set of constraints.

In this section, a minimal set of optimal ESCL constraints for the online auction use case (Section 2.1.1) is considered as an example. This set is not complete for the sake of brevity.

7.2.2 Modularization

Listing 7.1: The set of cardinality constraints which hold during the application state *Item offer*. Compare Figure 6.1.

```
1  WHILE state: itemOffer( auctionID(A), itemID(I) )
2  LET
3        IN ANY CASE
4          ∃≥0 event: bid( auctionID(A), itemID(I), bidderID(B1), value(V1) )
5          group-by {A, I}
6        END
7        ∧
8        IN ANY CASE
9            ∃=0∨≥3 event: hammerBeat( auctionID(A), itemID(I), bidderID(B2), value(V2) )
10           group-by {A, I}
11       END
12       ∧
13       IN ANY CASE
14           ∃≤1 event: sell( auctionID(A), itemID(I), bidderID(B3), value(V3) )
15           group-by {A, I}
16       END
17       ∧
18       ( IN ANY CASE
19             ∃=1 event: itemDescription( auctionID(A), itemID(I1), bidderID(B4), value(V4) )
20             group-by {A}
21         END
22         ∨
23         IN ANY CASE ∃=1 event: auctionEnd( auctionID(A) ) END )
24       ∧
25       IN ANY CASE ∃=1 state: I0( auctionID(A), itemID(I) ) END
26       ∧
27       IN ANY CASE ∃≥0 state: I1( auctionID(A), itemID(I) ) END
28       ∧
29       IN ANY CASE ∃≥0 state: I2( auctionID(A), itemID(I) ) END
30       ∧
31       IN ANY CASE ∃≥0 state: I3( auctionID(A), itemID(I) ) END
32       ∧
33       IN ANY CASE ∃≤1 state: I4( auctionID(A), itemID(I) ) END
34       ∧
35       IN ANY CASE ∃=1 state: I5( auctionID(A), itemID(I) ) END
36       ∧
37       LET
38           DETECT
39             event: firstBid ( auctionID(A), itemID(I) )
40           ON
41             state s: I0( auctionID(A), itemID(I) ) ∧
42             while s: event: bid( auctionID(A), itemID(I), bidderID(B5), value(V5) )
43           END
```

```
44      IN
45          IF    event: firstBid ( auctionID(A), itemID(I) )
46          THEN ∃≥3 event: hammerBeat( auctionID(A), itemID(I), bidderID(B6), value(V6) )
47               group-by {A, I}
48          END
49          ∧
50          IF    event: firstBid ( auctionID(A), itemID(I) )
51          THEN ∃=1 event: sell( auctionID(A), itemID(I), bidderID(B7), value(V7) )
52               group-by {A, I}
53          END
54          ∧
55          IF    event: firstBid ( auctionID(A), itemID(I) )
56          THEN ∃≥1 state: I1( auctionID(A), itemID(I) )
57          END
58          ∧
59          IF    event: firstBid ( auctionID(A), itemID(I) )
60          THEN ∃≥1 state: I2( auctionID(A), itemID(I) )
61          END
62          ∧
63          IF    event: firstBid ( auctionID(A), itemID(I) )
64          THEN ∃≥1 state: I3( auctionID(A), itemID(I) )
65          END
66          ∧
67          IF    event: firstBid ( auctionID(A), itemID(I) )
68          THEN ∃=1 state: I4( auctionID(A), itemID(I) )
69          END
70      END
71  END
```

Compare the set of cardinality constraints in Listing 7.1 with the IHTA in Figure 6.1. These cardinality constraints hold during the application state *Item offer*, therefore they are specified within this state by means of the statement WHILE state: itemOffer(auctionID(A), itemID(I)) in Lines 1–2.

To differentiate between atomic event and state queries they are prefixed by the key words *event:* or *state:* respectively. The values of the attributes $auctionID$ and $itemID$ differentiate between auctions and items respectively. The values of these attributes must be the same in all literals of the constraint set.

This constraint set build a module the validity time of which is the state *Item offer*. This state is the default evaluation time of all literals in the constraint bodies (IF part of the constraints). However, tighter evaluation time intervals can be specified for the literals, for example, the evaluation time of the atomic event query in Line 43 is the state I_0. (Consider the use of the state identifier s helping to formulate this temporal condition

shortly.) The important thing is that a literal evaluation time interval must be covered by the default literal evaluation time interval. Without modularization the default time interval would have to be added to each constraint of the module containing at least one literal without a time interval. This would make constraints not only unreadable but also more difficult to evaluate since only those constraints are relevant for a semantic optimization of a query the validity time of which overlaps the query evaluation time.

The literals in a constraint head, a constraint body as well as constraints in a module are connected by means of logic *and* $\wedge$ or *or* $\vee$. For example, each item description is followed by either the description of the next item in the auction or the end of the auction. This is expressed by connecting the constraint in Lines 19–22 to the constraint in Line 24 by means of the logic $\vee$. This disjunction of constraints is connected to the conjunction of all other constraints of the module by logic $\wedge$ meaning that all these constraints hold for an item during it is presented.

7.2.3 Local Definitions

Like modularization, local definitions help to keep constraints short. A local definition is a CEP query deriving a complex event (or a state) from a combination of multiple events and/or states. This derived events/states are than used in the respective constraints as a shortcut for the data they were derived from. For example, the CEP query in Lines 39–44 derives the complex event $firstBid$ from a *bid* event arriving during the state I_0 which is the start state of the IHTA describing an item offer during an auction, see Figure 6.1. This complex event is queried in all respective constraints in Lines 46–70. Without local definition the conjunction in the body of the CEP query in Lines 42–43 would have to be copied in the bodies of all constraints in Lines 46–70.

7.2.4 Cardinality Constraints

A cardinality constraint describes the number of events or states during a time interval. More exactly: The events and states happening or holding during this time interval are saved, grouped together according to the data they carry and the number of elements in each group is specified by the constraint. The question is according to which attribute values these events or states are grouped. To express this the statement group-by is used. Consider for example the cardinality constraint in Lines 9–12. It says that there are either zero, 3, or more than 3 hammer beats for each item in an auction. Therefore the respective events are grouped according to the values of attributes *auctionID* and *itemID* only. If a cardinality constraint contains no group-by statement, the respective matched events and states are grouped according to all their (data) attribute values, like for example in the

constraint in Line 24.

7.2.5 Temporal Constraints

Constraints expressing temporal dependencies between events and states are a very important kind of constraints needed in all CEP applications. Actually, (almost) all CEP constraints are also temporal constraints since they usually hold during particular time intervals and not always.

Consider Listing 7.2 containing some constraints describing the temporal order of states during an item is presented. These simple temporal constraints belong to the same module as the constraints in Listing 7.1.

Listing 7.2: The set of temporal constraints which hold during the application state *Item offer*. Compare Figure 6.1.

```
1  WHILE state: itemOffer( auctionID(A), itemID(I) )
2  LET
3      IF state i: I_0( auctionID(A), itemID(I) ) ∧
4          state j: I_1( auctionID(A), itemID(I) )
5      THEN i before j
6      END
7      ∧
8      IF state i: I_0( auctionID(A), itemID(I) ) ∧
9          state j: I_2( auctionID(A), itemID(I) )
10     THEN i before j
11     END
12     ∧
13     IF state i: I_0( auctionID(A), itemID(I) ) ∧
14         state j: I_3( auctionID(A), itemID(I) )
15     THEN i before j
16     END
17     ∧
18     IF state i: I_0( auctionID(A), itemID(I) ) ∧
19         state j: I_4( auctionID(A), itemID(I) )
20     THEN i before j
21     END
22     ∧
23     IF state i: I_0( auctionID(A), itemID(I) ) ∧
24         state j: I_5( auctionID(A), itemID(I) )
25     THEN i before j
26     END
27     ∧
28     ...
29 END
```

ESCL temporal constraints cover Allen's temporal relations [3] but are not restricted to them. ESCL allows for formulating temporal constraints on the beginning $b(i)$ and the end $e(i)$ of the occurrence time (or the validity time) of the event (or the state) referenced by i directly. All Allen's relations can be led back to the comparisons of values returned by $b(i)$ and $e(j)$ where i and j are identifiers (see Section 7.5). But there are constraints which can be expressed by a comparison of the values returned by these functions but are not covered by Allen's relations, e.g., $e(i) \leq b(j)$ is a valid ESCL temporal constraint which means that j begins at the earliest when i is over. Such comparisons may also involve constants, e.g., $e(i) + 1min \leq b(j)$ means that j begins at the earliest one minute after i is over. In other words, Allen's relations are build-in in ESCL as syntactic sugar for the most commonly used comparisons of values returned by the functions $b()$ and $e()$. But these comparisons are also allowed in ESCL to express other temporal relations.

Module validity time can be specified by means of an (relative timer) event or state or a window. Relative timer events are events the occurrence time of which is defined relatively to other events. They have been initially proposed in [50]. Relative timer states are defined analogously. An example of constraints which hold during an application state is given in Listing 7.1. Consider the example of a constraint which holds during the sliding window in Listing 7.3. The number of auctions running during the last three hours is computed and the result is returned every hour.

Listing 7.3: A cardinality constriant which holds during the sliding window

```
1 WHILE Range 3h Slide 1h
2 LET
3       IN ANY CASE  ∃=1 event: numberOfAuctions ( value(V) ) END
4 END
```

Literal validity time can be defined by an (relative timer) event or state or their combination using the interval combination operator. Consider Lines 4–16 in Listing 7.4 for the constraint in which the validity time of the literal in Lines 12–13 is defined by means of the interval combination operator each. This constraint means that if there is a sell event for an item then during each of the unique states I_1, I_2, and I_3 (which hold in this order, compare Figure 6.1) there is exactly one hammer beat event which carries exactly the same data as the sell event such that there is no bid event between these hammer beat events and the sell event. Consider also the use of the apart-by statement in Lines 14–15 addressing the time intervals between the hammer beat events and the sell event.

Listing 7.4: Constraints using the interval combination operator each and the apart-by statement.

```
1 WHILE state: itemOffer( auctionID(A), itemID(I) )
2 LET
```

```
3    IF    event s: sell( auctionID(A), itemID(I), bidderID(B), value(V) )
4    THEN ∃=1 state i1: I1( auctionID(A), itemID(I) ) ∧
5         ∃=1 state i2: I2( auctionID(A), itemID(I) ) ∧
6         ∃=1 state i3: I3( auctionID(A), itemID(I) ) ∧
7         i1 before i2 ∧
8         i2 before i3 ∧
9         while each{i1, i2, i3}: ∃=1 event h:
10                hammerBeat( auctionID(A), itemID(I), bidderID(B), value(V) ) ∧
11        {h,s} apart-by no event:
12                bid( auctionID(A), itemID(I), bidderID(B1), value(V1) )
13    END
14    ∧
15    IF event h1: hammerBeat( auctionID(A), itemID(I), bidderID(B1), value(V1) ) ∧
16       event h2: hammerBeat( auctionID(A), itemID(I), bidderID(B2), value(V2) ) ∧
17       {h1,h2} apart-by min 1 event:
18                bid( auctionID(A), itemID(I), bidderID(B3), value(V3) )
19    THEN V1 < V2
20    END
21 END
```

Let i and j be identifiers and q be an atomic query. There are six interval combination operators. They are:

1. each{i,j}: q means that q holds during both time intervals of i and j.

2. any{i,j}: q means that q holds either during the time interval of i or j or both.

3. intersection{i,j}: q means that q holds during the time interval which is the intersection of the time intervals of i and j. If these time intervals do not overlap the intersection is not defined.

4. union{i,j}: q means that q holds during the time interval which is the union of the time intervals of i and j. If these time intervals do not overlap the union is not defined. In this case i and j must reference events. They may not reference states since per definition states must be evaluated as soon as they begin even if their end is not known yet. But in order to compute union of states the end of these states must be known which leads to a contradiction.

5. cover-each{i,j}: q means that q holds during the time interval covering both i and j but not the time interval in between. The result is the set of time intervals $\{i, j\}$ if i and j do not overlap.

6. cover-all{i,j}: q means that q holds during the time interval covering both i and j and the time interval in between. The result is always one time interval.

each and any are syntactic sugar in contrast to the other four interval combination operators.

By means of apart-by statement involved (data) constraints become possible. Consider the constraint in Lines 18–25 in Listing 7.4. It says that for two hammer beat events with at least one bid event (for the same item) in between, the price carried by the second hammer beat event must be higher than the price carried by the first hammer beat event.

Let i and j be identifiers. The statement {i,j} apart-by ... can specify an upper bound, a lower bound, or the exact number of events or states appearing between i and j as well as their combination by means of logic *and* $\wedge$ or *or* $\vee$. The statement can also specify an upper bound, a lower bound, or the exact duration of the time interval between i and j as well as their combination by the logic operators. apart-by statement is no syntactic sugar.

7.2.6 Data Constraints

CEP data constraints describe functional and other dependencies between the values of data attributes carried by events and states. Listing 7.5 contains simple data constraints expressing the functional dependency of an auction identifier from an item identifier which holds for all events and states carrying the data attributes $auctionID$ and $itemID$. Only some constraints are given in Listing 7.5. All the other are defined analogously.

Listing 7.5: Data constraints expressing the functional dependency of an auction identifier from an item identifier.

```
1  WHILE state: auction( auctionID(A) )
2  LET
3      IF    event: itemDescription( auctionID(A1), itemID(I) ) ∧
4            event: bid( auctionID(A2), itemID(I) )
5      THEN A1 = A2
6      END
7      ∧
8      IF    event: itemDescription( auctionID(A1), itemID(I) ) ∧
9            event: hammerBeat( auctionID(A2), itemID(I) )
10     THEN A1 = A2
11     END
12     ∧
13     IF    event: itemDescription( auctionID(A1), itemID(I) ) ∧
14           event: sell( auctionID(A2), itemID(I) )
15     THEN A1 = A2
16     END
17     ∧
18     IF    event: itemDescription( auctionID(A1), itemID(I) ) ∧
```

```
19            state: I0( auctionID(A2), itemID(I) )
20      THEN A1 = A2
21      END
22      ∧
23      ...
24 END
```

Consider more involved data constraints in Listing 7.6 expressing that a price for an item increases during its sell. Note the use of the local definition in Lines 4–9 in all three constraints.

Listing 7.6: Data constraints expressing that a price for an item increases during its sell.

```
1  WHILE state: itemOffer( auctionID(A), itemID(I) )
2  LET
3      LET
4          DETECT
5              event: e( auctionID(A), itemID(I), bidderID(B), value(V) )
6          ON
7              event: bid( auctionID(A), itemID(I), bidderID(B), value(V) ) ∨
8              event: hammerBeat( auctionID(A), itemID(I), bidderID(B), value(V) )
9          END
10     IN
11         IF event: itemDescription( auctionID(A), itemID(I), bidderID(B), value(V1) ) ∧
12             ( event: e( auctionID(A), itemID(I), bidderID(B), value(V2) ) ∨
13               event: sell( auctionID(A), itemID(I), bidderID(B), value(V2) ) )
14         THEN V1 ≤ V2
15         END
16         ∧
17         IF event i: e( auctionID(A), itemID(I), bidderID(B), value(V1) ) ∧
18             event j: e( auctionID(A), itemID(I), bidderID(B), value(V2) ) ∧
19             i before j
20         THEN V1 < V2
21         END
22         ∧
23         IF event: e( auctionID(A), itemID(I), bidderID(B), value(V1) ) ∧
24             event: sell( auctionID(A), itemID(I), bidderID(B), value(V2) )
25         THEN V1 ≤ V2
26         END
27     END
28 END
```

7.2.7 Spatial Constraints

Spatial constraints describe the dependencies between the locations events were sent from. This locations can be rooms, areas, or the like depending on the application. In contrast to temporal constraints, spatial constraints are no inherent part of the semantics of all CEP applications. The semantics of some CEP application does not involve spatial constraints like, for example, an online auction described in Section 2.1.1. But nevertheless there are application in the semantics of which spatial constraints play an important role. An example of such an application in emergency detection in a metro station described Section 2.1.2. That is why this group of constraints is also supported by ESCL.

Consider Listing 7.7. It gives an example of the spatial constraint expressing that the spread of smoke is limited to the same room or neighbor rooms during one minute. For this and the following examples we assume that each sensor sends an event *smoke* only when smoke is initially detected.

Note that this constraint holds always. That is why its validity time is an unbounded window. Nevertheless this constraint can be verified without problems since only the smoke events arriving during one minute after a smoke event are relevant for the constraint.

Listing 7.7: The spatial constraint expressing that the spread of smoke is limited during a short period of time.

```
1  WHILE unbounded
2  LET
3      IF event i: smoke( sensorID(S_1), area(A_1), intensity(I_1) ) ∧
4         event j: smoke( sensorID(S_2), area(A_2), intensity(I_2) ) ∧
5         i before j ∧
6         {i,j} within 1 min
7      THEN {i,j} sent from the same room ∨
8           {i,j} sent from the neighbor rooms
9      END
10 END
```

Spatial constraints can also express an upper bound, a lower bound or an exact distance between the locations events were sent from as well as the combination of this values by means of logic operators $\wedge$ and $\vee$. For instance, the spatial constraint in Listing 7.8 restricts the spread of smoke within the same area to be at most 100 meter per minute.

Listing 7.8: The spatial constraint expressing that the spread of smoke is limited during a short period of time.

```
1  WHILE unbounded
2  LET
3      IF event i: smoke( sensorID(S_1), area(A), intensity(I_1) ) ∧
```

```
4       event j: smoke( sensorID(S2), area(A), intensity(I2) ) ∧
5       i before j ∧
6       {i,j} within 1 min
7   THEN {i,j} apart-by max 100 m
8   END
9 END
```

Besides, spatial constraints express whether the location an event was sent from is south, north, east, west or the like of the location another event was sent from. For example, the constraint in Listing 7.9 expresses that the spread of smoke in an area depends on the ventilation direction in this area.

Listing 7.9: The spatial constraint expressing that the spread of smoke in an area depends on the ventilation direction in this area.

```
1 WHILE unbounded
2 LET
3   IF event i: smoke( sensorID(S1), area(A), intensity(I1) ) ∧
4       event j: smoke( sensorID(S2), area(A), intensity(I2) ) ∧
5       i before j ∧
6       ventilation( area(A), direction(south) )
7   THEN j south of i
8   END
9 END
```

The above constraint accesses data in a conventional database since ventilation is a usual database relation, i.e. ESCL expresses constraints not only on a stream but also on a database. In particular, the language allows the combination CEP and database queries within one constraint. This feature is indispensable since some constraints (e.g., the one in Listing 7.9) hold only under consideration of static knowledge saved in a database and not available on the stream.

7.3 Grammar

This section is devoted to the grammar of ESCL. For the sake of understandability and extensibility, it is divided into four modules. Section 7.3.1 defines the grammar of the core of the language. Sections 7.3.2, 7.3.3, and 7.3.4 specify the grammar of the ESCL data, temporal, and spatial constraints respectively.

7.3.1 The Core of the Language

This section is devoted to the grammar of the core of ESCL. Temporal constraints (the grammar of which is defined in Section 7.3.3) are refered to by the rules $ModuleValTime$, $LiteralValTime$, and $TempCond$. Constraints on the event and state data (the grammar of which is defined in Section 7.3.2) are refered to by the rule $DataCond$. Spatial constraints (the grammar of which is specified in Section 7.3.4) are refered to by the rule $SpatialCond$.

The grammar of a deductive rule (the rule *DeductiveRule* in the grammar below) and relative timer events and states (the rule $RelTimer$) is adopted from [50] with the only difference that deductive rules may quire not only events but also states and database relations and relative timer are defined not only for events but also for states.

$$
\begin{array}{lcl}
ModuleSet & ::= & Module* \\
Module & ::= & \text{"}WHILE\text{"}\ ModuleValTime \\
 & & \text{"}LET\text{"}\ ConstraintSet \\
 & & \text{"}END\text{"} \\
LocalDef & ::= & \text{"}LET\text{"}\ DeductiveRule+ \\
 & & \text{"}IN\text{"}\ ConstraintSet \\
 & & \text{"}END\text{"} \\
ConstraintSet & ::= & Module \mid LocalDef \mid Constraint \\
 & & \mid (\ \text{"(?}ConstraintSet\text{"} \wedge \text{"}ConstraintSet\text{")?}\ ) \\
 & & \mid (\ \text{"(?}ConstraintSet\text{"} \vee \text{"}ConstraintSet\text{")?}\ ) \\
Constraint & ::= & (\ \text{"}IN\ ANY\ CASE\text{"}\ Query\ \text{"}END\text{"}\ ) \\
 & & \mid (\ \text{"}IF\text{"}\ Query\ \text{"}THEN\text{"}\ Query\ \text{"}END\text{"}\ ) \\
Query & ::= & AtomicQuery \\
 & & \mid (\ \text{"(?}\ Query\ \text{"} \wedge \text{"}\ Query\ \text{")?}\ ) \\
 & & \mid (\ \text{"(?}\ Query\ \text{"} \vee \text{"}\ Query\ \text{")?}\ ) \\
AtomicQuery & ::= & Literal \mid RelTimer \mid Condition \\
 & & \\
Literal & ::= & (\text{"}while\text{"}\ LiteralValTime\ \text{":"})? \\
 & & (\text{"}\exists\text{"}_{CardSpec} \mid \text{"}\neg\text{"})?\ (Identifier \mid IdentifierDecl?\ Atom) \\
 & & (\text{"}group\text{-}by\text{"}\ \text{"}\{\text{"}(Variable\ \text{","}?) + \text{"}\}\text{"})? \\
IdentifierDecl & ::= & (\text{"}event\text{"} \mid \text{"}state\text{"})\ Identifier?\ \text{":"} \\
CardSpec & ::= & (\ CompOp\ (Number \mid Variable)\ ) \\
 & & \mid (\ \text{"(?}\ CardSpec\ \text{"} \wedge \text{"}\ CardSpec\ \text{")?}\ ) \\
 & & \mid (\ \text{"(?}\ CardSpec\ \text{"} \vee \text{"}\ CardSpec\ \text{")?}\ ) \\
CompOp & ::= & \text{"}=\text{"} \mid \text{"}\neq\text{"} \mid \text{"}<\text{"} \mid \text{"}\leq\text{"} \mid \text{"}>\text{"} \mid \text{"}\geq\text{"} \\
 & & \\
RelTimer & ::= & IdentifierDecl?\ RelTimerSpec\ \text{"["}\ Identifier\ \text{","}\ Duration\ \text{"]"} \\
RelTimerSpec & ::= & \text{"}extend\text{"} \mid \text{"}shorten\text{"} \mid \text{"}from\text{-}end\text{"} \mid \text{"}from\text{-}start\text{"} \\
 & & \mid \text{"}extend\text{-}begin\text{"} \mid \text{"}shorten\text{-}begin\text{"} \\
 & & \mid \text{"}shift\text{-}forward\text{"} \mid \text{"}shift\text{-}backward\text{"} \\
 & & \mid \text{"}from\text{-}end\text{-}backward\text{"} \mid \text{"}from\text{-}start\text{-}backward\text{"} \\
 & & \\
Condition & ::= & DataCond \mid TempCond \mid SpatialCond
\end{array}
$$

7.3.2 Data Constraints

This section defines the grammar of the ESCL constraints on the event and state data. These constraints are relevant for all CEP applications.

$$
\begin{array}{lcl}
DataCond & ::= & Exp\ CompOp\ Exp \\
Exp & ::= & Number \mid \text{"`"} String \text{"'"} \mid Variable \\
 & & \mid (\ Exp\ ArithOp\ Exp\ ) \mid (\ \text{"("} Exp \text{")"}\ ) \\
ArithOp & ::= & \text{"}*\text{"} \mid \text{"/"} \mid \text{"}-\text{"} \mid \text{"}+\text{"} \mid \text{"}mod\text{"}
\end{array}
$$

7.3.3 Temporal Constraints

This section defines the grammar of the ESCL temporal constraints. These constraints are relevant for all CEP applications.

The module validity time can be an event, a state, or a window (consider the rule $ModuleValTime$). Remember that a module validity time is the default validity time of each constraint of the module and of each literal in a constraint of the module.

The literal validity time (consider the rule $LiteralValTime$) can be a (relative timer) event or state or their combination by means of the interval combination operator (see the rule $InvervalCombOp$). Remember that a literal validity time interval is always covered by the validity time of the module the literal appears within.

Allen's temporal relations [3] are supported by introducing the temporal operator (see the rule $TempOp$). However, ESCL temporal constraints are not restricted to them since, for example, the rule $TempExp$ allows to construct arbitrary constraints on the beginning and the end of an event occurrence time or a state validity time.

$$
\begin{array}{lcl}
ModuleValTime & ::= & (\ IdentifierDecl?\ Atom\ ) \mid Window \\
Window & ::= & Unbounded \mid Now \mid Sliding \mid Tumbling \mid Other \\
Unbounded & ::= & \text{"}unbounded\text{"} \\
Now & ::= & \text{"}now\text{"} \\
Sliding & ::= & \text{"}Range\text{"}\ Duration\ (\text{"}Slide\text{"}\ Duration)? \\
Tumbling & ::= & \text{"}Range\text{"}\ Duration \\
Other & ::= & \text{"}[\text{"}\ String\ \text{"},\text{"}\ String\ \text{"}]\text{"} \\
 & & \\
LiteralValTime & ::= & Identifier \mid (\ IntervalCombOp\ \text{"}\{\text{"}\ (Identifier\ \text{"},\text{"}?) +\ \text{"}\}\text{"}\ ) \\
IntervalCombOp & ::= & \text{"}each\text{"} \mid \text{"}any\text{"} \mid \text{"}intersection\text{"} \mid \text{"}union\text{"} \mid \text{"}cover\text{-}each\text{"} \mid \text{"}cover\text{-}all\text{"} \\
 & & \\
TempCond & ::= & (\ Identifier\ TempOp\ Identifier\ ) \\
 & & \mid (\ TempExp\ CompOp\ TempExp\ ) \\
 & & \mid (\ \text{"}\{\text{"}\ (Identifier\ \text{"},\text{"}?) +\ \text{"}\}\text{"}\ \text{"}within\text{"}\ TempSpec\ ) \\
 & & \mid (\ \text{"}\{\text{"}\ (Identifier\ \text{"},\text{"}?) +\ \text{"}\}\text{"}\ \text{"}apart\text{-}by\text{"}\ TempSpec\ ) \\
 & & \mid (\ \text{"}\{\text{"}\ (Identifier\ \text{"},\text{"}?) +\ \text{"}\}\text{"}\ \text{"}apart\text{-}by\text{"}\ Quantity\ ) \\
TempOp & ::= & \text{"}before\text{"} \mid \text{"}after\text{"} \mid \text{"}during\text{"} \mid \text{"}contains\text{"} \mid \text{"}equals\text{"} \\
 & & \mid \text{"}overlaps\text{"} \mid \text{"}overlapped\text{-}by\text{"} \mid \text{"}meets\text{"} \mid \text{"}met\text{-}by\text{"} \\
 & & \mid \text{"}starts\text{"} \mid \text{"}started\text{-}by\text{"} \mid \text{"}finishes\text{"} \mid \text{"}finished\text{-}by\text{"} \\
TempExp & & Duration \mid (\ \text{"}b(\text{"}Identifier\text{"})\text{"}\ ) \mid (\ \text{"}e(\text{"}Identifier\text{"})\text{"}\ ) \\
 & & \mid (\ TempExp\ ArithOp\ TempExp\ ) \mid (\ \text{"}(\text{"}TempExp\text{"})\text{"}\ ) \\
TempSpec & ::= & (\ (\text{"}min\text{"}\mid\text{"}max\text{"})?\ Duration\ ) \\
 & & \mid (\ \text{"}(\text{"}?\ TempSpec\ \text{"} \wedge \text{"}\ TempSpec\ \text{"})\text{"}?\ ) \\
 & & \mid (\ \text{"}(\text{"}?\ TempSpec\ \text{"} \vee \text{"}\ TempSpec\ \text{"})\text{"}?\ ) \\
Quantity & ::= & (\ QuantitySpec\ IdentifierDecl?\ Atom\ ) \\
 & & \mid (\ \text{"}(\text{"}?\ Quantity\ \text{"} \wedge \text{"}\ Quantity\ \text{"})\text{"}?\ ) \\
 & & \mid (\ \text{"}(\text{"}?\ Quantity\ \text{"} \vee \text{"}\ Quantity\ \text{"})\text{"}?\ ) \\
QuantitySpec & ::= & \text{"}no\text{"} \mid (\ (\text{"}min\text{"}\mid\text{"}max\text{"})?\ (Number \mid Variable)\ ) \\
 & & \mid (\ \text{"}(\text{"}?\ QuantitySpec\ \text{"} \wedge \text{"}\ QuantitySpec\ \text{"})\text{"}?\ ) \\
 & & \mid (\ \text{"}(\text{"}?\ QuantitySpec\ \text{"} \vee \text{"}\ QuantitySpec\ \text{"})\text{"}?\ ) \\
Duration & ::= & (Number\ (\text{"}month\text{"}\mid\text{"}months\text{"}))? \\
 & & (Number\ (\text{"}week\text{"}\mid\text{"}weeks\text{"}))? \\
 & & (Number\ (\text{"}day\text{"}\mid\text{"}days\text{"}))?\ (Number\ (\text{"}hour\text{"}\mid\text{"}hours\text{"}))? \\
 & & (Number\ \text{"}min\text{"})?\ (Number\ \text{"}sec\text{"})?\ (Number\ \text{"}ms\text{"})?
\end{array}
$$

7.3.4 Spatial Constraints

This section defines the grammar of the ESCL spatial constraints. These constraints are relevant for many CEP applications like, for example, emergency detection in a metro station described in Section 2.1.2.

$$
\begin{array}{lll}
\textit{SpatialCond} & ::= & (\ \textit{Identifier SpatialOp Identifier}\ ) \\
 & & |\ (\ \text{"}\{\text{"}\ (\textit{Identifier}\ \text{","?})+\ \text{"}\}\text{"}\ \text{"}\textit{sent from}\text{"}\ \textit{Location}\ ) \\
 & & |\ (\ \text{"}\{\text{"}\ (\textit{Identifier}\ \text{","?})+\ \text{"}\}\text{"}\ \text{"}\textit{apart-by}\text{"}\ \textit{SpatialSpec}\ ) \\
\textit{SpatialOp} & ::= & \text{"}\textit{south of}\text{"}\ |\ \text{"}\textit{north of}\text{"}\ |\ \text{"}\textit{east of}\text{"}\ |\ \text{"}\textit{west of}\text{"} \\
 & & |\ \text{"}\textit{south-east of}\text{"}\ |\ \text{"}\textit{south-west of}\text{"} \\
 & & |\ \text{"}\textit{north-east of}\text{"}\ |\ \text{"}\textit{north-west of}\text{"} \\
\textit{Location} & ::= & \textit{String} \\
 & & |\ (\ \text{"}\textit{the same}\text{"}\ (\text{"}\textit{room}\text{"}|\text{"}\textit{area}\text{"})\ ) \\
 & & |\ (\ \text{"}\textit{neighbour}\text{"}\ (\text{"}\textit{rooms}\text{"}|\text{"}\textit{areas}\text{"})\ ) \\
\textit{SpatialSpec} & ::= & (\ (\text{"}\textit{min}\text{"}|\text{"}\textit{max}\text{"})?\ \textit{Distance}\ ) \\
 & & |\ (\ \text{"}(\text{"}?\ \textit{SpacialSpec}\ \text{"}\wedge\text{"}\ \textit{SpacialSpec}\ \text{"})\text{"}?\ ) \\
 & & |\ (\ \text{"}(\text{"}?\ \textit{SpacialSpec}\ \text{"}\vee\text{"}\ \textit{SpacialSpec}\ \text{"})\text{"}?\ ) \\
\textit{Distance} & ::= & (\textit{Number}\ \text{"}\textit{km}\text{"})?\ (\textit{Number}\ \text{"}\textit{m}\text{"})? \\
 & & (\textit{Number}\ \text{"}\textit{cm}\text{"})?\ (\textit{Number}\ \text{"}\textit{mm}\text{"})?
\end{array}
$$

7.4 Normalization of a Constraint Set

The grammar of ESCL presented in the previous section allows for formulating short and readable constraints by means of modularization, local definitions, and disjunction in the body of a constraint. However, these features are syntactic sugar and can be eliminated from each constraint set to facilitate its usage for semantic query optimization. This section is devoted to the normalization of a set of ESCL constraints.

Consider the constraint in Listing 7.10 which has already been explained in Section 7.2.6. The normalization of a constraint set is illustrated by this example in the following.

Listing 7.10: The data constraint expressing that a price for an item increases during its sell.

```
1 WHILE state: itemOffer( auctionID(A), itemID(I) )
2 LET
3     LET
```

```
4       DETECT
5           event: e( auctionID(A), itemID(I), bidderID(B), value(V) )
6       ON
7           event: bid( auctionID(A), itemID(I), bidderID(B), value(V) ) ∨
8           event: hammerBeat( auctionID(A), itemID(I), bidderID(B), value(V) )
9       END
10   IN
11      IF event: itemDescription( auctionID(A), itemID(I), bidderID(B), value(V1) ) ∧
12         ( event: e( auctionID(A), itemID(I), bidderID(B), value(V2) ) ∨
13           event: sell( auctionID(A), itemID(I), bidderID(B), value(V2) ) )
14      THEN V1 ≤ V2
15      END
16   END
17 END
```

The normalization rules of ESCL constraint sets are the following:

1. Each local definition is eliminated by replacing the head of the deductive rule by its body in each respective ESCL constraint.

```
1 WHILE state: itemOffer( auctionID(A), itemID(I) )
2 LET
3    IF event: itemDescription( auctionID(A), itemID(I), bidderID(B), value(V1) ) ∧
4     ( event: bid( auctionID(A), itemID(I), bidderID(B), value(V2) ) ∨
5       event: hammerBeat( auctionID(A), itemID(I), bidderID(B), value(V2) ) ∨
6       event: sell( auctionID(A), itemID(I), bidderID(B), value(V2) ) )
7    THEN V1 ≤ V2
8    END
9 END
```

2. Each disjunction in a constraint body is eliminated by replacing the constraint by an equivalent conjunction of constraints with no disjunction in their bodies.

```
1 WHILE state: itemOffer( auctionID(A), itemID(I) )
2 LET
3    IF event: itemDescription( auctionID(A), itemID(I), bidderID(B), value(V1) ) ∧
4       event: bid( auctionID(A), itemID(I), bidderID(B), value(V2) )
5    THEN V1 ≤ V2
6    END
7    ∧
8    IF event: itemDescription( auctionID(A), itemID(I), bidderID(B), value(V1) ) ∧
9       event: hammerBeat( auctionID(A), itemID(I), bidderID(B), value(V2) )
10   THEN V1 ≤ V2
11   END
12   ∧
13   IF event: itemDescription( auctionID(A), itemID(I), bidderID(B), value(V1) ) ∧
```

```
14        event: sell( auctionID(A), itemID(I), bidderID(B), value(V2) )
15    THEN V1 ≤ V2
16    END
17 END
```

3. Each module is eliminated by making the default literal evaluation time explicit for each literal.

```
1  IF state s: itemOffer( auctionID(A), itemID(I) ) ∧
2     while s: event:
3        itemDescription( auctionID(A), itemID(I), bidderID(B), value(V1) ) ∧
4     while s: event: bid( auctionID(A), itemID(I), bidderID(B), value(V2) )
5  THEN V1 ≤ V2
6  END
7  ∧
8  IF state s: itemOffer( auctionID(A), itemID(I) ) ∧
9     while s: event:
10       itemDescription( auctionID(A), itemID(I), bidderID(B), value(V1) ) ∧
11    while s: event: hammerBeat( auctionID(A), itemID(I), bidderID(B), value(V2) )
12 THEN V1 ≤ V2
13 END
14 ∧
15 IF state s: itemOffer( auctionID(A), itemID(I) ) ∧
16    while s: event:
17       itemDescription( auctionID(A), itemID(I), bidderID(B), value(V1) ) ∧
18    while s: event: sell( auctionID(A), itemID(I), bidderID(B), value(V2) )
19 THEN V1 ≤ V2
20 END
```

4. All constraints are brought into a clausal notation.

```
1  V1 ≤ V2 ←
2     state s: itemOffer( auctionID(A), itemID(I) ) ∧
3     while s: event:
4        itemDescription( auctionID(A), itemID(I), bidderID(B), value(V1) ) ∧
5     while s: event: bid( auctionID(A), itemID(I), bidderID(B), value(V2) )
6  ∧
7  V1 ≤ V2 ←
8     state s: itemOffer( auctionID(A), itemID(I) ) ∧
9     while s: event:
10       itemDescription( auctionID(A), itemID(I), bidderID(B), value(V1) ) ∧
11    while s: event: hammerBeat( auctionID(A), itemID(I), bidderID(B), value(V2) )
12 ∧
13 V1 ≤ V2 ←
14    state s: itemOffer( auctionID(A), itemID(I) ) ∧
15    while s: event:
```

```
16        itemDescription( auctionID(A), itemID(I), bidderID(B), value(V1) ) ∧
17    while s: event: sell( auctionID(A), itemID(I), bidderID(B), value(V2) )
```

5. Each disjunction of constraints in a constraint set (consider an example in Listing 7.1) is eliminated by replacing this constraint set by an equivalent set of constraint sets with no disjunction of constraints.

Declarative semantics of the language is defined for normalized sets of ESCL constraints in Section 7.5.

7.5 Declarative Semantics

This section is devoted to the declarative semantics of ESCL. In Section 7.5.1, we start with the basic notions and notation used throughout this section. Like the specification of the grammar of the language, the definition of its declarative semantics is also divided into four modules. Section 7.5.2 is devoted to the formal semantics of the core of the language. Sections 7.5.3, 7.5.4, and 7.5.5 specify the formal semantics of the ESCL data, temporal, and spatial constraints respectively.

7.5.1 Basic Notions and Notation

Definition 28 (**Interpretation**). An interpretation of an ESCL constraint is a 3-tuple $I = (D, \sigma, \tau)$ where:

1. D is a set of data involving events $Events$ arriving on the stream (Section 4.2), application states $States$, and tuples $Tuples$ saved in a conventional database, i.e. $D = Events \dot{\cup} States \dot{\cup} Tuples$
2. σ is a grounding substitution for data variables (as defined in [29]), i.e. a substitution the application of which to a non ground atom returns a ground atom
3. $\tau : Identifiers \rightarrow Events \dot{\cup} States$ is a substitution for event and state identifiers $Identifiers$, i.e. a mapping of the identifiers to events or states.

Definition 29 (**Entailment of a Constraint Set in an Interpretation**). Let C be a (normalized) set of ESCL constraints and I be an interpretation. $I \models C$ iff $\forall c \in C\ \forall t \in \mathbb{TI}\ \exists \sigma, \tau$ such that $I = D, \sigma, \tau \models c^t$.

As mentioned above, the definition of $D, \sigma, \tau \models c^t$ consists of the following cases:

- ESCL core (Section 7.5.2):
  - Base cases
  - Recursive cases
- Modules:
  - Data constraints (Section 7.5.3)
  - Temporal constraints (Section 7.5.4)
  - Spatial constraints (Section 7.5.5)

But before we start the definition of the formal semantics of ESCL, let us consider the notation used for it:

$Events$	is the set of events	
$States$	is the set of states	
S	$\subseteq Events \dot\cup States$ is the set of events and states	
$Tuples$	is the set of database tuples	
D	$\subseteq S \dot\cup Tuples$ is the entire set of data	
now	$\in \mathbb{T}$ is the current time point	
T	$\subseteq \mathbb{TI}$	
$t, t', t_1, ..., t_n$	$\in \mathbb{TI}$	
$b(t)$	$\in \mathbb{T}$ is the beginning of t	
$e(t)$	$\in \mathbb{T}$ is the end of t	
$\lvert t \rvert$	$= e(t) - b(t) \in Duration$ is the duration of t	
$t' \sqsubseteq t$	$\equiv b(t) \leq b(t')$ and $e(t') \leq e(t)$	
$t = t_1 \sqcup t_2$	$\equiv b(t) := min(b(t_1), b(t_2))$ and $e(t) := max(e(t_1), e(t_2))$	
$t = t_1 \cup t_2$	$\equiv \begin{cases} b(t) := min(b(t_1), b(t_2)) \text{ and } e(t) := max(e(t_1), e(t_2)) & \text{if } b(t_1) \leq b(t_2) \leq e(t_1) \\ & \text{or } b(t_2) \leq b(t_1) \leq e(t_2) \\ \emptyset & \text{otherwise} \end{cases}$	
$t = t_1 \cap t_2$	$\equiv \begin{cases} b(t) := max(b(t_1), b(t_2)) \text{ and } e(t) := min(e(t_1), e(t_2)) & \text{if } b(t_1) \leq b(t_2) \leq e(t_1) \\ & \text{or } b(t_2) \leq b(t_1) \leq e(t_2) \\ \emptyset & \text{otherwise} \end{cases}$	
$i, j, i_1, ..., i_n$	$\in Identifiers$	
$b(i)$	$\in \mathbb{T}$ is the beginning of the occurrence (or the validity) time of i	
$e(i)$	$\in \mathbb{T}$ is the end of the occurrence (or the validity) time of i	
$location(i)$	$\in Location$ is the location i was sent from	
k	$\in \mathbb{N}_0$	
d, d'	$\in Duration$	
w	$\in Distance$	
Θ	$= \{=, \neq, <, \leq, >, \geq, min, max, \ \}$	
X, Y	$\in Exp \dot\cup TempExp$	
b	$\in \mathbb{N}_0 \dot\cup Variables$	
$B_1, ..., B_n$	$\in \Theta \times (\mathbb{N}_0 \cup Variable \cup Duration \cup Distance)$	
a, a_x	$\in GroundAtoms$	
$q, q_1, ..., q_n$	$\in NongroundAtoms$	
$Q, Q_1, ..., Q_n$	$\in AtomicQuery$	
H, B	$\in Query$ are the head and the body of a constraint	

7.5.2 The Core of the Language

The declarative semantics of the core of ESCL consists of:

- Base cases:

$$
\begin{array}{ll}
D, \sigma, \tau \models q & \text{iff } \exists a \in Tupels \text{ with } q\sigma = a \\
D, \sigma, \tau \models q^T & \text{iff } \exists a^{t'} \in S \text{ with } q\sigma = a \text{ and } \exists t \in T \text{ with } t' \sqsubseteq t \\
D, \sigma, \tau \models (i : q)^T & \text{iff } \exists a^{t'} \in S \text{ with } i\tau = a^{t'},\ q\sigma = a \text{ and } \exists t \in T \text{ with } t' \sqsubseteq t \\
D, \sigma, \tau \models (\exists_{\Theta b} q)^T & \text{iff } \exists_{\Theta b\sigma} a^{t'} \in S \text{ with } q\sigma = a \text{ and } \exists t \in T \text{ with } t' \sqsubseteq t \\
D, \sigma, \tau \models (\exists_{\Theta b} i)^T & \text{iff } \exists_{\Theta b\sigma} a^{t'} \in S \text{ with } i\tau = a^{t'} \text{ and } \exists t \in T \text{ with } t' \sqsubseteq t \\
D, \sigma, \tau \models (\exists_{\Theta b} i : q)^T & \text{iff } \exists_{\Theta b\sigma} a^{t'} \in S \text{ with } q\sigma = a,\ i\tau = a^{t'} \text{ and } \exists t \in T \text{ with } t' \sqsubseteq t
\end{array}
$$

Literal validity time can be a set of time intervals T because of the interval combination operator *cover-each* the semantics of which is defined in Section 7.5.4. If literal validity time is a window, a (relative timer) event or state, or a combination of time intervals by means of the interval combination operators *each*, *any*, *intersection*, *union*, *cover-all*, then T contains only one time interval.

The semantics of relative timer events or states is adopted from [50]:

$$
\begin{aligned}
&D,\sigma,\tau \models (i : \textit{extend-begin}[j,d])^t && \text{iff } \exists a^{t'} \in \textit{Events} \text{ with } j\tau = a^{t'},\ i\tau = a^t,\\
&&& \quad b(t) = b(t') - d,\ e(t) = e(t')\\
&D,\sigma,\tau \models (i : \textit{shorten-begin}[j,d])^t && \text{iff } \exists a^{t'} \in S \text{ with } j\tau = a^{t'},\ i\tau = a^t,\\
&&& \quad b(t) = b(t') + d,\ e(t) = e(t')\\
&D,\sigma,\tau \models (i : \textit{extend-end}[j,d])^t && \text{iff } \exists a^{t'} \in S \text{ with } j\tau = a^{t'},\ i\tau = a^t,\\
&&& \quad b(t) = b(t'),\ e(t) = e(t') + d\\
&D,\sigma,\tau \models (i : \textit{shorten-end}[j,d])^t && \text{iff } \exists a^{t'} \in \textit{Events} \text{ with } j\tau = a^{t'},\ i\tau = a^t,\\
&&& \quad b(t) = b(t'),\ e(t) = e(t') - d\\
&D,\sigma,\tau \models (i : \textit{shift-forward}[j,d])^t && \text{iff } \exists a^{t'} \in S \text{ with } j\tau = a^{t'},\ i\tau = a^t,\\
&&& \quad b(t) = b(t') + d,\ e(t) = e(t') + d\\
&D,\sigma,\tau \models (i : \textit{shift-backward}[j,d])^t && \text{iff } \exists a^{t'} \in \textit{Events} \text{ with } j\tau = a^{t'},\ i\tau = a^t,\\
&&& \quad b(t) = b(t') - d,\ e(t) = e(t') - d\\
&D,\sigma,\tau \models (i : \textit{from-end}[j,d])^t && \text{iff } \exists a^{t'} \in S \text{ with } j\tau = a^{t'},\ i\tau = a^t,\\
&&& \quad b(t) = e(t'),\ e(t) = e(t') + d\\
&D,\sigma,\tau \models (i : \textit{from-begin}[j,d])^t && \text{iff } \exists a^{t'} \in S \text{ with } j\tau = a^{t'},\ i\tau = a^t,\\
&&& \quad b(t) = b(t'),\ e(t) = b(t') + d\\
&D,\sigma,\tau \models (i : \textit{from-end-backward}[j,d])^t && \text{iff } \exists a^{t'} \in \textit{Events} \text{ with } j\tau = a^{t'},\ i\tau = a^t,\\
&&& \quad b(t) = e(t') - d,\ e(t) = e(t')\\
&D,\sigma,\tau \models (i : \textit{from-begin-backward}[j,d])^t && \text{iff } \exists a^{t'} \in \textit{Events} \text{ with } j\tau = a^{t'},\ i\tau = a^t,\\
&&& \quad b(t) = b(t') - d,\ e(t) = b(t')
\end{aligned}
$$

- Recursive cases:

$$
\begin{aligned}
&D,\sigma,\tau \models (\neg Q)^T && \text{iff } \forall t \in T\ \nexists t' \sqsubseteq t \text{ with } D,\sigma,\tau \models Q^{\{t'\}}\\
&D,\sigma,\tau \models (\exists_{B_1 \wedge \cdots \wedge B_n} Q)^T && \text{iff } \forall B_x \in \{B_1, ..., B_n\}.\ D,\sigma,\tau \models (\exists_{B_x} Q)^T\\
&D,\sigma,\tau \models (\exists_{B_1 \vee \cdots \vee B_n} Q)^T && \text{iff } \exists B_x \in \{B_1, ..., B_n\} \text{ with } D,\sigma,\tau \models (\exists_{B_x} Q)^T\\
&D,\sigma,\tau \models (Q_1 \wedge \cdots \wedge Q_n)^t && \text{iff } \forall Q_x \in \{Q_1, ..., Q_n\}.\ D,\sigma,\tau \models Q_x^t\\
&D,\sigma,\tau \models (Q_1 \vee \cdots \vee Q_n)^t && \text{iff } \exists Q_x \in \{Q_1, ..., Q_n\} \text{ with } D,\sigma,\tau \models Q_x^t\\
&D,\sigma,\tau \models (H \leftarrow)^t && \text{iff } D,\sigma,\tau \models H^t\\
&D,\sigma,\tau \models (H \leftarrow B)^t && \text{iff } \nexists t' \sqsubseteq t \text{ with } D,\sigma,\tau \models B^{t'}\\
&&& \text{or } D,\sigma,\tau \models H^t
\end{aligned}
$$

7.5.3 Data Constraints

$$D, \sigma, \tau \models X = Y \text{ iff } X\sigma = Y\sigma$$
$$D, \sigma, \tau \models X \neq Y \text{ iff } X\sigma \neq Y\sigma$$
$$D, \sigma, \tau \models X < Y \text{ iff } X\sigma < Y\sigma$$
$$D, \sigma, \tau \models X \leq Y \text{ iff } X\sigma \leq Y\sigma$$
$$D, \sigma, \tau \models X > Y \text{ iff } X\sigma > Y\sigma$$
$$D, \sigma, \tau \models X \geq Y \text{ iff } X\sigma \geq Y\sigma$$

If $X, Y \in \mathit{TempExp}$ are temporal expressions for formulating constraints on the beginning and the end of time intervals, their semantics is defined analogously to that of data expressions but the application of σ to X and Y is not necessary since the temporal expressions X and Y may not contain data variables.

7.5.4 Temporal Constraints

The declarative semantics of the ESCL temporal constraints consists of the following cases:

- Validity time is a (relative timer) event or state:

$$D, \sigma, \tau \models (\mathit{while}\ i :\ Q)^t \text{ iff } \exists a^{t'} \in S \text{ with } i\tau = a^{t'},\ t = t',\ D, \sigma, \tau \models Q^{\{t'\}}$$

- Validity time is a window:

$$
\begin{aligned}
&D, \sigma, \tau \models (\mathit{while\ unbounded}:\ Q)^t && \text{iff } t = [0, \infty),\ D, \sigma, \tau \models Q^{\{t\}} \\
&D, \sigma, \tau \models (\mathit{while\ now}:\ Q)^t && \text{iff } t = \mathit{now},\ D, \sigma, \tau \models Q^{\{\mathit{now}\}} \\
&D, \sigma, \tau \models (\mathit{while}\ t':\ Q)^t && \text{iff } t = t',\ D, \sigma, \tau \models Q^{\{t'\}} \\
&D, \sigma, \tau \models (\mathit{while\ Range}\ d,\ \mathit{Slide}\ d':\ Q)^t && \text{iff } \exists t_1, ..., t_n \text{ with } |t_1| = ... = |t_n| = d, \\
& && \quad \forall t_x, t_y \in \{t_1, ..., t_n\} \text{ with } e(t_x) \leq b(t_y) \\
& && \quad\quad \nexists t_z \in \{t_1, ..., t_n\} \text{ with} \\
& && \quad\quad\quad e(t_x) \leq b(t_z) \leq b(t_y). \\
& && \quad\quad b(t_y) = e(t_x) + d' \\
& && \text{and } \exists t' \in \{t_1, ..., t_n\} \text{ with } t = t',\ D, \sigma, \tau \models Q^{\{t'\}} \\
&D, \sigma, \tau \models (\mathit{while\ Range}\ d:\ Q)^t && \text{iff } \exists t_1, ..., t_n \text{ with } |t_1| = ... = |t_n| = d, \\
& && \quad \forall t_x, t_y \in \{t_1, ..., t_n\} \text{ with } e(t_x) \leq b(t_y) \\
& && \quad\quad \nexists t_z \in \{t_1, ..., t_n\} \text{ with} \\
& && \quad\quad\quad e(t_x) \leq b(t_z) \leq b(t_y). \\
& && \quad\quad b(t_y) = e(t_x) + d \\
& && \text{and } \exists t' \in \{t_1, ..., t_n\} \text{ with } t = t',\ D, \sigma, \tau \models Q^{\{t'\}}
\end{aligned}
$$

- Validity time is a (set of) time interval(s) specified by means of the interval combination operator:

$D, \sigma, \tau \models (while\ each\{i_1, ..., i_n\} :\ Q)^t$ iff $\exists t_1, ..., t_n$ with $t = t_1 \sqcup ... \sqcup t_n$ and
$\forall i_x \in \{i_1, ..., i_n\}.\ \exists a_x^{t_x} \in S$ with
$i_x\tau = a_x^{t_x},\ t_x \in \{t_1, ..., t_n\}$ and
$D, \sigma, \tau \models Q^{\{t_x\}}$

$D, \sigma, \tau \models (while\ any\{i_1, ..., i_n\} :\ Q)^t$ iff $\exists t_1, ..., t_n$ with $t = t_1 \sqcup ... \sqcup t_n$ and
$\exists i_x \in \{i_1, ..., i_n\}$ with $\exists a_x^{t_x} \in S$ with
$i_x\tau = a_x^{t_x},\ t_x \in \{t_1, ..., t_n\}$ and
$D, \sigma, \tau \models Q^{\{t_x\}}$

$D, \sigma, \tau \models (while\ intersection\{i_1, ..., i_n\} :\ Q)^t$ iff $\exists t_1, ..., t_n$ with $t = t_1 \sqcup ... \sqcup t_n$ and
$\forall i_x \in \{i_1, ..., i_n\}.\ \exists a_x^{t_x} \in S$ with
$i_x\tau = a_x^{t_x},\ t_x \in \{t_1, ..., t_n\}$ and
$D, \sigma, \tau \models Q^{\{t_1 \cap ... \cap t_n\}}$

$D, \sigma, \tau \models (while\ union\{i_1, ..., i_n\} :\ Q)^t$ iff $\exists t_1, ..., t_n$ with $t = t_1 \sqcup ... \sqcup t_n$ and
$\forall i_x \in \{i_1, ..., i_n\}.\ \exists a_x^{t_x} \in Events$ with
$i_x\tau = a_x^{t_x},\ t_x \in \{t_1, ..., t_n\}$ and
$D, \sigma, \tau \models Q^{\{t_1 \cup ... \cup t_n\}}$

$D, \sigma, \tau \models (while\ cover\text{-}each\{i_1, ..., i_n\} :\ Q)^t$ iff $\exists t_1, ..., t_n$ with $t = t_1 \sqcup ... \sqcup t_n$ and
$\forall i_x \in \{i_1, ..., i_n\}.\ \exists a_x^{t_x} \in S$ with
$i_x\tau = a_x^{t_x},\ t_x \in \{t_1, ..., t_n\}$ and
$D, \sigma, \tau \models Q^{\{t_1, ..., t_n\}}$

$D, \sigma, \tau \models (while\ cover\text{-}all\{i_1, ..., i_n\} :\ Q)^t$ iff $\exists t_1, ..., t_n$ with $t = t_1 \sqcup ... \sqcup t_n$ and
$\forall i_x \in \{i_1, ..., i_n\}.\ \exists a_x^{t_x} \in S$ with
$i_x\tau = a_x^{t_x},\ t_x \in \{t_1, ..., t_n\}$ and
$D, \sigma, \tau \models Q^{\{t\}}$

If the time intervals of $i_1, ..., i_n$ are combined by means of *union* then all $i_1, ..., i_n$ must be event identifiers, they may not reference states. If $i_1, ..., i_n$ were state identifiers, the union of their respective time intervals can be computed as soon as all states referenced by $i_1, ..., i_n$ are over which contradicts the nature of states since states can be processed as soon as they start even if their end is unknown.

- Allen's temporal relations adapted from [3]:

$$
\begin{aligned}
&D, \sigma, \tau \models i\ before\ j && \text{iff } e(i) < b(j) \\
&D, \sigma, \tau \models i\ after\ j && \text{iff } e(j) < b(i) \\
&D, \sigma, \tau \models i\ during\ j && \text{iff } b(j) < b(i),\ e(i) < e(j) \\
&D, \sigma, \tau \models i\ contains\ j && \text{iff } b(i) < b(j),\ e(j) < e(i) \\
&D, \sigma, \tau \models i\ equals\ j && \text{iff } b(i) = b(j),\ e(i) = e(j) \\
&D, \sigma, \tau \models i\ overlaps\ j && \text{iff } b(j) < b(i) < e(j) < e(i) \\
&D, \sigma, \tau \models i\ overlapped\text{-}by\ j && \text{iff } b(i) < b(j) < e(i) < e(j) \\
&D, \sigma, \tau \models i\ meets\ j && \text{iff } e(i) = b(j) \\
&D, \sigma, \tau \models i\ met\text{-}by\ j && \text{iff } e(j) = b(i) \\
&D, \sigma, \tau \models i\ starts\ j && \text{iff } b(i) = b(j),\ e(i) < e(j) \\
&D, \sigma, \tau \models i\ started\text{-}by\ j && \text{iff } b(j) = b(i),\ e(j) < e(i) \\
&D, \sigma, \tau \models i\ finishes\ j && \text{iff } b(j) < b(i),\ e(i) = e(j) \\
&D, \sigma, \tau \models i\ finished\text{-}by\ j && \text{iff } b(i) < b(j),\ e(j) = e(i)
\end{aligned}
$$

- Other temporal constraints:

$$
\begin{aligned}
&D, \sigma, \tau \models \{i_1, ..., i_n\}\ within\ d && \text{iff } e - b = d \text{ with} \\
&&& \quad e := max\{e(i_1), ..., e(i_n)\}, \\
&&& \quad b := min\{b(i_1), ..., b(i_n)\} \\
&D, \sigma, \tau \models \{i_1, ..., i_n\}\ within\ min\ d && \text{iff } e - b \geq d \text{ with} \\
&&& \quad e := max\{e(i_1), ..., e(i_n)\}, \\
&&& \quad b := min\{b(i_1), ..., b(i_n)\} \\
&D, \sigma, \tau \models \{i_1, ..., i_n\}\ within\ max\ d && \text{iff } e - b \leq d \text{ with} \\
&&& \quad e := max\{e(i_1), ..., e(i_n)\}, \\
&&& \quad b := min\{b(i_1), ..., b(i_n)\} \\
&D, \sigma, \tau \models \{i_1, ..., i_n\}\ within\ B_1 \wedge \cdots \wedge B_n && \text{iff } \forall B_x \in \{B_1, ..., B_n\}. \\
&&& \quad D, \sigma, \tau \models \{i_1, ..., i_n\}\ within\ B_x \\
&D, \sigma, \tau \models \{i_1, ..., i_n\}\ within\ B_1 \vee \cdots \vee B_n && \text{iff } \exists B_x \in \{B_1, ..., B_n\} \text{ with} \\
&&& \quad D, \sigma, \tau \models \{i_1, ..., i_n\}\ within\ B_x
\end{aligned}
$$

$$D, \sigma, \tau \models \{i_1, ..., i_n\} \textit{ apart-by } d \quad \text{iff } \forall i_x, i_y \in \{i_1, ..., i_n\} \text{ with } e(i_x) \leq b(i_y) \nexists i_z \in \{i_1, ..., i_n\} \text{ with } e(i_x) \leq b(i_z),\ e(i_z) \leq b(i_y). \; b(i_y) - e(i_x) = d$$

$$D, \sigma, \tau \models \{i_1, ..., i_n\} \textit{ apart-by min } d \text{ iff } \forall i_x, i_y \in \{i_1, ..., i_n\} \text{ with } e(i_x) \leq b(i_y) \nexists i_z \in \{i_1, ..., i_n\} \text{ with } e(i_x) \leq b(i_z),\ e(i_z) \leq b(i_y). \; b(i_y) - e(i_x) \geq d$$

$$D, \sigma, \tau \models \{i_1, ..., i_n\} \textit{ apart-by max } d \text{ iff } \forall i_x, i_y \in \{i_1, ..., i_n\} \text{ with } e(i_x) \leq b(i_y) \nexists i_z \in \{i_1, ..., i_n\} \text{ with } e(i_x) \leq b(i_z),\ e(i_z) \leq b(i_y). \; b(i_y) - e(i_x) \leq d$$

$$D, \sigma, \tau \models \{i_1, ..., i_n\} \textit{ apart-by } B_1 \wedge \cdots \wedge B_n \text{ iff } \forall B_x \in \{B_1, ..., B_n\}. \; D, \sigma, \tau \models \{i_1, ..., i_n\} \textit{ apart-by } B_x$$

$$D, \sigma, \tau \models \{i_1, ..., i_n\} \textit{ apart-by } B_1 \vee \cdots \vee B_n \text{ iff } \exists B_x \in \{B_1, ..., B_n\} \text{ with } D, \sigma, \tau \models \{i_1, ..., i_n\} \textit{ apart-by } B_x$$

$$D, \sigma, \tau \models \{i_1, ..., i_n\} \textit{ apart-by no } Q \quad \text{iff } \forall i_x, i_y \in \{i_1, ..., i_n\} \text{ with } e(i_x) \leq b(i_y) \nexists i_z \in \{i_1, ..., i_n\} \text{ with } e(i_x) \leq b(i_z),\ e(i_z) \leq b(i_y). \; \nexists t \text{ with } b(t) = e(i_x),\ e(t) = b(i_y),\ D, \sigma, \tau \models Q^t$$

$$D, \sigma, \tau \models \{i_1, ..., i_n\} \textit{ apart-by } k\ Q \quad \text{iff } \forall i_x, i_y \in \{i_1, ..., i_n\} \text{ with } e(i_x) \leq b(i_y) \nexists i_z \in \{i_1, ..., i_n\} \text{ with } e(i_x) \leq b(i_z),\ e(i_z) \leq b(i_y). \; \exists t \text{ with } b(t) = e(i_x),\ e(t) = b(i_y),\ D, \sigma, \tau \models (\exists_{=k} Q)^t$$

$$D, \sigma, \tau \models \{i_1, ..., i_n\} \textit{ apart-by min } k\ Q \text{ iff } \forall i_x, i_y \in \{i_1, ..., i_n\} \text{ with } e(i_x) \leq b(i_y) \nexists i_z \in \{i_1, ..., i_n\} \text{ with } e(i_x) \leq b(i_z),\ e(i_z) \leq b(i_y). \; \exists t \text{ with } b(t) = e(i_x),\ e(t) = b(i_y),\ D, \sigma, \tau \models (\exists_{\geq k} Q)^t$$

$$D, \sigma, \tau \models \{i_1, ..., i_n\} \textit{ apart-by max } k\ Q \text{ iff } \forall i_x, i_y \in \{i_1, ..., i_n\} \text{ with } e(i_x) \leq b(i_y) \nexists i_z \in \{i_1, ..., i_n\} \text{ with } e(i_x) \leq b(i_z),\ e(i_z) \leq b(i_y). \; \exists t \text{ with } b(t) = e(i_x),\ e(t) = b(i_y),\ D, \sigma, \tau \models (\exists_{\leq k} Q)^t$$

$D, \sigma, \tau \models \{i_1, ..., i_n\}$ *apart-by* $B_1 \wedge \cdots \wedge B_n$ Q iff $\forall B_x \in \{B_1, ..., B_n\}$.
$D, \sigma, \tau \models \{i_1, ..., i_n\}$ *apart-by* B_x Q

$D, \sigma, \tau \models \{i_1, ..., i_n\}$ *apart-by* $B_1 \vee \cdots \vee B_n$ Q iff $\exists B_x \in \{B_1, ..., B_n\}$ with
$D, \sigma, \tau \models \{i_1, ..., i_n\}$ *apart-by* B_x Q

7.5.5 Spatial Constraints

$D, \sigma, \tau \models i$ *south of* j iff $location(i)$ is south of $location(j)$

$D, \sigma, \tau \models i$ *north of* j iff $location(i)$ is north of $location(j)$

$D, \sigma, \tau \models i$ *east of* j iff $location(i)$ is east of $location(j)$

$D, \sigma, \tau \models i$ *west of* j iff $location(i)$ is west of $location(j)$

$D, \sigma, \tau \models i$ *south-east of* j iff $location(i)$ is south-east of $location(j)$

$D, \sigma, \tau \models i$ *south-west of* j iff $location(i)$ is south-west of $location(j)$

$D, \sigma, \tau \models i$ *north-east of* j iff $location(i)$ is north-east of $location(j)$

$D, \sigma, \tau \models i$ *north-west of* j iff $location(i)$ is north-west of $location(j)$

$D, \sigma, \tau \models \{i_1, ..., i_n\}$ *sent from* l iff $\forall i_x \in \{i_1, ..., i_n\}$. $location(i_x) = l$

$D, \sigma, \tau \models \{i_1, ..., i_n\}$ *sent from the same room* iff $location(i_1), ..., location(i_n)$ are in the same room

$D, \sigma, \tau \models \{i_1, ..., i_n\}$ *sent from the same area* iff $location(i_1), ..., location(i_n)$ are in the same area

$D, \sigma, \tau \models \{i_1, ..., i_n\}$ *sent from neighbour rooms* iff $location(i_1), ..., location(i_n)$ are in neighbour rooms

$D, \sigma, \tau \models \{i_1, ..., i_n\}$ *sent from neighbour areas* iff $location(i_1), ..., location(i_n)$ are in neighbour areas

$D, \sigma, \tau \models \{i_1, ..., i_n\}$ *apart-by* w iff $\forall i_x, i_y \in \{i_1, ..., i_n\}$. $location(i_x)$ is exactly w away from $location(i_y)$

$D, \sigma, \tau \models \{i_1, ..., i_n\}$ *apart-by min* w iff $\forall i_x, i_y \in \{i_1, ..., i_n\}$. $location(i_x)$ is at least w away from $location(i_y)$

$D, \sigma, \tau \models \{i_1, ..., i_n\}$ *apart-by max* w iff $\forall i_x, i_y \in \{i_1, ..., i_n\}$. $location(i_x)$ is at most w away from $location(i_y)$

$$D, \sigma, \tau \models \{i_1, ..., i_n\} \textit{ apart-by } B_1 \wedge \cdots \wedge B_n \text{ iff } \forall B_x \in \{B_1, ..., B_n\}. \\ D, \sigma, \tau \models \{i_1, ..., i_n\} \textit{ apart-by } B_x$$

$$D, \sigma, \tau \models \{i_1, ..., i_n\} \textit{ apart-by } B_1 \vee \cdots \vee B_n \text{ iff } \exists B_x \in \{B_1, ..., B_n\} \text{ with} \\ D, \sigma, \tau \models \{i_1, ..., i_n\} \textit{ apart-by } B_x$$

Chapter 8

Propagation of ESCL Cardinality Constraints

In this chapter, the propagation of ESCL cardinality constraints is formally defined. The use of cardinality constraints for semantic query optimization is described in Section 8.1. The idea of constraint propagation is presented in Section 8.2. It is dependent neither from an event query language nor from a constraint language. To illustrate the approach in this section, ESCL cardinality constraints are propagated with respect to StreamLog queries. Section 8.3 presents the assumptions and Section 8.4 describes the challenges of the approach. Section 8.5 and Section 8.6 are devoted to the formal specification of the cardinality propagation. In some cases the derived cardinality constraints can be simplified. Section 8.7 defines the rules for the simplification of cardinality constraints. Propagation of ESCL causal, temporal, and data constraints is a subject of future work. Examples are given in Section 12.2.

8.1 Cardinality Constraints

ESCL cardinality constraints specify the number of events or states during time intervals. These constraints make the following semantic query optimization techniques possible:

1. *Recognition of query unsatisfiability.* Assume a query derives an event a if there is no event b during a time interval i. If the cardinality of b during i is not zero then the query is unsatisfiable and will not be evaluated.

2. *Query suspension* as long as the queried data is incomplete. CEP queries are usually satisfiable during particular time intervals or states, i.e. the cardinality of queried

events during these time intervals is either not null if the respective query is positive or null if the respective query is negative. If these time intervals or states have not begun or are already over the respective queries are not satisfiable and their evaluation can be suspended.

3. *Scan reduction*, i.e. the interruption of reading of buffer as soon as all relevant data has been found.

4. *Query termination* as soon as all relevant data has arrived. This is especially important for queries with grouping. If the number of events in a group is known then an aggregation function (e.g., *max* or *sum*) can be applied to the group as soon as the group is complete even if the time interval for the grouping is not over.

5. *Choice of join algorithms* depends on the number of joined events which is specified by cardinality constraints.

6. *Join introduction*, i.e. introduction of a join with a seldom event or state x into a query to suspend the query as long as x is not available and to reduce the amount of data which must be further processed.

These semantic optimization techniques (except for query suspension and query termination) are also applicable to database queries.

As motivated in Section 2.2.3, in order to semantically optimize queries which are based on (database or CEP) materialized views, constraints have to be defined for these views and not only for base database relations, events or states. To reduce the number of manually specified constraints, constraints for views can be automatically derived from the (manually defined) constraints for base database relations, events or states with respect to the queries defining the views. Consider Example 7.

Example 7.
If an event a is derived from two events b and c arriving during some time interval i and it is known that the events b and c are unique during i, then the derived event a is also unique during i.

Example 7 demonstrates an essential difference between the propagation of database constraints and the propagation of CEP constraints, namely database constraints are valid always in contrast to CEP constraints which are usually valid during particular time intervals (see Section 2.2.2 for more details). If the validity time of CEP constraints is not the same as the evaluation time of CEP queries (which is usually the case), the propagation of constraints is not that trivial. We are aware of existing approaches to this topic neither in database systems nor in CEP.

Without automatic propagation of CEP constraints, these constraints would have to be defined not only for each kind of queried events and states but also in each evaluation time interval of a CEP query for which these constraints are relevant.

For the purpose of step-wise inductive specification of CEP cardinality constraint propagation, we differentiate between constraints with simple and complex cardinality specifications.

Definition 30 (**Simple cardinality specification**)**.** A simple cardinality specification consists of a comparison operator $=$, $\leq$ or $\geq$ and a natural number $n \in \mathbb{N}_0$, both written in the index of the existentional quantifier.

For example, $i : \exists_{\geq 0}\ a \leftarrow$ is a constraint involving the simple cardinality specification ≥ 0. It means that there is an arbitrary number of events matching a during the time interval i. The constraint holds unconditionally, therefore its body is empty.

Each simple cardinality specification describes an interval of natural numbers. Let $n \in \mathbb{N}_0$. $= n$ corresponds to the interval $[n, n]$, $\leq n$ specifies the interval $[0, n]$, and $\geq n$ describes the interval $[n, \infty)$.

Note that simple cardinality specifications involving comparison operators $<$ and $>$ can be simulated by $\leq$ and $\geq$ respectively since $< n$ is equivalent to $\leq n - 1$ and $> n$ to $\geq n - 1$ where $n \in \mathbb{N}_0$.

Definition 31 (**Complex Cardinality Specification**)**.** A complex cardinality specification is a conjunction or a disjunction of an arbitrary number of simple or complex cardinality specifications in the index of the existentional quantifier.

$i : \exists_{(\geq 2 \wedge \leq 20) \vee (\geq 30 \wedge \leq 42) \vee (= 0 \vee \geq 6)}\ a \leftarrow$ is a constraint with the complex cardinality specification $(\geq 2 \wedge \leq 20) \vee (\geq 30 \wedge \leq 42) \vee (= 0 \vee \geq 6)$ of the number of events matching a during the time interval i. The constraint holds unconditionally, that is why its body is empty.

A disjunction of cardinality specifications defines the union of the intervals specified by the disjuncts. For example, $(= 0 \vee \geq 6)$ is equivalent to $[0, 0] \cup [6, \infty)$. A conjunction of cardinality specifications defines the intersection of the intervals specified by the conjuncts. For example, $(\geq 2 \wedge \leq 20)$ is corresponds to $[2, 20]$.

Note that a cardinality specification involving the comparison operator $\neq$ is also complex since, e.g. $\neq 3$ is equivalent to $(\leq 2 \vee \geq 4)$.

In order to exclude cardinality constraints which make no sense, the following definition formally specifies the validity of a cardinality constraint.

Definition 32 (**Valid cardinality specification**)**.** A simple cardinality specification is always valid. A disjunction of valid cardinality specifications is valid. A conjunction of valid cardinality specifications is valid if the intersection of the intervals specified by the conjuncts is not empty.

For example, $(\leq 2 \wedge \geq 20)$ is invalid since $[0, 2] \cap [20, \infty) = \emptyset$.

8.2 Cardinality Propagation Function

Let *Queries* be the set of StreamLog queries as defined in Chapter 5. Let $q \in Queries$ by a query of the form $i : h \leftarrow b$ where i is the evaluation time interval of q, h is the head and b is the body of q. Remember that b is an arbitrarily nested conjunction or disjunction of literals $l_1, \ldots l_n$ and conditions on them. With the help of these conditions tighter evaluation time intervals can be specified for the literals than i, i.e. i is the default evaluation time for each literal of q. Let $i_1, \ldots, i_n$ be the evaluation time intervals of $l_1, \ldots, l_n$ respectively.

Let *Cardinalities* be the set of cardinality specifications (Definitions 30 and 31) of the cardinality constraints for the events or states matching atoms $m_1, \ldots, m_k$ such that $\forall l_x \in \{l_1, \ldots, l_n\}$ $\exists m_y \in \{m_1, \ldots, m_k\}$ $\exists$ a substitution θ (Definition 34) such that $m_y \supseteq_\theta l_x$ (Definition 38) with θ. Let $j_1, \ldots, j_k$ be the validity time intervals of these constraints respectively.

The **cardinality propagation function**:

$$\mathcal{C} : \mathit{Cardinalities} \rightarrow \mathit{Cardinalities}$$

takes the cardinality specifications of cardinality constraints for the events or states matching $m_1, \ldots, m_k$ as parameters and returns the cardinality specification of the constraint for h with the validity time i. This function will be formally defined in Section 8.5 and Section 8.6.

8.3 Assumptions

As the informal introduction of the cardinality propagation function $\mathcal{C}$ in Section 8.2 shows, the result of cardinality propagation (with respect to a query q) can be based on the cardinality of either more specific or more general atoms as the atoms appearing in the body of q. Consider the following example.

Example 8.
Assume there are exactly two events matching the atom $f(a)$ within some time interval i (i.e. $i : \exists_{=2}\ f(a) \leftarrow$). Assume there is an atom $f(X)$ in a query body such that the cardinality of $f(X)$ is unknown. Instead of using the default cardinality constraint $i : \exists_{\geq 0}\ f(X) \leftarrow$, one could infer $i : \exists_{\geq 2}\ f(X) \leftarrow$.

For simplicity reasons in the following, we assume that the function $\mathcal{C}$ takes the cardinality constraints for atoms appearing in the body of the respective query. But keep in mind that these constraints can be inferred as shown by the above example.

For simplicity reasons, we assume also that all cardinality constraints considered in the following are facts, i.e. their bodies are empty. But remember that cardinality constraint propagation is also possible when the constraints are rules as the following example demonstrates.

Example 9.
Consider the cardinality constraint $i : \exists_{=2}\ l_1 \leftarrow l_2$. The constraint means that if there is an event matching l_2 during the time interval i than the number of events matching l_1 during i is exactly 2. Consider the rule $i : h \leftarrow l_1 \wedge l_2$. The above cardinality constraint can be used for the constraint propagation with respect to this rule since l_2 is implied by the rule. But the above cardinality constraint cannot be used for the constraint propagation with respect to the rule $i : h \leftarrow l_1 \vee l_2$ in general. Except for the case if the number of events matching l_2 in i cannot be zero, i.e. l_2 is implied by another constraint.

For the sake of brevity, cardinality constraints involving comparison operators $<, >$ and $\neq$ are not considered in the following. They can be simulated by the comparison operators $\leq, \geq$ and $=$ as described in Section 8.1.

We assume all considered cardinality constraints to be valid (Definition 32) and simplified (Section 8.7).

Let q be a StreamLog query of the form $i : h \leftarrow b$. Negative literals in b do not have to be considered for the purpose of constraint propagation because of the following reason. Assume there is a negative literal $\neg l$ in b. If the cardinality of l in i is not 0, then q is unsatisfiable and the cardinality of h is 0. Otherwise (the cardinality of l in i is 0), the cardinality of h depends only on positive literals in b. Therefore only positive literals in a query body are considered in the following.

And finally, note that the number of all events is defined in all time intervals since in the worst case the default cardinality constraint $i : \exists_{\geq 0}\ l \leftarrow$ holds (i is a time interval and l is an atom specifying events). Therefore, the propagation of cardinality constraints

is possible even if the constraint set is incomplete.

8.4 Challenges

There are the following challenges of the formal definition of the cardinality propagation function:

1. The evaluation time of a literal in a query body does usually not coincide with the validity time interval of the cardinality constraint restricting the number of events (states) matching the literal. We start in Section 8.5 with the case in which both time intervals coincide and continue in Section 8.6 with the case in which they do not coincide.

2. The number of literals in a conjunction or a disjunction within a query body can be arbitrary. Section 8.5 and Section 8.6 are divided into two subsections. The first one of them defines the cardinality propagation function $\mathcal{C}$ with respect to a query the body of which is a conjunction or a disjunction of only two atoms. This is the base case of the function. The second subsection defines the inductive case of the function, i.e. cardinality propagation with respect to a query the body of which is an arbitrarily nested conjunction or disjunction of atoms.

3. The cardinality specifications within a cardinality constraint can be arbitrarily nested. We introduce the propagation of cardinality constraints with simple cardinality specifications (Definition 30) first and then concentrate our attention on constraints with complex cardinality specifications (Definition 31).

8.5 Constraint Validity Time Equals Query Evaluation Time

This section is devoted to the case in which the query evaluation time coincides with the validity time intervals of the constraints which are propagated with respect to the query. We start in Section 8.5.1 with the cardinality propagation with respect to a query with only two atoms in the body (this is the base case of the propagation) and continue in Section 8.5.2 with the cardinality propagation with respect to a query with an arbitrary number of atoms in the body (this is the inductive case of the propagation). In both cases, constraints with simple and complex cardinality specifications are considered.

8.5.1 Two Atoms in a Query Body

Simple Cardinality Specifications

c_B \ c_A	$= 0$	$= c$	$\leq a$	$\geq a$
$= 0$	$= 0$	$= 0$	$= 0$	$= 0$
$= d$	$= 0$	$= cd$	$\leq ad$	$\geq ad$
$\leq b$	$= 0$	$\leq cb$	$\leq ab$	$= 0 \vee \geq a$
$\geq b$	$= 0$	$\geq cb$	$= 0 \vee \geq b$	$\geq ab$

Table 8.1: $\mathcal{C}_\wedge(c_A, c_B)$ defines the first base case of the cardinality propagation function $\mathcal{C}$: The query body is a conjunction of two atoms the cardinality specifications of which are simple. $a, b \in \mathbb{N}_0,\ c, d \in \mathbb{N}$.

Let $i : h \leftarrow A \wedge B$ be a StreamLog query where A, B and h are atoms and i is a time interval. Let c_A and c_B be simple cardinality specifications in the cardinality constraints for A and B respectively with the validity time i. Table 8.1 contains the cardinality specification $\mathcal{C}_\wedge(c_A, c_B)$ of the cardinality constraint for h with the validity time i.

Example 10.
If $c_A =\leq 5$ and $c_B == 3$ then $\mathcal{C}_\wedge(c_A, c_B) =\leq 15$.

c_B \ c_A	$= a$	$\leq a$	$\geq a$
$= b$	$= a + b$	$\geq b \wedge \leq a + b$	$\geq a + b$
$\leq b$	$\geq a \wedge \leq a + b$	$\leq a + b$	$\geq a$
$\geq b$	$\geq a + b$	$\geq b$	$\geq a + b$

Table 8.2: $\mathcal{C}_\vee(c_A, c_B)$ defines the second base case of the cardinality propagation function $\mathcal{C}$: The query body is a disjunction of two atoms the cardinality specifications of which are simple. $a, b \in \mathbb{N}_0$.

Let $i : h \leftarrow A \vee B$ be a StreamLog query where A, B and h are atoms and i is a time interval. Let c_A and c_B be simple cardinality specifications in the cardinality constraints for A and B respectively with the validity time i. Table 8.2 contains the cardinality specification $\mathcal{C}_\vee(c_A, c_B)$ of the cardinality constraint for h with the validity time i.

Example 11.
If $c_A =\leq 5$ and $c_B == 3$ then $\mathcal{C}_\vee(c_A, c_B) = (\geq 3 \wedge \leq 8)$.

All the following cases of cardinality constraint propagation are based upon these two base cases.

Complex Cardinality Specifications

c_B \ c_A	$c_{A_1} \vee \cdots \vee c_{A_n}$	$c_{A_1} \wedge \cdots \wedge c_{A_n}$
$c_{B_1} \vee \cdots \vee c_{B_m}$	$\mathcal{C}_\wedge(c_{A_1}, c_{B_1}) \vee \cdots \vee \mathcal{C}_\wedge(c_{A_n}, c_{B_1}) \vee$ $\vdots \quad \vdots \quad \vdots$ $\mathcal{C}_\wedge(c_{A_1}, c_{B_m}) \vee \cdots \vee \mathcal{C}_\wedge(c_{A_n}, c_{B_m})$	$(\ \mathcal{C}_\wedge(c_{A_1}, c_{B_1}) \wedge \cdots \wedge \mathcal{C}_\wedge(c_{A_n}, c_{B_1})\ ) \vee$ $\vdots \quad \vdots \quad \vdots \quad \vdots$ $(\ \mathcal{C}_\wedge(c_{A_1}, c_{B_m}) \wedge \cdots \wedge \mathcal{C}_\wedge(c_{A_n}, c_{B_m})\ )$
$c_{B_1} \wedge \cdots \wedge c_{B_m}$	$(\ \mathcal{C}_\wedge(c_{A_1}, c_{B_1}) \wedge \cdots \wedge \mathcal{C}_\wedge(c_{A_1}, c_{B_m})\ ) \vee$ $\vdots \quad \vdots \quad \vdots \quad \vdots$ $(\ \mathcal{C}_\wedge(c_{A_n}, c_{B_1}) \wedge \cdots \wedge \mathcal{C}_\wedge(c_{A_n}, c_{B_m})\ )$	$\mathcal{C}_\wedge(c_{A_1}, c_{B_1}) \wedge \cdots \wedge \mathcal{C}_\wedge(c_{A_n}, c_{B_1}) \wedge$ $\vdots \quad \vdots \quad \vdots$ $\mathcal{C}_\wedge(c_{A_1}, c_{B_m}) \wedge \cdots \wedge \mathcal{C}_\wedge(c_{A_n}, c_{B_m})$

Table 8.3: $\mathcal{C}_\wedge(c_A,\, c_B)$ defines the first inductive case of the cardinality propagation function $\mathcal{C}$: The query body is a conjunction of two atoms the cardinality specifications of which are complex.

Let $i : h \leftarrow A \wedge B$ be a StreamLog query where A, B and h are atoms and i is a time interval. Let c_A and c_B be complex cardinality specifications in the cardinality constraints for A and B respectively with the validity time i. Table 8.3 contains the cardinality specification $\mathcal{C}_\wedge(c_A,\, c_B)$ of the cardinality constraint for h with the validity time i.

Example 12.
If $c_A = (= 6 \vee \leq 3)$ and $c_B = (\geq 4 \wedge \leq 7)$ then
$\mathcal{C}_\wedge(c_A, c_B) = \mathcal{C}_\wedge(\ (= 6 \vee \leq 3),\ (\geq 4 \wedge \leq 7)\ ) =$
$(\ \mathcal{C}_\wedge(= 6, \geq 4) \wedge \mathcal{C}_\wedge(= 6, \leq 7)\ ) \vee (\ \mathcal{C}_\wedge(\leq 3, \geq 4) \wedge \mathcal{C}_\wedge(\leq 3, \leq 7)\ ) =$
$(\ (\geq 24 \wedge \leq 42) \vee ((= 0 \vee \geq 4) \wedge \leq 21)\ )$

c_B \ c_A	$c_{A_1} \vee \cdots \vee c_{A_n}$	$c_{A_1} \wedge \cdots \wedge c_{A_n}$
$c_{B_1} \vee \cdots \vee c_{B_m}$	$\mathcal{C}_\vee(c_{A_1}, c_{B_1}) \vee \cdots \vee \mathcal{C}_\vee(c_{A_n}, c_{B_1}) \vee$ $\vdots \quad \vdots \quad \vdots$ $\mathcal{C}_\vee(c_{A_1}, c_{B_m}) \vee \cdots \vee \mathcal{C}_\vee(c_{A_n}, c_{B_m})$	$(\ \mathcal{C}_\vee(c_{A_1}, c_{B_1}) \wedge \cdots \wedge \mathcal{C}_\vee(c_{A_n}, c_{B_1})\ ) \vee$ $\vdots \quad \vdots \quad \vdots \quad \vdots$ $(\ \mathcal{C}_\vee(c_{A_1}, c_{B_m}) \wedge \cdots \wedge \mathcal{C}_\vee(c_{A_n}, c_{B_m})\ )$
$c_{B_1} \wedge \cdots \wedge c_{B_m}$	$(\ \mathcal{C}_\vee(c_{A_1}, c_{B_1}) \wedge \cdots \wedge \mathcal{C}_\vee(c_{A_1}, c_{B_m})\ ) \vee$ $\vdots \quad \vdots \quad \vdots \quad \vdots$ $(\ \mathcal{C}_\vee(c_{A_n}, c_{B_1}) \wedge \cdots \wedge \mathcal{C}_\vee(c_{A_n}, c_{B_m})\ )$	$\mathcal{C}_\vee(c_{A_1}, c_{B_1}) \wedge \cdots \wedge \mathcal{C}_\vee(c_{A_n}, c_{B_1}) \wedge$ $\vdots \quad \vdots \quad \vdots$ $\mathcal{C}_\vee(c_{A_1}, c_{B_m}) \wedge \cdots \wedge \mathcal{C}_\vee(c_{A_n}, c_{B_m})$

Table 8.4: $\mathcal{C}_\vee(c_A,\, c_B)$ defines the second inductive case of the cardinality propagation function $\mathcal{C}$: The query body is a disjunction of two atoms the cardinality specifications of which are complex.

Let $i : h \leftarrow A \vee B$ be a StreamLog query where A, B and h are atoms and i is a time interval. Let c_A and c_B be complex cardinality specifications in the cardinality constraints for A and B respectively with the validity time i. Table 8.4 contains the cardinality specification $\mathcal{C}_\vee(c_A,\, c_B)$ of the cardinality constraint for h with the validity time i.

Example 13.
If $c_A = (= 6 \vee \leq 3)$ and $c_B = (\geq 4 \wedge \leq 7)$ then
$\mathcal{C}_\vee(c_A, c_B) = \mathcal{C}_\vee(\ (= 6 \vee \leq 3),\ (\geq 4 \wedge \leq 7)\ ) =$
$(\ \mathcal{C}_\vee(= 6, \geq 4) \wedge \mathcal{C}_\vee(= 6, \leq 7)\ ) \vee (\ \mathcal{C}_\vee(\leq 3, \geq 4) \wedge \mathcal{C}_\vee(\leq 3, \leq 7)\ ) =$
$(\ (\geq 10 \wedge (\geq 6 \wedge \leq 13)) \vee (\geq 4 \wedge \leq 10)\ ) = (\ (\geq 10 \wedge \leq 13) \vee (\geq 4 \wedge \leq 10)\ ) = (\geq 4 \wedge \leq 13).$

The resulting cardinality specification $(\ (\geq 10 \wedge (\geq 6 \wedge \leq 13)) \vee (\geq 4 \wedge \leq 10)\ )$ in Example 13 can be simplified according to the rules defined in Section 8.7. This simplification is not only necessary for the readability of the respective derived constraint but also for the efficiency of constraint propagation since derived constraints can also be propagated.

As can be seen above, the first inductive case (Table 8.3) is led back to the first base case (Table 8.1) and the second inductive case (Table 8.4) is led back to the second base case (Table 8.2). The idea of inductive cases is to push the operator $\mathcal{C}_\wedge$ (if the respective query body is a conjunction of atoms) or the operator $\mathcal{C}_\vee$ (if the respective query body is a disjunction of atoms) inside a complex cardinality specification until simple cardinality specifications are reached.

8.5.2 Arbitrary Number of Atoms in a Query Body

The general case of cardinality constraint propagation with respect to queries with multiple atoms in their bodies can be led back to the base cases of cardinality propagation with regards to queries with two atoms in their bodies as described in the following.

Simple Cardinality Specifications

A conjunction of multiple atoms is logically equivalent to a nested conjunction of each pair of these atoms or their nested conjunctions, e.g. let A_1, A_2, A_3, A_4, A_5 be atoms, than $A_1 \wedge A_2 \wedge A_3 \wedge A_4 \wedge A_5$ is equivalent to $(\ (\ (A_1 \wedge A_2) \wedge (A_3 \wedge A_4)\ ) \wedge A_5\ )$.

Let $i : h \leftarrow A_1 \wedge \cdots \wedge A_k$ be a StreamLog query where $k \in \mathbb{N}$, $A_1, \ldots, A_k, h$ are atoms, and i is a time interval. Let $c_{A_1}, \ldots, c_{A_k}$ be simple cardinality specifications of the cardinality constraints of $A_1, \ldots, A_k$ respectively with the validity time i. The atoms $A_1, \ldots, A_k$ of the query body are divided into the following tree groups according to the form of their simple cardinality specifications:

1. Let $A_{i_1}, \ldots, A_{i_r}$ be the set of all atoms in the query body with the simple cardinality specifications $= n_{i_1}, \ldots, = n_{i_r}$, $n_{i_1}, \ldots, n_{i_r} \in \mathbb{N}_0$, then $\mathcal{C}_\wedge(c_{A_{i_1}}, \ldots, c_{A_{i_r}}) = = \prod_{i=i_1}^{i_r} n_i$.

2. Let $A_{j_1}, \ldots, A_{j_s}$ be the set of all atoms in the query body with the simple cardinality specifications $\leq n_{j_1}, \ldots, \leq n_{j_s},\ n_{j_1}, \ldots, n_{j_s} \in \mathbb{N}_0$, then $\mathcal{C}_\wedge(c_{A_{j_1}}, \ldots, c_{A_{j_s}}) = \leq \prod_{j=j_1}^{j_s} n_j$.

3. Let $A_{l_1}, \ldots, A_{l_v}$ be the set of all atoms in the query body with the simple cardinality specifications $\geq n_{l_1}, \ldots, \geq n_{l_v},\ n_{l_1}, \ldots, n_{l_v} \in \mathbb{N}_0$, then $\mathcal{C}_\wedge(c_{A_{l_1}}, \ldots, c_{A_{l_v}}) = \geq \prod_{l=l_1}^{l_v} n_l$.

As explained above, $A_1 \wedge \cdots \wedge A_k \equiv (A_{i_1} \wedge \cdots \wedge A_{i_r}) \wedge (A_{j_1} \wedge \cdots \wedge A_{j_s}) \wedge (A_{l_1} \wedge \cdots \wedge A_{l_v})$ and $\mathcal{C}_\wedge(c_{A_1}, \ldots, c_{A_k}) = \mathcal{C}_\wedge(\mathcal{C}_\wedge(c_{A_{i_1}}, \ldots, c_{A_{i_r}}), \mathcal{C}_\wedge(c_{A_{j_1}}, \ldots, c_{A_{j_s}}), \mathcal{C}_\wedge(c_{A_{l_1}}, \ldots, c_{A_{l_v}})) = (\ = 0 \vee\ \geq \prod_{i=i_1}^{i_r} n_i \cdot \prod_{l=l_1}^{l_v} n_l\ )$.

Remember that three comparison operators are considered, they are: $=, \leq, \geq$. Therefore, there are 7 $(= 2^3 - 1 = (\sum_{k=0}^{3} \binom{3}{k}) - 1)$ cases in general. But all of them can be led back to the base case (Table 8.1) analogously to the case described above.

Just like a conjunction, a disjunction of multiple atoms is logically equivalent to a nested disjunction of each pair of these atoms or their nested disjunctions, e.g. if A_1, A_2, A_3, A_4, A_5 are atoms than $A_1 \vee A_2 \vee A_3 \vee A_4 \vee A_5$ is equivalent to $(\ (\ (A_1 \vee A_2) \vee (A_3 \vee A_4)\ ) \vee A_5\ )$.

Let $i : h \leftarrow A_1 \vee \cdots \vee A_k$ be a StreamLog query where $k \in \mathbb{N}$, $A_1, \ldots, A_k, h$ are atoms, and i is a time interval. Let $c_{A_1}, \ldots, c_{A_k}$ be simple cardinality specifications of the cardinality constraints of $A_1, \ldots, A_k$ respectively with the validity time i. The atoms $A_1, \ldots, A_k$ of the query body are divided into the following three groups according to the form of their simple cardinality specifications:

1. Let $A_{i_1}, \ldots, A_{i_r}$ be the set of all atoms in the query body with the simple cardinality specifications $= n_{i_1}, \ldots, = n_{i_r},\ n_{i_1}, \ldots, n_{i_r} \in \mathbb{N}_0$, then $\mathcal{C}_\vee(c_{A_{i_1}}, \ldots, c_{A_{i_r}}) = = \sum_{i=i_1}^{i_r} n_i$.

2. Let $A_{j_1}, \ldots, A_{j_s}$ be the set of all atoms in the query body with the simple cardinality specifications $\leq n_{j_1}, \ldots, \leq n_{j_s},\ n_{j_1}, \ldots, n_{j_s} \in \mathbb{N}_0$, then $\mathcal{C}_\vee(c_{A_{j_1}}, \ldots, c_{A_{j_s}}) = \leq \sum_{j=j_1}^{j_s} n_j$.

3. Let $A_{l_1}, \ldots, A_{l_v}$ be the set of all atoms in the query body with the simple cardinality specifications $\geq n_{l_1}, \ldots, \geq n_{l_v},\ n_{l_1}, \ldots, n_{l_v} \in \mathbb{N}_0$, then $\mathcal{C}_\vee(c_{A_{l_1}}, \ldots, c_{A_{l_v}}) = \geq \sum_{l=l_1}^{l_v} n_l$.

As explained above, $A_1 \vee \cdots \vee A_k \equiv (A_{i_1} \vee \cdots \vee A_{i_r}) \vee (A_{j_1} \vee \cdots \vee A_{j_s}) \vee (A_{l_1} \vee \cdots \vee A_{l_v})$ and $\mathcal{C}_\vee(c_{A_1}, \ldots, c_{A_k}) = \mathcal{C}_\vee(\mathcal{C}_\vee(c_{A_{i_1}}, \ldots, c_{A_{i_r}}), \mathcal{C}_\vee(c_{A_{j_1}}, \ldots, c_{A_{j_s}}), \mathcal{C}_\vee(c_{A_{l_1}}, \ldots, c_{A_{l_v}})) =\geq \sum_{i=i_1}^{i_r} n_i + \sum_{l=l_1}^{l_v} n_l$.

Since three comparison operators $=, \leq, \geq$ are considered, also in this case there are 7 $(= 2^3 - 1 = (\sum_{k=0}^{3} \binom{3}{k}) - 1)$ cases in general. But all of them can be led back to the base case (Table 8.2) analogously to the case described above.

If a query body involves both conjunctions and disjunctions of atoms, the propagation of cardinality constraints is led back to the cases explained above. For example, let A_1, A_2, A_3, A_4, A_5, A_6 be atoms, than the query body $A_1 \vee A_2 \vee A_3 \vee A_4 \wedge A_5 \wedge A_6$ is equivalent to the query body ($(A_1 \vee A_2 \vee A_3) \vee (A_4 \wedge A_5 \wedge A_6)$). If $c_{A_1}, c_{A_2}, c_{A_3}, c_{A_4}, c_{A_5}, c_{A_6}$ are the simple cardinality specifications of the cardinality constraints for A_1, A_2, A_3, A_4, A_5, A_6 respectively, then the operator $\mathcal{C}$ is applied to these cardinality specifications and its index depends on whether the respective atoms are connected by $\wedge$ or by $\vee$. In the example, $\mathcal{C}_\vee(\mathcal{C}_\vee(c_{A_1}, c_{A_2}, c_{A_3}), \mathcal{C}_\wedge(c_{A_4}, c_{A_5}, c_{A_6}))$ is the cardinality specification for the derived cardinality constraint for the query head.

Complex Cardinality Specifications

If a query body is an arbitrary nested conjunction or disjunction of atoms such that the cardinality specifications of the constraints for these atoms are complex, then the method to treat arbitrary nesting of atoms in the query body and the method to deal with complex cardinality specifications are applied in combination. More exactly the operator $\mathcal{C}$ as defined in Table 8.3 and Table 8.4 is applied to each pair of complex cardinality specifications of the constraints for the atoms in the query body, than to the result of two such applications and so on until the cardinality specifications of the constraints of all atoms in the query body are considered.

Algorithm

The algorithm in Listing 8.1 derives the cardinality specification *cardSpec* of the atom derived by the rule the body of which is a *query*, i.e. an arbitrary nested conjunction or disjunction of atoms. The cardinality specifications of these atoms may be simple or complex. The algorithm is based on Tables 8.1–8.4. The evaluation time of the *query* is equal to the validity time of the propagated constraints.

Listing 8.1: *propagateCardSpecs*(*query*) propagates cardinality constraints with respect to a *query* the evaluation time of which equals to the validity time of the propagated constraints

```
1 propagateCardSpecs(query) {
2   if query is a conjunction then
3       leftConjunct ← getLeftConjunct(query)
```

```
4       rightConjunct ← getRightConjunct(query)
5       leftCardSpec ← propagateCardSpecs(leftConjunct)
6       rightCardSpec ← propagateCardSpecs(rightConjunct)
7       cardSpec ← C∧(leftCardSpec,rightCardSpec)
8   else if query is a disjunction then
9       leftDisjunct ← getLeftDisjunct(query)
10      rightDisjunct ← getRightDisjunct(query)
11      leftCardSpec ← propagateCardSpecs(leftDisjunct)
12      rightCardSpec ← propagateCardSpecs(rightDisjunct)
13      cardSpec ← C∨(leftCardSpec,rightCardSpec)
14  else /* query is an atom */
15      cardSpec ← getCardSpec(query)
16  end end
17  return cardSpec
18 }
```

8.6 Constraint Validity Time Differs from Query Evaluation Time

This section is devoted to the case in which the validity time intervals of cardinality constraints do not coincide with the evaluation time intervals of the CEP queries with respect to which the constraints are propagated.

Listing 8.2: StreamLog query recognizing high temperature in an area

```
1 Range 1h:
2   highTemp(area(A), value(V)) ←
3     temp(area(A), value(V)) ∧ V > 30
```

Listing 8.3: ESCL constraint restricting the number of temperature measurements

```
1 Range 1min:
2   ∃=1 temp(area(A), value(V)) ←
```

Consider the query in Listing 8.2. If the temperature in an area is higher than 30 Degrees, the query reports the measured temperature every hour. Consider the constraint in Listing 8.3. It restricts the number of temperature measurements in an area during every minute to be exactly one. The constraint can be propagated with respect to the query to determine the cardinality of the high temperature events even though the query evaluation time differs from the constraint validity time: There are at most 60 high temperature events during each hour.

The case in which the time intervals of CEP queries and constraints are not equal is more general, more common, and also more difficult than that considered in Section 8.5. In the following, we start with the base case in which a query body contains only one atom and continue with the inductive case in which a query body is an arbitrary nested conjunction or disjunction of atoms. Both simple and complex cardinality specifications are considered.

Let $i : h \leftarrow b$ be a StreamLog query where i is the evaluation time interval, h is the head, and b is the body of the query, i.e. a new event (state) is derived as specified by h if b matches the stream during i.

Let $j : \exists_c l \leftarrow$ be an ESCL cardinality constraint where j is the validity time interval, c is the cardinality specification, and l is the atom of the constraint, i.e. there are c events (states) matching l during j.

Assume l is one of the atoms in b such that the above cardinality constraint can be propagated with respect to the above query. Both i and j can be tumbling or sliding windows or time intervals. This time interval can be repeating or non-repeating, user-defined, now window, simple or complex event, relative timer event, or application state. The bounds of i and j must be known since otherwise the constraint propagation is not possible.

8.6.1 One Atom in a Query Body

This section is devoted to the base case in which $b = l$.

Simple Cardinality Specifications

c has the form Θn where $\Theta \in \{\geq, \leq, =\}$ and $n \in \mathbb{N}_0$. The following cases are possible:

1. i and j are tumbling windows
   (a) $i < j$[1]

[1] Actually, the duration of i is compared with the duration of j, i.e. $|i| < |j|$ is the accurate notation. But for the sake of readability, we write i and j when we mean the duration of the respective time intervals in the following.

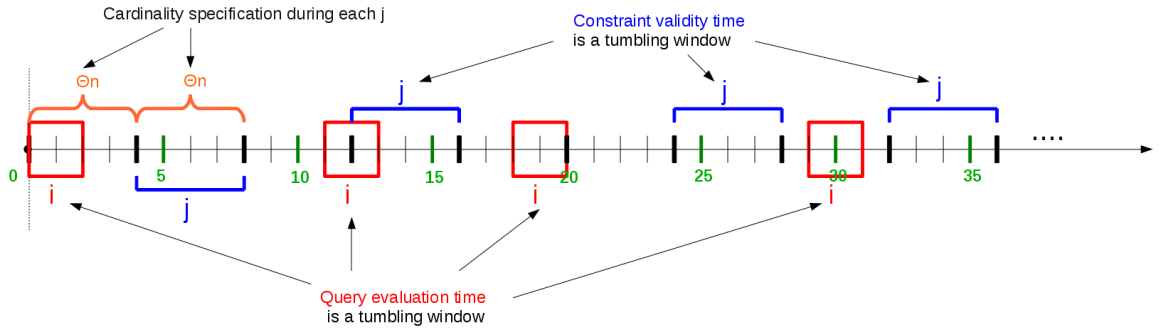

Figure 8.1: i and j are tumbling windows, $i < j$

i. $c = = n$

The lower bound of the derived cardinality specification is 0, since, e.g., there may be no events matching l during the time interval $[0, 2)$ (consider Figure 8.1). The upper bound of the derived cardinality specification is $2n$, since, e.g., there may be n events matching l in each of the time intervals $[11, 12)$ and $[12, 13)$. Hence, $i : \exists_{\leq 2n}\, h \leftarrow$ is the derived cardinality constraint.

ii. $c = \leq n$

By applying the same arguments as above, we obtain the cardinality constraint $i : \exists_{\leq 2n}\, h \leftarrow$.

iii. $c = \geq n$

During the time interval $[0, 2)$ there may be no, less than n, n, or more than n events matching l. Hence, 0 is the lower bound of the derived cardinality specification. It has no upper bound since $c = \geq n$. Therfore, $i : \exists_{\geq 0}\, h \leftarrow$ is derived.

(b) $i \geq j$

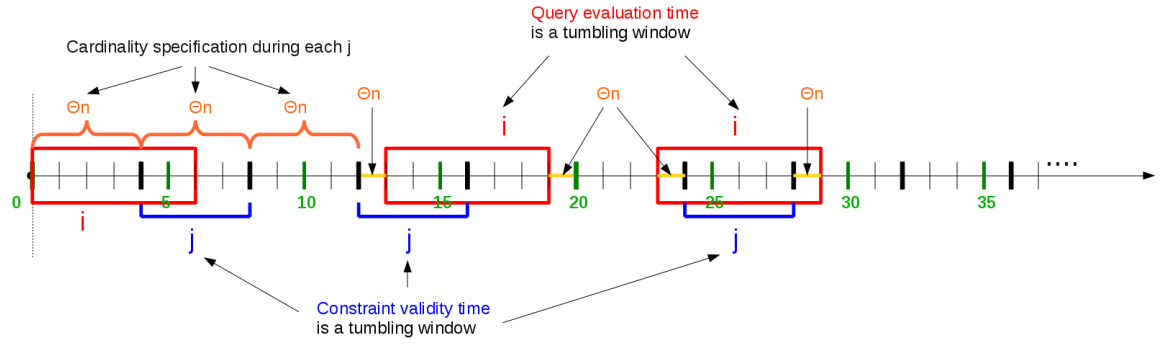

Figure 8.2: i and j are tumbling windows, $i \geq j$

i. $c = = n$

The lower bound of the derived cardinality specification is $\left(\lfloor \frac{i}{j} \rfloor - 1\right) \cdot n$ because, e.g., there can only be exactly n events matching l in each of the time intervals $[12, 13)$ and $[19, 20)$ and no such events during the interval $[13, 19)$ (consider Figure 8.2). So that $\left(\lfloor \frac{6}{4} \rfloor - 1\right) \cdot n = 0$ is yielded. The upper bound of the derived cardinality specification is $\left(\lceil \frac{i}{j} \rceil + 1\right) \cdot n$ since, e.g., there can be n events matching l in each of the intervals $[23, 24), [24, 28)$ and $[28, 29)$. Therefore, $\left(\lceil \frac{6}{4} \rceil + 1\right) \cdot n = 3n$. Hence, the derived cardinality constraint is $i : \exists_{\,w}\, h \leftarrow$ where $w := \left(\geq \left(\lfloor \frac{i}{j} \rfloor - 1\right) \cdot n \wedge \ \leq \left(\lceil \frac{i}{j} \rceil + 1\right) \cdot n\right)$.

ii. $c = \leq n$

Because of $c =\leq n$, there can be no events matching l during i. Therefore, 0 is the lower bound of the derived cardinality specification. Its upper bound is $\left(\lceil \frac{i}{j} \rceil + 1\right) \cdot n$, as argued above. Altogether, the derived cardinality constraint is $i : \exists_{\,w}\, h \leftarrow$ where $w :=\leq \left(\lceil \frac{i}{j} \rceil + 1\right) \cdot n$.

iii. $c = \geq n$

The derived constraint is $i : \exists_{\,w}\, h \leftarrow$ where $w :=\geq \left(\lfloor \frac{i}{j} \rfloor - 1\right) \cdot n$.

2. i is a tumbling window and j is a sliding window

(a) $i < j$

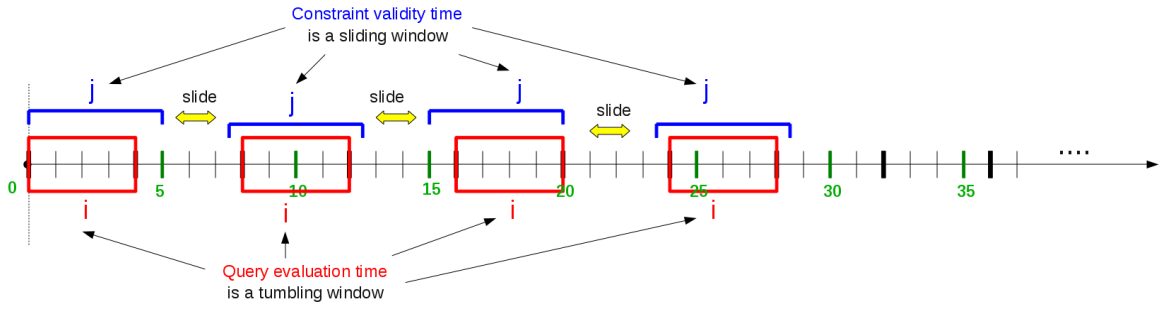

Figure 8.3: Each tumbling window i is completely within one sliding window j.

i. $c = = n$

$i : \exists_{\leq n}\, h \leftarrow$ is the derived constraint.

ii. $c = \leq n$

This case is analogous to the previous one and the constraint $i : \exists_{\leq n}\, h \leftarrow$ is derived.

However, in the above cases (i) and (ii), if there is some i which is not completely within one j but overlaps two intervals j, then in both cases $i : \exists_{\leq 2n}\, h \leftarrow$ is derived analogously to Cases 1(a)(i) and 1(a)(ii) respectively.

iii. $c = \geq n$

Consider Figure 8.3. If there are more than n events matching l during the time interval $[15, 16)$, there may be no such events within $[16, 20)$. Thus $i : \exists_{\geq 0}\, h \leftarrow$ is derived.

(b) $i \geq j$

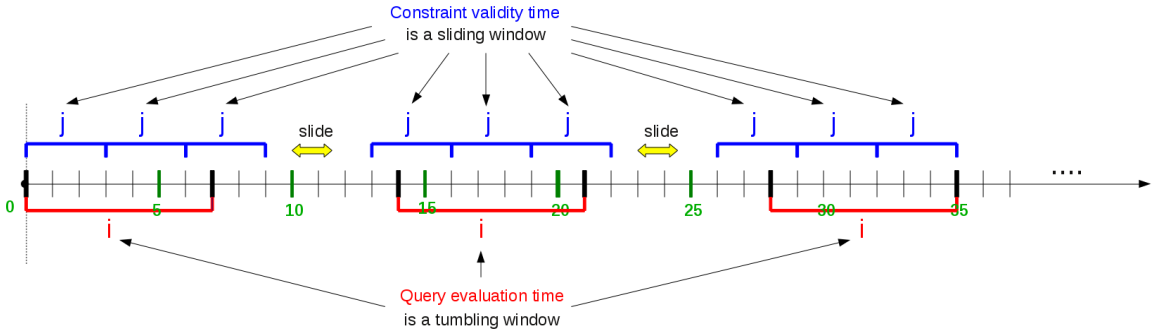

Figure 8.4: i is a tumbling window, j is a sliding window and $i \geq j$

Cardinality constraint propagation in this case coincides with Case 1(b). Compare Figure 8.4 with the results in Case 1(b).

i \ j	tumbling window	sliding window	time interval
tumbling window, sliding window or time interval	$\leq 2n$	$\leq n$	≥ 0

Table 8.5: $i < j$ and $c == n$ (or $c = \leq n$)

i \ j	tumbling window, sliding window or time interval
tumbling window, sliding window or time interval	≥ 0

Table 8.6: $i < j$ and $c = \geq n$

Cardinality constraint propagation in all other cases works just analogously to the cases described above. Tables 8.5 – 8.9 summarize the results.

These tables also contain the results of cardinality constraint propagation if the constraint validity time j or the query evaluation time i is a time interval. This time interval can be a now window, a user defined window, an event, or an application state. (Consider

i \ j	tumbling window or sliding window	time interval
tumbling window, sliding window or time interval	$\geq \left(\lfloor \frac{i}{j} \rfloor - 1\right) \cdot n \wedge \leq \left(\lceil \frac{i}{j} \rceil + 1\right) \cdot n$	≥ 0

Table 8.7: $i \geq j$ and $c == n$

i \ j	tumbling window or sliding window	time interval
tumbling window, sliding window or time interval	$\leq \left(\lceil \frac{i}{j} \rceil + 1\right) \cdot n$	≥ 0

Table 8.8: $i \geq j$ and $c = \leq n$

Chapter 4 for the definitions of these notions.) This time interval can be repeating or non-repeating. If it is repeating, it is neither tumbling nor sliding window because these kinds of windows are explicitly treated. The bounds of this time interval must be known at compile time since otherwise no constraint propagation is possible at compile time (more exactly: the default cardinality specification ≥ 0 will be derived). The bounds of an event or even an application state are often (but not always) known at compile time.

Example 14.
It is known that during the time periods from 6 am until 9 am and from 16 pm until 19 pm on work days the state of the central station of the city is "*overcrowded*" because of rush-hour.

If the constraint validity time j is a time interval and the query evaluation time i is a tumbling window, a sliding window, or a time interval, the general case is that there are some intervals i which overlap no interval j. As a consequence, no propagation of the constraints with the validity time j with respect to the queries with the evaluation time i is possible (more exactly: the default cardinality specification ≥ 0 is derived in this case), compare the last columns of Tables 8.5 – 8.9.

In seldom cases, each tumbling window, sliding window, or time interval i may contain the time interval j or be contained in the time interval j. In this case, the propagation of the constraints with validity time j with respect to the queries with the evaluation time i coincides with the respective cases in which i and j are tumbling or sliding windows.

i \ j	tumbling window or sliding window	time interval
tumbling window, sliding window or time interval	$\geq \left(\lfloor \frac{i}{j} \rfloor - 1\right) \cdot n$	≥ 0

Table 8.9: $i \geq j$ and $c = \geq n$

Complex Cardinality Specifications

If the cardinality specification c is complex (i.e., a conjunction or a disjunction of simple or complex cardinality specifications), the cardinality propagation function is recursively applied to each conjunct or each disjunct of c as the following examples demonstrate.

Example 15.
Let *Range* $1h$, *Slide* $30min : h \leftarrow l$ be the StreamLog query where h is the query head, l is the query body, and $i = (Range\ 1h,\ Slide\ 30min)$ is the query evaluation time which is a sliding window.

Let *Range* $30min$, *Slide* $10min : \exists_{\geq 4 \wedge \leq 7}\, l \leftarrow$ be the ESCL cardinality constraint where l is the atom of the constraint, $c = (\geq 4 \wedge \leq 7)$ is the complex cardinality specification, and $j = (Range\ 30min,\ Slide\ 10min)$ is the constraint validity time which is a sliding window.

j and i are both sliding windows and $i > j$. The cardinality propagation function $\mathcal{C}$ is applied to each conjunct of c and the following result is derived according to Table 8.9 and Table 8.8:
$\mathcal{C}(c) = (\ \mathcal{C}(\geq 4) \wedge \mathcal{C}(\leq 7)\ ) = (\ \geq \left(\lfloor \frac{1h}{30min} \rfloor - 1\right) \cdot 4 \wedge\ \leq \left(\lceil \frac{1h}{30min} \rceil + 1\right) \cdot 7\ ) = (\ \geq 4 \wedge \leq 21\ )$
Therefore, *Range* $1h$, *Slide* $30min : \exists_{\geq 4 \wedge \leq 21}\, h \leftarrow$ is the derived constraint.

Example 16.
Let *Range* $10min$, *Slide* $1min : h \leftarrow l$ be the StreamLog query where h is the query head, l is the query body, and $i = (Range\ 10min,\ Slide\ 1min)$ is the query evaluation time which is a sliding window.

Let *Range* $30min : \exists_{(=0 \vee \geq 3) \wedge \leq 7}\, l \leftarrow$ be the ESCL cardinality constraint where l is the atom of the constraint, $c = ((= 0 \vee \geq 3) \wedge \leq 7)$ is the complex cardinality specification, and $j = (Range\ 30min)$ is the constraint validity time which is a tumbling window.

i is a sliding window, j is a tumbling window, and $i < j$. The cardinality propagation function $\mathcal{C}$ is applied to each conjunct and each disjunct of c and the following result is derived according to Table 8.5, Table 8.6, and the simplification rules in Section 8.7:
$\mathcal{C}(c) = (\ (\mathcal{C}(= 0) \vee \mathcal{C}(\geq 3)) \wedge \mathcal{C}(\leq 7)\ ) = (\ (\leq 2 \cdot 0 \vee \geq 0) \wedge \leq 2 \cdot 7\ ) = (\ \geq 0 \wedge \leq 14\ ) \equiv \leq 14$

Therefore, *Range* $10min : \exists_{countryroads \leq 14}\, h \leftarrow$ is the derived constraint.

Example 16 can seem quite confusing: There are not more than 7 events during 30 minutes (j) then there are not more than 14 events during 10 minutes (i). But remember that some sliding window i can overlap two sequential tumbling windows j_1 and j_2 such that during each of the time intervals $j_1 \cap i$ and $j_2 \cap i$ 7 events matching l arrive. Therefore, 14 events will be derived by the query with the evaluation time i.

For the sake of generality in this report, we do not assume events to be equally distributed during a time interval. If this assumption is made, more concise cardinality constraints could be propagated. This is a topic of future research.

Algorithm

The algorithm in Listing 8.4 derives the cardinality specification *cardSpec* of the atom derived by the query the body of which is a single atom. The evaluation time i of the query does not coincide with the validity time j of the propagated constraint.

The cardinality specification c of the only atom in the query body may be simple or complex. If c is an arbitrary nested conjunction (or disjunction) of cardinality specifications, the algorithm in Listing 8.4 transforms each conjunct (or disjunct) of c with the validity time j into the time interval i (which is the evaluation time of the query), builds a conjunction (or a disjunction) of these transformed cardinality specifications, simplifies it according to the rules in Section 8.7, and returns the result.

The algorithm is based on Tables 8.5–8.9. The access to these tables is performed by the function $lookTable(c, i, j)$. It depends on the parameters of this function which table is accessed.

Listing 8.4: $transformCardSpec(c, i, j)$ transforms the cardinality specification c from the validity time j into the validity time i

```
1  transformCardSpec(c, i, j) {
2    if c is a conjunction then
3       leftConjunct ← getLeftConjunct(c)
4       rightConjunct ← getRightConjunct(c)
5       transformedLeftConjunct ← transformCardSpec(leftConjunct, i, j)
6       transformedRightConjunct ← transformCardSpec(rightConjunct, i, j)
7       result ← buildConjunction(transformedLeftConjunct,
8                                 transformedRightConjunct) else
9    if c is a disjunction then
10      leftDisjunct ← getLeftDisjunct(c)
```

```
11      rightDisjunct ← getRightDisjunct(c)
12      transformedLeftDisjunct ← transformCardSpec(leftDisjunct, i, j)
13      transformedRightDisjunct ← transformCardSpec(rightDisjunct, i, j)
14      result ← buildDisjunction(transformedLeftDisjunct,
15                                transformedRightDisjunct) else
16      /* c is simple */
17      result ← lookTable(c, i, j)
18    end end
19    cardSpec ← simplify(result)
20    return cardSpec
21  }
```

8.6.2 Arbitrary Number of Atoms in a Query Body

This section is devoted to the most general case of cardinality propagation: A query body is an arbitrary nested conjunction or disjunction of atoms. The query evaluation time does not coincide with the validity time intervals of the respective cardinality constraints. The cardinality specifications of these constraints may be simple or complex. To implement the cardinality propagation in this case, a combination of the solutions described above is necessary. Example 17 illustrates it.

Example 17.
Let q of the form

$$\mathit{Range}\ 1h : h \leftarrow l_1 \vee (l_2 \wedge l_3)$$

be the StreamLog query. The evaluation time of q is the tumbling window i.

Let

$$\begin{array}{lll} \mathit{Range}\ 2h & : \exists_{(\geq 5 \wedge \leq 7) \vee (=2)} & l_1 \leftarrow \\ \mathit{Range}\ 3h,\ \mathit{Slide}\ 15min & : \exists_{\leq 2} & l_2 \leftarrow \\ \mathit{Range}\ 30min,\ \mathit{Slide}\ 10min & : \exists_{=4} & l_3 \leftarrow \end{array}$$

be the ESCL cardinality constraints. The validity time of the first constraint is the tumbling window. The validity time intervals of the second and the third constraints are sliding windows. The cardinality specification of the first constraint is complex. The cardinality specifications of the second and the third constraints are simple.

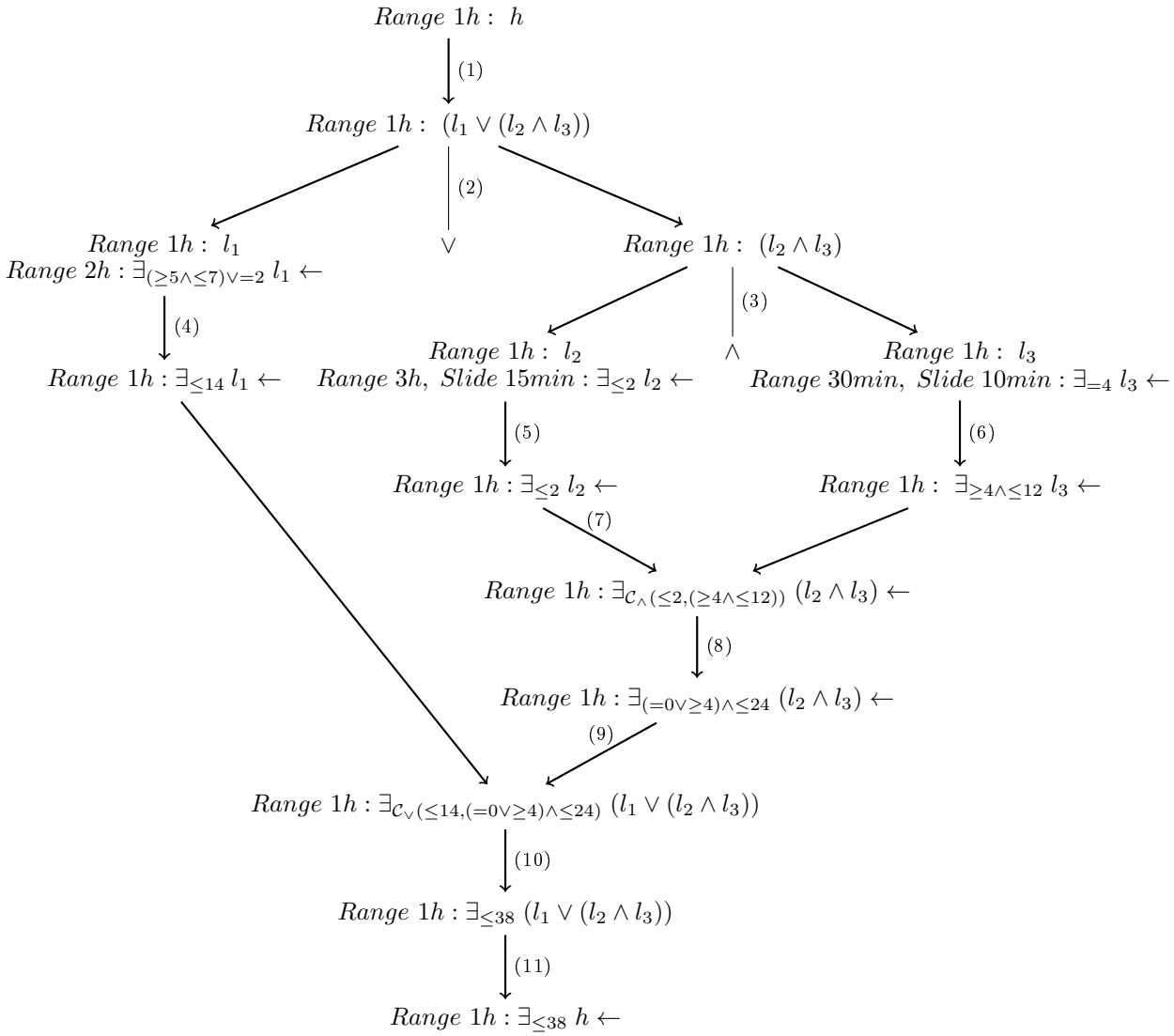

Figure 8.5: Illustration of Example 17

In order to propagate these constraints with respect to the query q, i.e. to derive the cardinality of h within the tumbling window i, the following steps are made (compare Figure 8.5):

1. *Range* $1h:\ h$ is led back to *Range* $1h:\ (l_1 \vee (l_2 \wedge l_3))$ because of the query q.
2. *Range* $1h:\ (l_1 \vee (l_2 \wedge l_3))$ is split into disjuncts *Range* $1h:\ l_1$ and *Range* $1h:\ (l_2 \wedge l_3)$.
3. *Range* $1h:\ (l_2 \wedge l_3)$ is split into conjuncts *Range* $1h:\ l_2$ and *Range* $1h:\ l_3$.
4. The cardinality constraint *Range* $2h : \exists_{(\geq 5 \wedge \leq 7) \vee = 2}\ l_1 \leftarrow$ is transformed into the tumbling window i according to Tables 8.5 and 8.6. The result is *Range* $1h : \exists_{\leq 14}\ l_1 \leftarrow$.

   More exactly (compare Listing 8.4): $transformCardSpec((\geq 5 \wedge \leq 7) \vee = 2,\ Range\ 1h,\ Range\ 2h) = (\geq 0 \wedge \leq 14) \vee \leq 14 = \leq 14$.

5. The cardinality constraint *Range* $3h$, *Slide* $15min : \exists_{\leq 2}\ l_2 \leftarrow$ is transformed into the tumbling window i according to Table 8.5. The result is *Range* $1h : \exists_{\leq 2}\ l_2 \leftarrow$.

6. The cardinality constraint *Range* $30min$, *Slide* $10min : \exists_{=4}\ l_3 \leftarrow$ is transformed into the tumbling window i according to Table 8.7. The result is *Range* $1h : \exists_{\geq 4 \wedge \leq 12}\ l_3 \leftarrow$.

7. Since l_2 and l_3 are conjuncts (compare Step 3), $\mathcal{C}_\wedge$ is applied to the transformed cardinality constraints for l_2 and l_3 obtained in Steps 5 and 6 respectively. The result is *Range* $1h : \exists_{\mathcal{C}_\wedge(\leq 2,(\geq 4 \wedge \leq 12))}\ (l_2 \wedge l_3) \leftarrow$.

8. According to Table 8.3, the result of Step 7 is simplified to *Range* $1h : \exists_{(=0 \vee \geq 4) \wedge \leq 24}\ (l_2 \wedge l_3) \leftarrow$.

   More exactly: $\mathcal{C}_\wedge(\leq 2, (\geq 4 \wedge \leq 12)) = \mathcal{C}_\wedge(\leq 2, \geq 4) \wedge \mathcal{C}_\wedge(\leq 2, \leq 12) = (= 0 \vee \geq 4) \wedge \leq 24$.

9. Since l_1 and $(l_2 \wedge l_3)$ are disjuncts (compare Step 2), $\mathcal{C}_\vee$ is applied to the transformed cardinality constraints for l_1 and $(l_2 \wedge l_3)$ obtained in Steps 4 and 8 respectively. The result is *Range* $1h : \exists_{\mathcal{C}_\vee(\leq 14,(=0 \vee \geq 4) \wedge \leq 24)}\ (l_1 \vee (l_2 \wedge l_3)) \leftarrow$.

10. According to Table 8.4, the result of Step 9 is simplified to *Range* $1h : \exists_{\leq 38}\ (l_1 \vee (l_2 \wedge l_3)) \leftarrow$.

    More exactly: $\mathcal{C}_\vee(\leq 14, (= 0 \vee \geq 4) \wedge \leq 24) = (\mathcal{C}_\vee(\leq 14, = 0) \vee \mathcal{C}_\vee(\leq 14, \geq 4)) \wedge \mathcal{C}_\vee(\leq 14, \leq 24) = (\leq 14 \vee \geq 4) \wedge \leq 38 = \leq 38$.

11. *Range* $1h : \exists_{\leq 38}\ h \leftarrow$ is the derived cardinality constraint.

Algorithm

The algorithm in Listing 8.5 is the extension of the algorithm in Listing 8.1 to propagate cardinality constraints the validity time intervals of which do not coincide with the evaluation time of the query with respect to which these constraints are propagated. The algorithm in Listing 8.5 is based on the algorithm in Listing 8.4 transforming cardinality specifications into the evaluation time interval of the query.

Listing 8.5: *propagateCardSpecs*(*query*) propagates cardinality constraints with respect to a *query*

```
1 propagateCardSpecs(query) {
2   if query is a conjunction then
3     leftConjunct ← getLeftConjunct(query)
4     rightConjunct ← getRightConjunct(query)
```

```
5       leftCardSpec ← propagateCardSpecs(leftConjunct)
6       rightCardSpec ← propagateCardSpecs(rightConjunct)
7       cardSpec ← C∧(leftCardSpec, rightCardSpec)
8   else if query is a disjunction then
9       leftDisjunct ← getLeftDisjunct(query)
10      rightDisjunct ← getRightDisjunct(query)
11      leftCardSpec ← propagateCardSpecs(leftDisjunct)
12      rightCardSpec ← propagateCardSpecs(rightDisjunct)
13      cardSpec ← C∨(leftCardSpec, rightCardSpec)
14  else /* query is an atom */
15      i ← getQueryEvaluationTime(query)
16      j ← getConstraintValidityTime(query)
17      c ← getCardSpec(query)
18      if i ← j then
19          cardSpec ← c else
20          cardSpec ← transformCardSpec(c, i, j)
21  end end end
22  return cardSpec
23 }
```

8.7 Simplification of Complex Cardinality Specifications

Example 13 shows that some (derived) cardinality specifications can be simplified. This simplification is not only necessary for the readability of constraints but also for the efficiency of constraint propagation since the cost for the propagation depends on the size of propagated constraints. Example 18 shows the propagation of the non-simplified cardinality specification obtained in Example 13.

Example 18.
If $c_A = (\ (\geq 10 \wedge (\geq 6 \wedge \leq 13)) \vee (\geq 4 \wedge \leq 10)\ )$ and $c_B = = 1$ then
$\mathcal{C}_\vee(c_A, c_B) = \mathcal{C}_\vee(\ (\ (\geq 10 \wedge (\geq 6 \wedge \leq 13)) \vee (\geq 4 \wedge \leq 10)\ ), = 1\ ) =$
$\mathcal{C}_\vee(\ (\geq 10 \wedge (\geq 6 \wedge \leq 13)), = 1\ ) \vee \mathcal{C}_\vee(\ (\geq 4 \wedge \leq 10), = 1\ ) =$
$(\ \mathcal{C}_\vee(\geq 10, = 1) \wedge \mathcal{C}_\vee((\geq 6 \wedge \leq 13), = 1)\ ) \vee (\ \mathcal{C}_\vee(\geq 4, = 1) \wedge \mathcal{C}_\vee(\leq 10, = 1)\ ) =$
$(\ \mathcal{C}_\vee(\geq 10, = 1) \wedge \mathcal{C}_\vee(\geq 6, = 1) \wedge \mathcal{C}_\vee(\leq 13, = 1)\ ) \vee (\ \mathcal{C}_\vee(\geq 4, = 1) \wedge \mathcal{C}_\vee(\leq 10, = 1)\ ) =$
$(\ \geq 11 \wedge \geq 7 \wedge (\geq 1 \wedge \leq 14)\ ) \vee (\ \geq 5 \wedge (\geq 1 \wedge \leq 11)\ )$

Let $\theta \in \{=, \leq, \geq\}, n_1, \ldots, n_k, n_i, n_j \in \mathbb{N}_0$. Simplification rules of complex cardinality

specifications are the following:

1. $(\theta n_1 \wedge \cdots \wedge \theta n_k) \equiv (\geq n_i \wedge \leq n_j)$ if n_i is the lower bound and n_j is the upper bound of the intersection of all the intervals specified by the conjuncts $\theta n_1, \ldots, \theta n_k$.[2]

   $(\theta n_1 \wedge \cdots \wedge \theta n_k) \equiv = n_i$ if $n_i = n_j$.

2. $(\theta n_1 \vee \cdots \vee \theta n_k) \equiv (\geq n_i \wedge \leq n_j)$ if n_i is the lower bound and n_j is the upper bound of the union of all the intervals specified by the disjuncts $\theta n_1, \ldots, \theta n_k$

3. $(= n_1 \vee = n_2) \equiv = n_1$, if $n_1 = n_2$

4. $(= n_1 \vee \leq n_2) \equiv \begin{cases} \leq n_2, \text{ if } n_1 \leq n_2 \\ \leq n_1, \text{ if } n_1 = n_2 + 1 \end{cases}$

5. $(= n_1 \vee \geq n_2) \equiv \begin{cases} \geq n_2, \text{ if } n_1 \geq n_2 \\ \geq n_1, \text{ if } n_1 = n_2 - 1 \end{cases}$

6. $(\leq n_1 \vee \leq n_2) \equiv \leq max(n_1, n_2)$

7. $(\geq n_1 \vee \geq n_2) \equiv \geq min(n_1, n_2)$

8. $(\geq n_1 \vee \leq n_2) \equiv \geq 0$, if $n_1 \leq n_2 + 1$

According to the first rule $(\geq 10 \wedge (\geq 6 \wedge \leq 13)) \equiv (\geq 10 \wedge \leq 13)$, according to the second rule $(\geq 10 \wedge \leq 13) \vee (\geq 4 \wedge \leq 10) \equiv (\geq 4 \wedge \leq 13)$. Compare the propagation of $(\ (\geq 10 \wedge (\geq 6 \wedge \leq 13)) \vee (\geq 4 \wedge \leq 10)\ )$ in Example 18 with the propagation of its equivalent simplified version $(\geq 4 \wedge \leq 13)$ in Example 19.

Example 19.
If $c_A = (\geq 4 \wedge \leq 13)$ and $c_B = = 1$ then
$\mathcal{C}_\vee(c_A, c_B) = \mathcal{C}_\vee(\ (\geq 4 \wedge \leq 13), = 1\ ) = \mathcal{C}_\vee(\geq 4, = 1) \wedge \mathcal{C}_\vee(\leq 13, = 1) =$
$(\ \geq 5 \wedge (\geq 1 \wedge \leq 14)\ ) \equiv (\ \geq 5 \wedge \leq 14\ )$

According to the simplification rules defined above, the result in Example 19 is equivalent to the result in Example 18:
$(\ \geq 11 \wedge \geq 7 \wedge (\geq 1 \wedge \leq 14)\ ) \vee (\ \geq 5 \wedge (\geq 1 \wedge \leq 11)\ ) \equiv$
$(\geq 11 \wedge \leq 14) \vee (\geq 5 \wedge \leq 11) \equiv (\geq 5 \wedge \leq 14)$.

[2]Remember that if this intersection is empty the cardinality specification $(\theta n_1 \wedge \cdots \wedge \theta n_k)$ is invalid, compare Definition 32.

Algorithm

The algorithm in Listing 8.6 simplifies the cardinality specification c by building the interval c represents (consider the algorithm in Listing 8.7) and by converting the interval into a cardinality specification which is the simplified equivalent of c.

Listing 8.6: $simplify(c)$ simplifies the cardinality specification c

```
1 simplify(c) {
2   interval ← buildInterval(c)
3   simplifiedCardSpec ← convertToCardSpec(interval)
4   return simplifiedCardSpec
5 }
```

Listing 8.7: $buildInterval(c)$ builds the interval represented by the cardinality specification c

```
1 buildInterval(c) {
2   if c is simple then
3      interval ← generateInterval(c) else
4      /* c is complex */
5      leftAdjunct ← getLeftAdjunct(c)
6      rightAdjunct ← getRightAdjunct(c)
7      leftInterval ← buildInterval(leftAdjunct)
8      rightInterval ← buildInterval(rightAdjunct)
9      if c is conjunction then
10        interval ← buildIntersection(leftInterval,rightInterval) else
11        /* c is disjunction */
12        interval ← buildUnion(leftInterval,rightInterval)
13  end end
14  return interval
15 }
```

The function $generateInterval(c)$ takes a simple cardinality specification c as an argument and generates its correspondent interval. For example, given a simple cardinality specification $c = (\geq 2)$, the function $generateInterval(c)$ will associate c to the right-open interval $[2, \infty)$.

The function $buildIntersection(i_1, i_2)$ takes two intervals i_1 and i_2 as parameters and returns their intersection as a result. For example, let $i_1 := [3, 5]$ and $i_2 := [2, \infty)$. $buildIntersection(i_1, i_2) = [3, 5] \cap [2, \infty) = [3, 5]$.

The function $buildUnion(i_1, i_2)$ takes two intervals i_1 and i_2 as parameters and returns their union as a result. For example, let $i_1 := [3, 5]$ and $i_2 := [2, \infty)$. $buildUnion(i_1, i_2) =$

$[3,5] \cup [2,\infty) = [2,\infty)$.

The function $convertToCardSpec(i)$ maps the interval i to its respective cardinality specification. For example, $convertToCardSpec([2,\infty)) =\geq 2$ and $convertToCardSpec([3,5]) = (\geq 3 \wedge \leq 5)$.

Chapter 9

Subsumption of CEP Conjunctive Queries

This chapter is devoted to the adaption of the subsumption algorithm for Horn clauses [56] (described in Section 3.3) to CEP queries. This adaption is necessary because CEP queries are different from clauses, in particular temporal conditions must be taken into consideration by an algorithm deciding subsumption of CEP queries. Section 9.1 motivates the need to consider subsumption between CEP queries for their semantic optimization. Section 9.2 defines the notion of substitution of CEP queries. Section 9.3 specifies the subsumption between the conjuncts of CEP queries. Finally, Section 9.4 presents the code of the algorithm, illustrates it by examples, and investigates its properties.

9.1 Motivation

Definition 33 (**CEP Conjunctive Query**)**.** A CEP conjunctive query is a conjunction of atomic event or state queries and conditions on the events or states matched by them.

Disjunctive queries do not have to be considered since each disjunctive query can be transformed into an equivalent set of conjunctive queries. For the sake of brevity, we say CEP queries and mean CEP conjunctive queries in the following.

An algorithm deciding subsumption of CEP queries is necessary for two reasons:

1. Multi-query optimization: The results of a more general CEP query can be saved and used to look up the answers for more specific queries instead of evaluating these queries against the whole event stream. As motivated in Section 2.2.3, the multi-query optimization in CEP is has a greater potential than in database systems.

2. Determination of the relevance of a constraint for a query (which is a base for the algorithm for the semantic rewriting of CEP queries, see Chapter 10): If a constraint body subsumes a query body, the constraint is relevant for the query and can be used for its semantic optimization.

Because of these reasons, an algorithm deciding subsumption between two StreamLog queries and subsumption of a StreamLog query by an ESCL query is needed. Subsumption between two ESCL queries is not needed. Only the fragments of StreamLog and ESCL are considered. The fragments are described in Section 9.3.

θ-subsumption of CEP queries is based on the same idea as θ-subsumption of clauses, compare Definition 2 with Definition 38. The subsumption algorithms for Horn clauses C and D described in Section 3.3 (e.g., [56], [78], [105, 106]) are applicable to CEP queries. They are based on the following steps:

1. Test whether the head of C subsumes the head of D with the substitution θ_0,
2. For each conjunct c_i in the body $c_1 \wedge \cdots \wedge c_n$ of C find a conjunct d_j in the body $d_1 \wedge \cdots \wedge d_m$ of D such that $c_i \supseteq_\theta d_j$ with the substitution $\theta_i,\ i \in \{1, \ldots, n\},\ j \in \{1, \ldots, m\}$,
3. Compute the substitutions θ of C and D which are compatible (Definition 35) with all substitutions $\theta_i,\ i \in \{0, \ldots, n\}$.

In other words, these algorithms are language independent if: (1) The notion of substitution is adapted (Section 9.2) and (2) The subsumption of conjuncts $c_i \supseteq_\theta d_j$ is defined (Section 9.3). These conjucts may also be negated. If C and D are CEP queries the algorithm deciding their subsumption must consider the temporal relations between their conjuncts c_i and d_j.

9.2 Substitution of CEP Queries

As mentioned above, the definition of θ-subsumption of clauses can be adapted to CEP queries. However, a substitution of two CEP queries is different from a substitution of two clauses. In the following, we first illustrate the difference by examples and than formally define the notion of θ-substitution of CEP queries.

For the sake of simplicity, we start in this section with CEP queries without negative literals and continue in Section 9.3 with arbitrary CEP queries. Let q_1 and q_2 be CEP queries without negative literals such that q_1 θ-subsumes q_2, denoted $q_1 \supseteq_\theta q_2$ (Definition

38). The substitution θ of q_1 and q_2 maps not only variables of q_1 to the respective values of q_2 like the classical substitution of clauses, but also the identifiers of q_1 to the respective identifiers of q_2. The latter is indispensable since the (temporal) conditions of q_1 and q_2 may involve identifiers and the subsumption relation between these conditions must be defined too. Consider the following example.

Example 20.
Let $q_1 = (i : q(X) \wedge j : q(c) \wedge e(i) < b(j))$ and $q_2 = (m : q(a) \wedge n : q(c) \wedge e(m) < b(n))$. $q_1 \supseteq_\theta q_2$ with the substitution $\theta = \{X \mapsto a, i \mapsto m, j \mapsto n\}$. In this case $q_1\theta = (m : q(a) \wedge n : q(c) \wedge e(m) < b(n)) = q_2$. But this is not always the case as the following example shows.

Example 21.
Let $q_1 = (q(X) \wedge q(c))$ and $q_2 = (m : q(a) \wedge n : q(c) \wedge e(m) < b(n))$. $q_1 \supseteq_\theta q_2$ with the substitution $\theta = \{X \mapsto a\}$ and $q_1\theta = (q(a) \wedge q(c)) \neq q_2 = (m : q(a) \wedge n : q(c) \wedge e(m) < b(n))$.

The above examples show that the application of a substitution θ to a CEP query q, denoted $q\theta$, is, of course, also different from the application of a substitution to a clause since not only the variables of q must be mapped to their respective terms as defined by θ but also the identifiers in q must be replaced by the respective identifiers as defined by θ.

Since StreamLog and ESCL are first-order logic languages, the notions *signature*, *term*, and *atom* can be adopted from the classical logic [29].

Let $Terms$ be a set of terms and $Variables$ be the set of variables. Note that $Variables \subseteq Terms$. Let $EventIdentifiers$ be the set of event identifiers and $StateIdentifiers$ be the set of state identifiers such that $Terms \cap EventIdentifiers \cap StateIdentifiers = \emptyset$.

Definition 34 (Substitution of CEP queries). A substitution θ of CEP queries is a mapping of variables to terms, event identifiers to event identifiers, and state identifiers to state identifiers, formally:

$$\theta : Variables \dot\cup EventIdentifiers \dot\cup StateIdentifiers \rightarrow Terms \dot\cup EventIdentifiers \dot\cup StateIdentifiers$$

$$\begin{aligned} &X \in Variables && \mapsto X\theta \in Terms \\ &e \in EventIdentifiers && \mapsto e\theta \in EventIdentifiers \\ &s \in StateIdentifiers && \mapsto s\theta \in StateIdentifiers \end{aligned}$$

Definition 35 (Compatible Substitutions). Two substitutions are compatible if they do not match the same variable to different terms and do not match the same identifier to different identifiers.

c \ d	while w: $i : q$	while w: $\neg\, i : q$
while v: $j : g$	1) $g\theta = q$ 2) v $\sqsupseteq$ w $(j \mapsto i)$	-
while v: $\neg\, j : g$	-	1) $g = q\theta$ 2) v $\sqsubseteq$ w $(i \mapsto j)$
while v: $\exists_s\, j : g$	1) $g\theta = q$ 2) v $\sqsupseteq$ w 3) $s = \geq 0$ or $s = \geq 1$ $(j \mapsto i)$	1) $g = q\theta$ 2) v $\sqsubseteq$ w 3) $s = = 0$ $(i \mapsto j)$

Table 9.1: (Part of the) definition of $c \supseteq_\theta d$ where $q, g \in Atoms,\ i, j \in EventIdentifiers \dot{\cup} StateIdentifiers,\ \mathsf{w}, \mathsf{v} \in \mathbb{TI},\ s \in Cardinalities$

Definition 36 (Application of a Substitution). Let θ be a substitution of CEP queries. The application of θ to:

- A constant c is an identical mapping, i.e. $c\theta = c$.
- A variable X is either $X\theta = X'$ if $X \mapsto X' \in \theta$ or $X\theta = X$ otherwise.
- An identifier i is either $i\theta = i'$ if $i \mapsto i' \in \theta$ or $i\theta = i$ otherwise.
- A beginning $b(i)$ or end $e(i)$ of the time interval of an event or a state referenced by i: $b(i)\theta = b(i\theta)$ and $e(i)\theta = e(i\theta)$.
- An arithmetical expression $x\Theta y$ (where x and y are expressions and $\Theta \in \{+, -, /, *, mod\}$) is led back to the application of θ to both parts of the expression, i.e. $(x\Theta y)\theta = x\theta\Theta y\theta$.
- A condition $x\Theta y$ (where x and y are expressions and $\Theta \in \{=, \neq, <, \leq, >, \geq\}$) is led back to the application of θ to both parts of the condition, i.e. $(x\Theta y)\theta = x\theta\Theta y\theta$.
- An atomic query $i : f(t_1, \ldots, t_n)$: $(i : f(t_1, \ldots, t_n))\theta = i\theta : f(t_1\theta, \ldots, t_n\theta)$.

Like in classical logic, the test of subsumption between two CEP queries must be preceded by the renaming substitution ν of common variables and identifiers to avoid their mishmash.

c \ d	$X < b$	$X > b$	$X \leq b$	$X \geq b$	$X = b$	$X \neq b$
$X < a$	$b \leq a$	-	$b < a$	-	$b < a$	-
$X > a$	-	$b \geq a$	-	$b > a$	$b > a$	-
$X \leq a$	$b \leq a + 1$	-	$b \leq a$	-	$b \leq a$	-
$X \geq a$	-	$b \geq a - 1$	-	$b \geq a$	$b \geq a$	-
$X = a$	-	-	-	-	$b = a$	-
$X \neq a$	$b \leq a$	$b \geq a$	$b < a$	$b > a$	$b \neq a$	$b = a$

Table 9.2: (Part of the) definition of $c \supseteq_\theta d$ where $X \in Variables,\ a, b \in Constants$

9.3 Subsumption between Conjuncts of CEP Queries

Let C be an ESCL or a StreamLog query and D be a StreamLog query. As motivated in Section 9.1, subsumption between ESCL queries is not needed. Tables 9.1–9.3 define the subsumption of the conjuncts d of D by the conjuncts c of C. The first column of the tables describes the forms c may have. The first row of the tables describes the forms of d. Minus in a cell of a table means that the respective $c \not\supseteq_\theta d$. If c and d belong to different tables then $c \not\supseteq_\theta d$. Plus in a cell of a table means that the respective $c \supseteq_\theta d$ unconditionally. If there is a condition in a cell of a table it means that the respective $c \supseteq_\theta d$ if the condition holds. If a cell of the table contains mappings of variables to terms or identifiers to identifiers these mappings are added to the substitution θ of the respective c and d. If g and q are atoms and $g\theta = q$ or $g = q\theta$ (Table 9.1) then their substitution θ is obtained like for atoms in a first-order language [29]. Example 22 illustrates the subsumption of StreamLog or ESCL queries.

Example 22.
Let $C = Range\ 2h:\ i : p(X) \wedge while\ i:\ j : q(X) \wedge e(i) < b(j) \wedge e(j) < b(i) + 1h \wedge while\ j:\ \neg q(a)$ and $D = Range\ 1h:\ m : p(a) \wedge Range\ 1h:\ n : p(b) \wedge while\ n:\ k : q(b) \wedge e(n) < b(k) \wedge e(k) < b(n) + 10min \wedge while\ k:\ \neg q(Y)$ be CEP queries. $C \supseteq_\theta D$ with $\theta = \{X \mapsto b, i \mapsto n, j \mapsto k, Y \mapsto a\}$ since

$Range\ 2h:\ i : p(X)$	$\supseteq_\theta\ Range\ 1h:\ n : p(b)$	with $\theta = \{X \mapsto b, i \mapsto n\}$	(see Table 9.1)
$while\ i:\ j : q(X)$	$\supseteq_\theta\ while\ n:\ k : q(b)$	with $\theta = \{X \mapsto b, i \mapsto n, j \mapsto k\}$	(see Table 9.1)
$e(i) < b(j)$	$\supseteq_\theta\ e(n) < b(k)$	with $\theta = \{i \mapsto n, j \mapsto k\}$	(see Table 9.3)
$e(j) < b(i) + 1h$	$\supseteq_\theta\ e(k) < b(n) + 10min$	with $\theta = \{i \mapsto n, j \mapsto k\}$	(see Table 9.3)
$while\ j:\ \neg q(a)$	$\supseteq_\theta\ while\ k:\ \neg q(Y)$	with $\theta = \{j \mapsto k, Y \mapsto a\}$	(see Table 9.1)

For the sake of brevity, only fragments of the languages ESCL and StreamLog are considered. Spatial conditions are not considered in this section. However, the subsumption

between spatial conditions can be defined just analogously to the subsumption between temporal conditions. Arithmetical conditions considered in this section are also restricted since they do not allow the operators $*, /$, and *mod* and cannot be arbitrarily nested.

Note that all ESCL and StreamLog temporal conditions can be led back to the comparisons of time points. For example, i before j $\wedge$ {i,j} within 1h corresponds to $e(i) < b(j) \wedge e(j) < b(i) + 1h$ (compare Example 22). Note also that a window is not necessarily local for an atomic query but can be coded into temporal conditions and derived from them. For example, while i: j:q(X) corresponds to $j : q(X) \wedge b(i) \leq b(j) \wedge e(j) \leq e(i)$. Since ESCL constraints and StreamLog queries are defined within time intervals, there is a window for each atomic query.

Table 9.1 defines the subsumption of a StreamLog atomic query d by a StreamLog or ESCL atomic query c under consideration of their evaluation time intervals w and v respectively.

If g and q are positive literals of c and d respectively, $c \supseteq_\theta d$ if

1. There is a substitution θ such that $g\theta = q$ (defined like in classical logic [29]) and
2. (Each repeating) time interval w is completely covered by some (a set of repeating) time interval(s) v, denoted v $\sqsupseteq$ w.

If g and q are negative literals of c and d respectively, $c \supseteq_\theta d$ if the opposite conditions are satisfied, i.e. $g = q\theta$ and v $\sqsubseteq$ w.

An atom g of an ESCL atomic query c may have a quantifier $\exists_s$ where s is a cardinality specification. Since an atom q in a StreamLog atomic query d may neither have a quantifier nor be optional, q is equivalent to $\exists_{\geq 1} q$. The cardinality specification s of g must subsume the cardinality specification ≥ 1 of q for $c \supseteq_\theta d$ to hold. This is the case if $s = \geq 0$ or $s = \geq 1$.

The cardinality specification s of g can simulate the negation of g since $\neg g$ is equivalent to $\exists_{=0} g$. Therefore the subsumption of a negative StreamLog atomic query by a positive ESCL atomic query is possible.

9.4 Subsumption Algorithm for CEP Queries and its Properties

Definition 37 (Semantic Subsumption of CEP Queries). Let C be a StreamLog or ESCL query and D be a StreamLog query. C semantically subsumes D, denoted $C \supseteq_{sem}$

D, if each model of D is also a model for C.

However, θ-subsumption of CEP queries is the basis for the algorithm semantically rewriting CEP queries with respect to constraints (Chapter 10) where θ denotes the substitution of the CEP queries. Definition 2 of θ-subsumption for clauses can be adapted to CEP queries but it cannot be borrowed without change. The reason for this is the following: If a clause C θ-subsumes a clause D with a substitution θ, each literal of C can be found in D, denoted $C\theta \subseteq D$. This does not hold for CEP queries C and D in general because C and D can contain conditions. Consider Example 21 illustrating this point. Definition 38 is the inductive definition of θ-subsumption of CEP queries.

Definition 38 (**θ-Subsumption of CEP Queries**)**.** Let C be a StreamLog or ESCL query and D be a StreamLog query.

- If C and D are atomic queries, C θ-subsumes D, denoted $C \supseteq_\theta D$, if the respective conditions in Table 9.1 are satisfied. As a result of the substitution test a single substitution θ is returned. θ contains variable bindings (analogously to a substitution of atoms in classical logic) and identifier bindings as defined in Table 9.1.
- If $C = c_1 \wedge \cdots \wedge c_n$ and $D = d_1 \wedge \cdots \wedge d_m$, $C \supseteq_\theta D$ if:
  1. There is (at least one) injective (partial) mapping $p : c_i \mapsto d_j$ (where $i \in \{1, \ldots, n\}$ and $j \in \{1, \ldots, m\}$) such that $c_i \supseteq_\theta p(c_i)$ (as defined in Tables 9.1–9.3) with the substitution θ_i which is compatible with all previous atomic query substitutions $\theta_1, \ldots, \theta_{i-1}$ of C and D. Let σ be the set of all substitutions which are compatible with all atomic query substitutions of C and D.
  2. For all c_k, $k \in \{1, \ldots, n\}$ for which $p(c_k)$ is not defined the following holds: Each c_k is a condition which is satisfied by all bindings of sime substitutions $\theta \in \sigma$.

  As a result of the subsumption test the set of substitutions θ is returned.

The second condition of the inductive case in Definition 38 is necessary to ensure the completeness of θ-subsumption of CEP queries which will be proven at the end of this section.

Listing 9.1 contains the algorithm computing θ-subsumption of CEP queries which is an adaption of the θ-subsumption algorithm for Horn clauses proposed in [56]. The code of the algorithm is described in detail in the following.

Listing 9.1: θ-subsumption algorithm for CEP qeuries adapted from [56]

```
1  require: C : c_1 ∧ ··· ∧ c_n, D : d_1 ∧ ··· ∧ d_m are CEP queries
2  S_0 = {}
3  Conditions ← ∅
4  for i := 1 to n do
5     if uni(C, c_i, D) ≠ ∅
6     then S_i ← S_{i−1} ⊓ merge(uni(C, c_i, D))
7     else Conditions ← Conditions ∪ c_i
8     end
9  end
10 result ← filter(S_n, Conditions)
11 return result
```

The input of the algorithm are two CEP conjunctive queries C and D. Like the algorithm in [56], the algorithm in Listing 9.1 is based on the following auxiliary functions (which are formally defined in [56] and only informally described in this report):

- $uni(C, c_i, D)$ returns a set of matching substitutions from a conjunct c_i of C to some conjunct of D. This substitution set is represented as a multisubstitution mapping each variable to a set of terms.

  Example 23.
  The multisubstitution $\{ X \mapsto \{10, 20\}, Y \mapsto \{30, 40\} \}$ is a short hand notation for the following set of substitutions $\{ X \mapsto 10, Y \mapsto 30 \}$, $\{ X \mapsto 10, Y \mapsto 40 \}$, $\{ X \mapsto 20, Y \mapsto 30 \}$, and $\{ X \mapsto 20, Y \mapsto 40 \}$.

- $merge(\Theta)$ returns a merge of the set of multisubstitutions Θ. If two multisubstitutions in Θ map the same variable to different sets of terms $merge(\Theta)$ returns the multisubstitution mapping this variable to the union of these sets of terms and other variables to the set of terms as defined by some multisubstitution in Θ.

- $\Theta \sqcap \Sigma$ returns the intersection of two multisubstitutions Θ and Σ. If Θ and Σ map the same variable to different sets of terms $\Theta \sqcap \Sigma$ returns the multisubstitution mapping this variable to the intersection of these sets of terms and other variables to the set of terms as defined by Θ or Σ.

There are the following differences between the subsumption algorithms in [56] and in Listing 9.1:

1. Multisubstitutions of the algorithm in Listing 9.1 contain not only mappings of variables to sets of terms, but also mappings of identifiers to sets of identifiers.

2. The algorithm in [56] tests whether the head of the Horn clause C subsumes the head of the Horn clause D. The algorithm in Listing 9.1 omitts this test.

3. The set $Conditions$ contains all conjuncts c_i of C which do not subsume any conjunct of D. Compare Lines 3, 5, and 7 of the algorithm.

   The function $filter(\Theta, Conditions)$ takes the multisubstitution Θ and the set $Conditions$ as input and returns the filtered multisubstitution containing only those substitutions of Θ which satisfy the $Conditions$. Compare Line 10 of Listing 9.1. If the set $Conditions$ contains at least one element which is no condition (but a literal) or which is a condition satisfied by no substitution represented by Θ, the function $filter(\Theta, Conditions)$ returns an empty set, i.e. C does not θ-subsume D. The function $filter$ is necessary to realize the second condition of the inductive case in Definition 38.

   Example 24.
   $filter(\{\ X \mapsto \{10, 20\}, Y \mapsto \{30, 40\}\ \},\ X < 15) = \{\ X \mapsto 10, Y \mapsto \{30, 40\}\ \}$
   $filter(\{\ X \mapsto \{10, 20\}, Y \mapsto \{30, 40\}\ \},\ X < 20) = \emptyset$
   $filter(\{\ X \mapsto \{10, 20\}, Y \mapsto \{30, 40\}\ \},\ X < Y - 20) = \{\ X \mapsto 10, Y \mapsto 40\ \}$
   $filter(\{\ X \mapsto \{10, 20\}, Y \mapsto \{30, 40\}\ \},\ X = Y - 20) = \{\ X \mapsto 10, Y \mapsto 30\ \}, \{\ X \mapsto 20, Y \mapsto 40\ \}$

Termination and Complexity

Let

$\|C\|$	be the number of conjuncts in C,
a_i	be the maximal arity of a literal c_i in C,
m_i	be the number of literals in D with the same predicate symbol as c_i,
c	be the number of distinct constants and
t	be the number of terms and identifiers in C and D.

As shown in [56]:

- Computing $uni(C, c_i, D)$ has complexity $a_i \cdot m_i$.
- Merge complexity is in $O(a_i \cdot m_i^4)$.
- The complexity of the intersection is $a_i \cdot m_i \cdot min(c, |D|)$.

The complexity of filtering is in $O(|C| \cdot t^2)$ since, in the worst case:

1. The are at most $|C|$ conditions in the set $Conditions$.

2. Each condition compares all bindings of at most two variables or at most two identifiers with each other, compare Tables 9.2 and 9.3.

3. There are at most t bindings for each variable or identifier.

As discussed in [56], the overall complexity of the subsumption algorithm is polynomial in the bast case and exponential in the worst case. Filtering does not change this result.

Correctness and Completeness

Proposition 1. *If C and D are conjunctive (CEP) queries then θ-subsumption of C and D is correct and complete, i.e. $C \supseteq_\theta D \Leftrightarrow C \supseteq_{sem} D$.*

The proof of Proposition 1 consists of two parts:

First part is the left to right direction: θ-subsumption of CEP conjunctive queries C and D is correct, i.e. $C \supseteq_\theta D \Rightarrow C \supseteq_{sem} D$.

Proof by induction over the number of conjuncts in C. Let n be the number of conjuncts in C, i.e. $C = c_1 \wedge \cdots \wedge c_n$.

Induction base: $n = 1$. Three cases are possible (compare Table 9.1):

1. $C = \mathsf{while\ v}\colon j : g$.

   Since $C \supseteq_\theta D$ there is the conjunct of the form $\mathsf{while\ w}\colon i : q$ in D such that (1) there is the substitution θ with $g\theta = q$ and $j \mapsto i \in \theta$ and (2) $\mathsf{v} \sqsupseteq \mathsf{w}$.

   Let S, σ, τ be an interpretation such that $S, \sigma, \tau \models D$. Since D is a conjunction, $S, \sigma, \tau \models \mathsf{while\ w}\colon i : q$. Since according to (1) q is an instance of g and according to (2) w is covered by v, $\mathsf{while\ w}\colon i : q \models \mathsf{while\ v}\colon j : g$. Since implication is transitive, $S, \sigma, \tau \models \mathsf{while\ v}\colon j : g = C$, i.e. $C \supseteq_{sem} D$.

2. $C = \mathsf{while\ v}\colon \neg\, j : g$.
   The proof is analogous to the proof of Case 1.

3. $C = \mathsf{while\ v}\colon \exists_s\, j : g$.
   The proof is analogous to the proof of Case 1.

Induction assumption: The statement holds if the number of conjuncts in C is n, i.e. if $C = c_1 \wedge \cdots \wedge c_n$ and $C \supseteq_\theta D$ then $C \supseteq_{sem} D$.

Induction step: $n \mapsto n + 1$, i.e. $C = c_1 \wedge \cdots \wedge c_n \wedge c_{n+1}$.

$C \supseteq_\theta D$ with the substitution set θ. Two cases are possible:

1. There is the conjunct d_j in D such that $c_{n+1} \supseteq_\theta d_j$ with the substitution set θ. $c_{n+1} \supseteq_\theta d_j$ implies $d_j \models c_{n+1}$, compare Tables 9.1–9.3. Let S, σ, τ be an interpretation such that $S, \sigma, \tau \models D$. Since D is a conjunction, $S, \sigma, \tau \models d_j$. Since implication is transitive, $S, \sigma, \tau \models c_{n+1}$ $(\star)$. According to the induction assumption $S, \sigma, \tau \models c_1 \wedge \cdots \wedge c_n$ $(\star\star)$. $(\star)$ and $(\star\star)$ imply $S, \sigma, \tau \models c_1 \wedge \cdots \wedge c_n \wedge c_{n+1} = C$, i.e. $C \supseteq_{sem} D$.

2. c_{n+1} is a condition which is satisfied by all bindings in θ. Since these bindings are yielded while matching C against D and since according to the induction assumption $D \models c_1 \wedge \cdots \wedge c_n$, $D \models c_1 \wedge \cdots \wedge c_n \wedge c_{n+1}$, i.e. $C \supseteq_{sem} D$.

□

Second part is the right to left direction: θ-subsumption of CEP conjunctive queries C and D is complete, i.e. $C \supseteq_\theta D \Leftarrow C \supseteq_{sem} D$.

Proof by induction over the number of conjuncts in C. Let n be the number of conjuncts in C, i.e. $C = c_1 \wedge \cdots \wedge c_n$.

Induction base: $n = 1$. Three cases are possible (compare Table 9.1):

1. $C = \mathsf{while\ v}\colon j : g$.

   Since $C \supseteq_{sem} D$ each model of D is also a model for C. Let S, σ, τ be an interpretation such that $S, \sigma, \tau \models D$ and therefore $S, \sigma, \tau \models C = \mathsf{while\ v}\colon j : g$, i.e. $D \models \mathsf{while\ v}\colon j : g$. As a consequence of this and the fact that D is a conjunctive query, there must be the conjunct of the form $\mathsf{while\ w}\colon i : q$ in D such that $\mathsf{while\ w}\colon i : q \models \mathsf{while\ v}\colon j : g$. In order for this condition to hold, the following two conditions must be satisfied: (1) q must be an instance of g, i.e. there is the substitution θ with $g\theta = q$ and $j \mapsto i \in \theta$ and (2) w must be covered by v, i.e. $\mathsf{v} \sqsupseteq \mathsf{w}$. Because of (1) and (2), $\mathsf{while\ v}\colon j : g \supseteq_\theta \mathsf{while\ w}\colon i : q$, i.e. $C \supseteq_\theta D$.

2. $C = \mathsf{while\ v}\colon \neg\, j : g$.
   The proof is analogous to the proof of Case 1.

3. $C = \mathsf{while\ v}\colon \exists_s j : g$.
   The proof is analogous to the proof of Case 1.

Induction assumption: The statement holds if the number of conjuncts in C is n, i.e. if $C = c_1 \wedge \cdots \wedge c_n$ and $C \supseteq_{sem} D$ then $C \supseteq_\theta D$ with the substitution set θ.

Induction step: $n \mapsto n + 1$, i.e. $C = c_1 \wedge \cdots \wedge c_n \wedge c_{n+1}$.

Since $C \supseteq_{sem} D$ each model of D is also a model for C. Let S, σ, τ be an interpretation such that $S, \sigma, \tau \models D$ and therefore $S, \sigma, \tau \models C = c_1 \wedge \cdots \wedge c_n \wedge c_{n+1}$, i.e. $D \models c_1 \wedge \cdots \wedge c_n \wedge c_{n+1}$ $(\star)$.

Since C is a conjunctive query, $S, \sigma, \tau \models c_1 \wedge \cdots \wedge c_n$. According to the induction assumption, there is the substitution set θ such that $c_1 \wedge \cdots \wedge c_n \supseteq_\theta D$ with θ.

Because of $(\star)$, two cases are possible:

1. There is the conjunct d_j in D such that $d_j \models c_{n+1}$. Therefore the respective conditions for c_{n+1} and d_j in Tables 9.1–9.3 must be satisfied. Hence there are substitutions $\theta' \in \theta$ such that $c_{n+1} \supseteq_\theta d_j$ with the substitution set θ' (where θ is the substitution set of $c_1 \wedge \cdots \wedge c_n \supseteq_\theta D$). Then $C \supseteq_\theta D$ with the substitution set θ'.

2. c_{n+1} is a condition satisfied by all bindings of some substitutions $\theta' \in \theta$ (where θ is the substitution set of $c_1 \wedge \cdots \wedge c_n \supseteq_\theta D$). Then $C \supseteq_\theta D$ with the substitution set θ'.

□

c \ d	$X < Y + b$	$X \leq Y + b$	$X = Y + b$
$X < Y + a$	$b \leq a$	$b < a$	$b < a$
$X \leq Y + a$	$b \leq a + 1$	$b \leq a$	$b \leq a$
$X = Y + a$	-	-	$b = a$

c \ d	$X < Y - b$	$X \leq Y - b$	$X = Y - b$
$X < Y - a$	$b \leq a$	$b < a$	$b < a$
$X \leq Y - a$	$b \leq a + 1$	$b \leq a$	$b \leq a$
$X = Y - a$	-	-	$b = a$

c \ d	$X > Y + b$	$X \geq Y + b$	$X = Y + b$
$X > Y + a$	$b \geq a$	$b > a$	$b > a$
$X \geq Y + a$	$b \geq a - 1$	$b \geq a$	$b \geq a$
$X = Y + a$	-	-	$b = a$

c \ d	$X > Y - b$	$X \geq Y - b$	$X = Y - b$
$X > Y - a$	$b \geq a$	$b > a$	$b > a$
$X \geq Y - a$	$b \geq a - 1$	$b \geq a$	$b \geq a$
$X = Y - a$	-	-	$b = a$

c \ d	$X = b$	$X = Y + b$	$X = Y - b$
$X = Y + a$	$+$ $(Y \mapsto b - a)$	$b = a$	-
$X = Y - a$	$+$ $(Y \mapsto b + a)$	-	$b = a$

c \ d	$X \neq Y + b$	$X \neq Y - b$
$X \neq Y + a$	$b = a$	-
$X \neq Y - a$	-	$b = a$

Table 9.3: (Part of the) definition of $c \supseteq_\theta d$ where $X, Y \in Variables \dot{\cup} \mathbb{T},\ a, b \in Constants$

Chapter 10

Semantic Rewriting of CEP Queries

This chapter is devoted to the adaption of the residue method for semantic optimization of database queries [31] to CEP queries. Section 10.1 presents the algorithm in pseudo code and illustrates it by examples. Afterwards its termination, complexity, and correctness are proven in Section 10.2.

10.1 Algorithm Illustrated by Examples

The semantic optimization algorithm extracts portions of constraints relevant for a query and uses them to rewrite the query into a more efficient form. The modified query may be *syntactically quite different* from the original one but it must be *semantically equivalent* to the initial query, i.e. it must return the same results as the original query for any streams satisfying the constraints. In other words, the algorithm does not change the semantics of a query but it changes its syntax to improve its evaluation.

The algorithm presented in this section is an adaption of the residue method[1] [31] for semantic optimization of database queries to CEP queries. The method is static and rather expensive. (Consider its complexity analysis in Section 10.2). That is why the residue method is not wide-spread in database systems where queries are usually ad hoc, data is completely available when queries are put, and hence, query answers are expected (almost) immediately. However, as motivated in Section 2.2.1, static semantic optimization of CEP queries may be inefficient if it happens before events are available and queries can be evaluated. Besides, repeated evaluation of CEP queries over a long period of time is worth their expensive static semantic optimization.

[1] Residues are portions of constraints relevant for a query which are used for the semantic optimization of the query.

The algorithm for the semantic rewriting of CEP queries is presented in pseudo code in Listing 10.1. The algorithm will be explained line-by-line and illustrated by examples in this section. The algorithm takes a CEP query and a set of CEP constraints as input and returns a set of semantically optimized queries as output. Therefore, the semantic rewriting of CEP queries is followed by the choice of a semantically optimized query to evaluate instead of each original query. To this end, the evaluation costs of all alternative semantically modified queries for an initial query are determined using cost models, and one of them with the lowest (estimated) evaluation cost is processed instead of the initial query. The algorithm specifying the query choice is a subject of future work.

Listing 10.1: Pseudo code of the algorithm for the semantic rewriting of CEP queries

```
1  Let q := (H_q ← B_q) be a query and C be a set of constraints.
2  optimize(q,C) {
3    S_q = ∅
4    FOR EACH c := (H_c ← B_c) ∈ C
5      IF B_c subsumes B_q with unifier σ and
6         a heuristic applies to q and H_cσ
7      THEN q' := applyHeuristics(q,H_cσ)
8           IF q' =⊥ or q' provides answers for q
9           THEN return q'
10          ELSE S_q := S_q ∪ q' ∪ optimize(q',C)
11          END
12     END
13   END
14   return S_q
15 }
```

Consider Listing 10.1 for the pseudo code of the algorithm for the semantic rewriting of CEP queries. Let q be a query in any event query language, let C be a set of constraints in any constraint language. q and C are the input of the algorithm, compare Line 1.

For the sake of readability and without loss of generality, we assume that a query q has the form $H_q \leftarrow B_q$ (where H_q is the head and B_q is the body of q) and each constraint $c \in C$ has the form $H_c \leftarrow B_c$ (where H_c is the head and B_c is the body of c). H_q and H_c are atoms. B_q and B_c are CEP queries, i.e. arbitrarily nested conjunctions or disjunctions of literals and conditions on events matched by the literals. Note that the evaluation time of q and the validity time of c are expressed by the conditions in B_q and B_c respectively.

Despite large syntactical similarity of constraints and queries, they are semantically different. A query derives new complex events (i.e. higher-level knowledge) out of base events arriving on streams (i.e. lower-level knowledge). If the body B_q of a query matches

events of a stream, its head H_q constructs new events. A constraint, in contrast, describes the properties of a stream. If the body B_c of a constraint matches events of a stream, its head H_c must be satisfied by the stream. This difference is important for semantic query rewriting as we will see below.

The algorithm is independent from an event query language and from a constraint language if the subsumption of a query body B_q by a constraint body B_c is defined (which is language dependent, of course). Consider Chapter 9 for the definition of subsumption of a StreamLog query body by an ESCL constraint body.

For a CEP query q of the form $H_q \leftarrow B_q$ (Line 1 in Listing 10.1), the algorithm returns the set of alternative semantically optimized queries S_q. The set S_q is empty at the beginning, compare Line 3.

For each CEP constraint $c \in C$ of the form $H_c \leftarrow B_c$ (Line 4) if the constraint body B_c is equal to or more general than the query body B_q (Line 5) than B_q implies B_c and, therefore, the modified constraint head $H_c\sigma$ (where σ is the unifier of B_c and B_q) holds for all streams for which the query q returns answers (i.e., if B_q matches a stream). In other words, $H_c\sigma$ does not change the semantics of q and can be used to optimize q,[2] e.g., by appending $H_c\sigma$ to B_q. In this way subsumption is used to check the relevance of a constraint for a query.

However, if this idea is applied naively to each query and each constraint relevant for the query, very many modified heads of constraints $H_c\sigma$ (called *residues* in [31]) can be appended of the query body. These residues can not only contribute nothing to the query optimization but even make the query evaluation less efficient as Example 25 demonstrates.

Example 25.
Consider the query q of the form

$$\begin{aligned} &items\_007\_is\_interested\_in(auctionID(A), itemID(I)) \leftarrow \\ &\quad i : bid(auctionID(A), itemID(I), bidderID(007), value(V)) \wedge \\ &\quad a : auction(auctionID(A)) \wedge \\ &\quad b(a) \leq b(i) \wedge e(i) \leq e(a) \end{aligned}$$

The query q selects the items the bidder with identifier 007 is interested in during each auction, i.e. the items 007 has bid for. More exactly: If there is an event *bid* during the state *auction* such that the *auction* state has the same value of the attribute *auctionID* as the *bid* event and the value of the attribute *bidderID* of the *bid* event is 007, then the new event $items\_007\_is\_interested\_in$ is derived. The event carries the respective

[2]This is the idea of the proof of the correctness of the algorithm for the semantic rewriting of CEP queries. The formal proof is given in Section 10.2.

auction identifier and item identifier.

Consider the constraint c of the form

$$\begin{aligned}
&\exists_{=1}\, j : bidderEnrollment(auctionID(A), bidderID(B)) \wedge \\
&b(a) \leq b(j) \wedge e(j) \leq b(i) \leftarrow \\
&\quad i : bid(auctionID(A), itemID(I), bidderID(B), value(V)) \wedge \\
&\quad a : auction(auctionID(A)) \wedge \\
&\quad b(a) \leq b(i) \wedge e(i) \leq e(a)
\end{aligned}$$

which states that each bidder participating in an auction must be enrolled for this auction. More precisely: If there is a *bid* event during the state *auction* which has the same value of the attribute *auctionID* as the *bid* event, then there is exactly one *bidderEnrollment* event after the *auction* state has started but before the *bid* event such that the *bidderEnrollment* event and the *bid* event have the same values of the attributes $auctionID$ and $bidderID$.

The body of c subsumes the body of q with the unifier $\sigma = \{B \mapsto 007\}$.[3] Therefore, the constraint c is relevant for the query q. Let H_c denote the head of c. The residue $H_c\sigma$ of c for q does not change the semantics of q with respect to all streams satisfying c. Hence, $H_c\sigma$ can be appended to the body of q. The result is the modified query q':

$$\begin{aligned}
&items_007_is_interested_in(auctionID(A), itemID(I)) \leftarrow \\
&\quad i : bid(auctionID(A), itemID(I), bidderID(007), value(V)) \wedge \\
&\quad a : auction(auctionID(A)) \wedge \\
&\quad b(a) \leq b(i) \wedge e(i) \leq e(a) \wedge \\
&\quad \{\ \exists_{=1}\, j : bidderEnrollment(auctionID(A), bidderID(007)) \wedge \\
&\quad b(a) \leq b(j) \wedge e(j) \leq b(i)\ \}
\end{aligned}$$

However, this residue $H_c\sigma$ (written in curly braces above) contributes nothing to the optimization of q. It makes the evaluation of q even less efficient by introducing the additional join and two additional conditions into the query.

To avoid the effect illustrated by Example 25, the search space of the algorithm must be restricted in some way. To this end, semantic optimization heuristics are used. If a heuristic applies to a query q and its residue $H_c\sigma$, $H_c\sigma$ contributes to the optimization of q and is used for this purpose. Otherwise the constraint c is uninteresting for the semantic optimization of q and is not further considered.

[3] Actually, the test of subsumption between two query bodies must be preceded by the renaming ν of common variables and identifiers. As a consequence, σ would also contain mappings for the identifiers and mappings for the other variables. To keep σ short and examples readable, in the following, the examples provoking problems without ν are avoided.

As the most overviews of semantic optimization techniques for database queries agree [31], [98], [38], there are six primary semantic optimization heuristics which have to be slightly adapted to CEP to cope with its peculiarities compared to database systems described in Section 2.2. In the following, the additional conditions which are not needed in database systems but are indispensable in CEP, are marked by italic type. Arguments which apply to database systems but which are hardly applicable to CEP, are disabled. The main semantic query optimization heuristics are the following:

1. Result by contradiction: A query does not have any answer if the existence of an answer would violate a constraint.

Example 26.
Consider the query q

$$\begin{aligned} &items\_007\_is\_interested\_in(auctionID(A), itemID(I)) \leftarrow \\ &\quad i : bid(auctionID(A), itemID(I), bidderID(007), value(V)) \wedge \\ &\quad a : auction(auctionID(A)) \wedge \\ &\quad b(a) \leq b(i) \wedge e(i) \leq e(a) \end{aligned}$$

which has already been presented in Example 25.

Consider the constraint c of the form

$$\begin{aligned} &B \neq 007 \leftarrow \\ &\quad i : bid(auctionID(A), itemID(I), bidderID(B), value(V)) \wedge \\ &\quad a : auction(auctionID(A)) \wedge \\ &\quad b(a) \leq b(i) \wedge e(i) \leq e(a) \end{aligned}$$

The constraint c prohibits the participation of the bidder with identifier 007 in any auction because of his bad behavior. More precisely: If there is an event *bid* during the state *auction*, then the value of the attribute *bidderID* of the event is not equal 007.

The body of c subsumes the body of q with the unifier $\sigma = \{B \mapsto 007\}$. The existence of answers for the query q would violate the constraint c. This is recognized by the algorithm since the residue $H_c\sigma = \{007 \neq 007\}$ (where H_c is the head of c) of c for q is a contradiction. That is why q is unsatisfiable (for all streams for which c holds) and is not evaluated.

This heuristic reduces the number of CEP queries which must be continuously evaluated against the event stream and, therefore, it reduces the workload of resources, in particular CPU and memory, such that these resources can evaluate more other (satisfiable) CEP queries quicker.

Since the body of the constraint c subsumes the body of the query q (which is tested

in Line 5 of Listing 10.1), c is relevant for q. Since the first heuristic applies to q and the residue $H_c\sigma$ of c for q (which is tested in Line 6), $H_c\sigma$ contributes to the semantic optimization of q. Since $H_c\sigma$ is a contradiction, the function $applyHeuristics(q, H_c\sigma)$ returns $\bot$ in Line 7, i.e. $q' = \bot$, which means that q is unsatisfiable. The condition in Line 8 evaluates to true and the evaluation of the algorithm is interrupted by returning q' (i.e. $\bot$) in Line 9. If $\bot$ is returned by the algorithm, the input query q is unsatisfiable with respect to all streams for which the input constraints C hold.

2. Result by transformation: It may be possible to answer a query without evaluating it if the semantic transformations provide the answers.

Example 27.
Consider the query q of the form

$$\begin{aligned} &averageTemp(sensor(s), area(a), value(avg(V))) \leftarrow \\ &\quad t : temp(sensor(s), area(a), value(V)) \wedge \\ &\quad w : Range\ 1h,\ Slide\ 30min \wedge \\ &\quad b(w) \leq b(t) \wedge e(t) \leq e(w) \end{aligned}$$

Every 30 minutes, the query q computes the average temperature measured by the sensor s in the area a during the last hour. More exactly: q looks for the events *temp* during the sliding window w of size one hour and with granularity of thirty minutes such that the value of the attribute *sensor* is s and the value of the attribute *area* is a. The value of the attribute *value* of such events is bound to the variable V. The average of all values bound to V is computed in the head of q and the result is a new event carrying the sensor identifier s, the area identifier a, and the computed average temperature.

Consider the constraint c of the form

$$\begin{aligned} &\exists_{=1}\, t : temp(sensor(s), area(a), value(20°)) \wedge \\ &b(t) = b(w) \wedge e(t) = b(w) \leftarrow \\ &\quad w : Range\ 1h,\ Slide\ 30min \end{aligned}$$

The constraint c says that every half an hour, there is exactly one event carrying a temperature measurement from the sensor s in the area a. The area is air-conditioned, that is way the temperature measured in the area is always 20°. The occurrence time of the event is a time point.

The body of c subsumes the body of q with the empty unifier $\sigma = \{\ \}$. Let H_c denote the head of c. $H_c\sigma = H_c = (\exists_{=1}\, t : temp(sensor(s), area(a), value(20°)) \wedge b(t) = b(w) \wedge e(t) = b(w))$ holds for q. Therefore, q is modified in the following way:

1. The part of the body of q $(t : temp(sensor(s), area(a), value(V)) \wedge b(w) \leq b(t) \wedge$

$e(t) \leq e(w))$ is omitted since it is implied by $H_c\sigma$.[4]

2. The variable V is bound to its only possible value 20°.

3. $(b(a) = b(w) \wedge e(a) = b(w))$ (where a is the identifier of the events derived by q) is appended to the query head since an event e referenced by a has the same the occurrence time as the event referenced by t e is derived from (compare Section 4.2).

The result is the ground query q' of the form:

$$a : averageTemp(sensor(s), area(a), value(20°)) \wedge$$
$$b(a) = b(w) \wedge e(a) = b(w) \leftarrow$$
$$w : Range\ 1h,\ Slide\ 30min$$

In other words, the event $averageTemp(sensor(s), area(a), value(20°))$ will be returned every half an hour and its occurrence time is a time point which coincides with the beginning of the sliding window w which is usually known at compile time. Hence, all results of q are known at compile time. There is no need to evaluate the query against the event stream. This allows to save resources (like the application of the first heuristic).

This example shows that the resulting semantically optimized query q' must not be a ground fact in order to provide answers to the initial query q and the set of the provided answers must not be finite. Both is usually not the case if q is a database query.

Consider Listing 10.1. Since the body of the constraint c subsumes the body of the query q (which is tested in Line 5), c is relevant for q. Since the second heuristic applies to q and the residue $H_c\sigma$ of c for q (which is tested in Line 6), $H_c\sigma$ contributes to the semantic optimization of q. Since the part of the body of q $(t : temp(sensor(s), area(a), value(V)) \wedge b(w) \leq b(t) \wedge e(t) \leq e(w))$ subsumes $H_c\sigma$, the function $applyHeuristics(q, H_c\sigma)$ returns the ground query q' (as defined above) in Line 7, which provides the answers for q with respect to all streams satisfying c. The condition in Line 8 evaluates to true and the evaluation of the algorithm is interrupted by returning q' in Line 9. If a ground query q' providing the answers for the input query q is returned by the algorithm, q can have only the provided answers with respect to all streams for which the input constraints C hold.

3. Join elimination: If a relation being joined does not contribute any attributes in the result of a query and *is not involved into the definition of the query evaluation time*, it can be dropped from the query. This implies join elimination with the relation. (An example illustrating this heuristic follows.)

[4]Two other heuristics, namely join elimination and predicate elimination, apply in this example. These heuristics will be described below.

4. Predicate elimination: If a predicate is known to be always true (i.e., the predicate is implied by the residue $H_c\sigma$) and *is not involved into the definition of the query evaluation time* it can be eliminated from a query.

In Example 27, both join elimination and predicate elimination were possible because the residue $H_c\sigma$ implied the atom and the temporal conditions on the events matched by the atom in the query body. In Example 28, this is also the case but neither join elimination nor predicate elimination is applicable.

Example 28.
Consider the query q of the form

$$\begin{aligned} &items\_007\_is\_interested\_in(auctionID(A), itemID(I)) \leftarrow \\ &\quad i : bid(auctionID(A), itemID(I), bidderID(007), value(V)) \wedge \\ &\quad a : auction(auctionID(A)) \wedge \\ &\quad b(a) \leq b(i) \wedge e(i) \leq e(a) \end{aligned}$$

which has already been introduced in Example 25.

Consider the constraint c of the form

$$\begin{aligned} &\exists_{=1}\, a : auction(auctionID(A)) \wedge \\ &b(a) \leq b(i) \wedge e(i) \leq e(a) \leftarrow \\ &\quad i : bid(auctionID(A), itemID(I), bidderID(B), value(V)) \end{aligned}$$

The constraint c implies that no bid is possible without its respective auction being running. More precisely: Each *bid* event happens during exactly one state *auction* so that the state carries the same value of the attribute $auctionID$ as the event.

Since the state *auction* does not contribute any attributes to the events derived by the query q (in particular, their occurrence time depends only on *bid* events, compare Section 4.2) and since according to the constraint c, the respective *auction* state is always active when a *bid* event arrives, the atom $auction(auctionID(A))$ could be omitted from q if it did not define the evaluation time of q (Join elimination). This would imply the elimination of both conditions $b(a) \leq b(i)$ and $e(i) \leq e(a)$ (Predicate elimination).

Consider Listing 10.1. Since the body of the constraint c subsumes the body of the query q (which is tested in Line 5), c is relevant for q. Since no heuristic applies to q and the residue $H_c\sigma$ of c for q (which is tested in Line 6), $H_c\sigma$ does not contribute anything to the semantic optimization of q. The code of the algorithm in Lines 7–11 is not evaluated. The algorithm continues with the next constraint of the input constraint set C in Line 4.

5. Predicate introduction can be split into the following three heuristics:

a) *Query termination, Garbage collection: If a new predicate specifies the upper bound on the number of queried events (states) during a query evaluation time interval and all of them have already been processed by the respective query, the query can return the answers and terminate before its evaluation time is over. As a consequence, all the data relevant for the evaluation step of the query becomes irrelevant and can be deleted before the end of the query evaluation time.*

b) Scan reduction: A new predicate on a join attribute may reduce the cost of the join.

c) Index introduction: A new predicate on an indexed attribute may allow for a more efficient access method. However, index introduction is hardly applicable to CEP since the data is deleted as soon as possible and therefore not optimized (e.g., indexed).

Example 29.

Consider the query q of the form

$$\begin{aligned} &number\_of\_offered\_items(auctionID(A), number(count(I))) \leftarrow \\ &\quad i : itemDescription(auctionID(A), itemID(I), bidderID(B), value(V)) \wedge \\ &\quad w : Range\ 24h,\ Slide\ 24h \wedge \\ &\quad b(w) \leq b(i) \wedge e(i) \leq e(w) \end{aligned}$$

The query q computes the total number of presented items in each auction per day (24 hours) and returns the results at the end of each day. More exactly: The *itemDescription* events arriving during the current tumbling window w of size 24 hours (consider Section 4.1) are grouped according to the value of the attribute $auctionID$, their number is computed and returned as a new $number\_of\_offered\_items$ event carrying the computed value and the respective auction identifier.

Consider the constraint c of the form

$$\begin{aligned} &\exists_{\leq 10000}\, i\ group\text{-}by\ (\ ) \leftarrow \\ &\quad i : itemDescription(auctionID(A), itemID(I), bidderID(B), value(V)) \wedge \\ &\quad w : Range\ 24h,\ Slide\ 24h \wedge \\ &\quad b(w) \leq b(i) \wedge e(i) \leq e(w) \end{aligned}$$

The constraint c states that there can be at most 10 000 items presented per day (24 hours) in all auctions. The *group-by* statement in the head of c restricts the variables according to the values of which the *itemDescription* events are grouped. In the example above, the number of all *itemDescription* events regardless their data is given. That is why the *group-by* statement is empty. Without *group-by( )*, c would mean that there can be at most 10 000 *itemDescription* events grouped by the values of all their attributes. Since the attributes $itemID$, $bidderID$, and $value$ build the key, there is at most one

itemDescription event grouped by the values of all the attributes. Hence, the *group-by( )* statement is indispensable in the example above.

The body of the constraint c subsumes the body of the query q with the empty unifier σ, the head H_c of c becomes true for q so that q can be modified to q' using H_c as follows:

$$
\begin{aligned}
&number\_of\_offered\_items(auctionID(A), number(count(I))) \leftarrow \\
&\quad i : itemDescription(auctionID(A), itemID(I), bidderID(B), value(V)) \wedge \\
&\quad w : Range\ 24h,\ Slide\ 24h \wedge \\
&\quad b(w) \leq b(i) \wedge e(i) \leq e(w) \wedge \\
&\quad \{\ \exists_{\leq 10000}\, i \ \ group\text{-}by\ (\ )\ \}
\end{aligned}
$$

Thanks to the residue $\{\ \exists_{\leq 10000}\, i \ \ group\text{-}by\ (\ )\ \}$, as soon as 10 000 events matching the first atom in the body of q have arrived, the CEP engine can stop searching in the buffer or waiting for the *itemDescription* events, return all answers, and delete all irrelevant *itemDescription* events even before the current window w is over.

In order to differentiate between the original body of q and the residue $H_c\sigma$, the latter is written in curly braces. Without this differentiation, the modified query q' would match the stream only if the window w contains at most 10 000 *itemDescription* events. This is, of course, always the case but the termination of q' would be possible only at the end of the window w.

Consider Listing 10.1. Since the body of the constraint c subsumes the body of the query q (which is tested in Line 5), c is relevant for q. Since the fifth heuristic applies to q and the residue $H_c\sigma$ of c for q (which is tested in Line 6), $H_c\sigma$ contributes to the semantic optimization of q. The function $applyHeuristics(q, H_c)$ modifies q to q' using $H_c\sigma$ and returns q' (as defined above) in Line 7. Since q' is neither $\bot$ nor provides the answers for q, the condition in Line 8 evaluates to false and q' is appended to the set of alternative semantically optimized queries S_q for q in Line 10. Since other constraints of the input constraint set C can be applicable to q', the algorithm is called recursively on q' and C and the results of this call are also appended to the set S_q in Line 10. (An example of repeated application of multiple constraints to the same query is given below.)

6. Join introduction: It may be advantageous to add a join with an additional relation, if that relation is relatively small compared to the original relations as well as highly selective. In database systems (but usually not in CEP, as explained above), this is even more appealing if the join attributes are indexed. *In CEP, join introduction can contribute to query termination before the query evaluation time interval is over if the added atom matches an event arriving after all queried events. In this case the query can compute the answers earlier, i.e. the queried events become earlier irrelevant and can be garbage*

collected. Example 30 illustrates early query termination and garbage collection enabled by join introduction.

Example 30.
Consider the query q of the form

$$\begin{aligned}&number\_of\_offered\_items(auctionID(A), number(count(I))) \leftarrow\\ &\quad i : itemDescription(auctionID(A), itemID(I), bidderID(B), value(V)) \wedge\\ &\quad w : Range\ 24h,\ Slide\ 24h \wedge\\ &\quad b(w) \leq b(i) \wedge e(i) \leq e(w) \wedge\\ &\quad \{\ \exists_{\leq 10000}\, i\ \ group\text{-}by\ (\ )\ \}\end{aligned}$$

which is the result of the predicate introduction into the query in Example 29.

Consider the constraint c of the form

$$\begin{aligned}&\exists_{=1}\, j : auctionEnd(auctionID(A)) \wedge\\ &e(i) \leq b(j) \leftarrow\\ &\quad i : itemDescription(auctionID(A), itemID(I), bidderID(B), value(V))\end{aligned}$$

The constraint c states that there is exacltly one *auctionEnd* event arriving after all *itemDescription* events. The *auctionEnd* event has the same value of the attribute $auctionID$ as the *itemDescription* events.

The body of c subsumes the body of q with the empty unifier σ. The head H_c of c is the residue of c for q. It holds for q and is appended to the body of q. The result is q' of the form:

$$\begin{aligned}&number\_of\_offered\_items(auctionID(A), number(count(I))) \leftarrow\\ &\quad i : itemDescription(auctionID(A), itemID(I), bidderID(B), value(V)) \wedge\\ &\quad w : Range\ 24h,\ Slide\ 24h \wedge\\ &\quad b(w) \leq b(i) \wedge e(i) \leq e(w) \wedge\\ &\quad \{\ \exists_{\leq 10000}\, i\ \ group\text{-}by\ (\ )\ \} \wedge\\ &\quad \{\ \exists_{=1}\, j : auctionEnd(auctionID(A)) \wedge\\ &\quad e(i) \leq b(j)\ \}\end{aligned}$$

The join introduction with only one event *auctionEnd* allows to terminate the query as soon as the event arrives even if the query evaluation time window w is not over and even if less then 10 000 *itemDescription* events have arrived. This heuristic also effects storage minimization since the *itemDescription* events can become irrelevant earlier and can be deleted. Note that the *auctionEnd* event does not have to be saved. It can be evaluated by the query on the fly. But if there were more then one event, they would have to be saved (provided they contribute to the query optimization).

Semantic optimization of CEP queries can surely rely on these slightly modified heuristics for the semantic optimization of database queries. No additional heuristics are required for the semantic optimization of CEP. In other words, the same algorithm can be used to optimize both database and CEP queries. This means that also combined queries (i.e., queries extracting some knowledge from a database and combining it with stream processing) as needed in many applications, can be optimized by the algorithm in Listing 10.1.

Summarizing this section, we would like to emphasize that the algorithm in Listing 10.1 is not only generic (i.e., language independent) but it also allows combined queries of a database and an event stream as needed in many applications like the emergency detection in a metro station described in Section 2.1.2. For each query q and a set of constraints C, the algorithm returns one of the following results:

1. $\perp$, if q is unsatisfiable with respect to all streams satisfying C
2. The ground query q' providing the answers for q with respect to all streams satisfying C
3. $\emptyset$, if no constraint in C is relevant for q and contributes to the semantic optimization of q
4. The (not empty) set S_q of alternative semantically optimized queries for q

As illustrated by examples above, in the first two cases, q is not evaluated. In the third case, q cannot be semantically optimized using C. Therefore, the unchanged initial query q is evaluated. The fourth case is the most interesting. If the set S_q of alternative semantically optimized queries for q contains only one query q', the query q' is evaluated instead of the original query q. If the set S_q contains more than one query, one of them must be chosen and processed instead of q. To this end, the evaluation costs of all queries in S_q are determined using cost models, and one of them with the lowest (estimated) evaluation cost is chosen and processed instead of q. The algorithm specifying the query choice is a subject for future work.

The case in which the algorithm in Listing 10.1 returns the set of multiple alternative semantically optimized queries for the initial query is the most general and usual case. We illustrate it by the following example.

Example 31.
For the query q (introduced in Example 29):

$$number\_of\_offered\_items(auctionID(A), number(count(I))) \leftarrow$$

$$
\begin{aligned}
&i : itemDescription(auctionID(A), itemID(I), bidderID(B), value(V)) \wedge \\
&w : Range\ 24h,\ Slide\ 24h \wedge \\
&b(w) \leq b(i) \wedge e(i) \leq e(w)
\end{aligned}
$$

and the set of constraints C (presented in Examples 29 and 30 respectively):

$$
\begin{aligned}
&\exists_{\leq 10000}\, i\ group\text{-}by\ (\ ) \leftarrow \\
&\quad i : itemDescription(auctionID(A), itemID(I), bidderID(B), value(V)) \wedge \\
&\quad w : Range\ 24h,\ Slide\ 24h \wedge \\
&\quad b(w) \leq b(i) \wedge e(i) \leq e(w)
\end{aligned}
$$

$$
\begin{aligned}
&\exists_{=1}\, j : auctionEnd(auctionID(A)) \wedge \\
&e(i) \leq b(j) \leftarrow \\
&\quad i : itemDescription(auctionID(A), itemID(I), bidderID(B), value(V))
\end{aligned}
$$

the algorithm in Listing 10.1 returns the set S_q of three alternative semantically optimized queries q' (as explained above):

$$
\begin{aligned}
&number\_of\_offered\_items(auctionID(A), number(count(I))) \leftarrow \\
&\quad i : itemDescription(auctionID(A), itemID(I), bidderID(B), value(V)) \wedge \\
&\quad w : Range\ 24h,\ Slide\ 24h \wedge \\
&\quad b(w) \leq b(i) \wedge e(i) \leq e(w) \wedge \\
&\quad \{\ \exists_{\leq 10000}\, i\ \ group\text{-}by\ (\ )\ \}
\end{aligned}
$$

$$
\begin{aligned}
&number\_of\_offered\_items(auctionID(A), number(count(I))) \leftarrow \\
&\quad i : itemDescription(auctionID(A), itemID(I), bidderID(B), value(V)) \wedge \\
&\quad w : Range\ 24h,\ Slide\ 24h\ \wedge \\
&\quad b(w) \leq b(i) \wedge e(i) \leq e(w)\ \}\ \wedge \\
&\quad \{\ \exists_{=1}\, j : auctionEnd(auctionID(A))\ \wedge \\
&\quad e(i) \leq b(j)\ \}
\end{aligned}
$$

$$
\begin{aligned}
&number\_of\_offered\_items(auctionID(A), number(count(I))) \leftarrow \\
&\quad i : itemDescription(auctionID(A), itemID(I), bidderID(B), value(V)) \wedge \\
&\quad w : Range\ 24h,\ Slide\ 24h \wedge \\
&\quad b(w) \leq b(i) \wedge e(i) \leq e(w) \wedge \\
&\quad \{\ \exists_{\leq 10000}\, i\ \ group\text{-}by\ (\ )\ \} \wedge \\
&\quad \{\ \exists_{=1}\, j : auctionEnd(auctionID(A))\ \wedge \\
&\quad e(i) \leq b(j)\ \}
\end{aligned}
$$

One of $q' \in S_q$ will be chosen by the query choice algorithm and evaluated instead of q.

10.2 Termination, Complexity, and Correctness

In the following, the properties of the algorithm presented in Section 10.1 are formally described. They are termination, complexity, and correctness. They all are derived from the pseudo code of the algorithm in Listing 10.1.

Termination

Proposition 2. *The algorithm terminates.*

Proof. One query and a finite set of constraints are the input of the algorithm. Both queries and constraints are hierarchical [50], i.e. they allow no recursion at all. Both queries and constraints contain a finite number of literals and a finite number of conditions on the events (states) matched by the literals in their bodies. Therefore, the algorithm computing the subsumption of CEP queries, terminates and returns a finite unifier or $false$. Hence, there is only one residue of each constraint for a query. There are six semantic optimization heuristics which are applied to a query and its residue. The application of each heuristic terminates and always returns a finite modified query. Since the number of constraints is finite, the algorithm can be finitely often recursively called for an input query.

□

Complexity

Let $|C|$ denote the number of constraints in the input constraint set C. Let S be the complexity of the subsumption of two CEP queries. Let H be the costs for the application of a heuristic to a query and a residue.

Proposition 3. *The complexity of the algorithm is in* $O(2^{|C|} \cdot |C| \cdot S \cdot H)$.

Proof. A constraint can be applied to a query only once and there is only one residue of a constraint for a query. Hence, the set S_q of semantically optimized queries returned by the algorithm for the input query q contains at most $2^{|C|} - 1$ elements (because of the power set construction of the elements of the input constraint set C without the consideration of the empty set). The algorithm is recursively called, first, for the input query q and for each of $|C|$ constraints in the input constraint set C and, second, for each of at most $2^{|C|} - 1$ semantically optimized queries q' for q and each of $|C|$ constraints. All in all, there can be at most $2^{|C|} \cdot |C|$ recursive calls. There are six semantic optimization heuristics. In the worst case, all of them are applied to a query and a residue. The complexity of the algorithm is in $O(2^{|C|} \cdot |C| \cdot S \cdot 6 \cdot H) \in O(2^{|C|} \cdot |C| \cdot S \cdot H)$.

□

The algorithm can be improved to consider only those constraints for each semantically optimized query which have not been considered for it yet. However, the optimization of the algorithm is out of the scope of this report and will be provided later.

Correctness

Proposition 4. *The algorithm is correct, i.e. its result is a query which is semantically equivalent to the original query.*

The proof of this statement uses the model theories of an event query language and a constraint language as well as the subsumption of CEP queries as black boxes. (Remember, the algorithm is language independent.) We consider StreamLog as an example of an event query language and ESCL as an example of a constraint language in the following. Consider [52] for the definition of the interpretation of a StreamLog query and Definition 28 for the definition of the interpretation of an ESCL constraint.

Proof. Let I be an interpretation and c be an ESCL constraint of the form $H_c \leftarrow B_c$. Assume c is satisfied in I, formally $I = D, \sigma, \tau \models c$. Let q be a query of the form $H_q \leftarrow B_q$. Let q be also satisfied in I, i.e. $I = D, \sigma, \tau \models q$, and return the answer set of all $\sigma_\tau(H_q)$ such that for all τ there is a σ with $D, \sigma, \tau \models B_q$. In other words, the head of q is an atom H_q for constructing new, derived events. This construction uses the grounding unifier σ_τ obtained while matching the query body B_q against events referenced by τ to replace variables with values. The application of the grounding unifier σ_τ to the atom H_q returns a ground atom. The atom is annotated with an occurrence time interval and represents an event derived by the query q.

Assume B_c subsumes B_q with the unifier σ (we assume the subsumption of two CEP queries to be correct). Then B_q implies $B_c\sigma$. $B_c\sigma$ implies $H_c\sigma$ by the definition of a constraint. Therefore, B_q implies $H_c\sigma$ ($\star$) since implication is transitive. The following cases are possible:

1. If result by transformation or result by contradiction applies to $H_c\sigma$ and q, $H_c\sigma$ provides the answers or the absence of answers for q. Because of ($\star$), the result implied by $H_c\sigma$ is the only result q can have with respect to all streams S satisfying c.

2. If predicate introduction or join introduction applies to $H_c\sigma$ and q, $H_c\sigma$ is appended to the query body and the algorithm returns the modified query $q' = H_q \leftarrow B_q \wedge \{H_c\sigma\}$. Because of ($\star$) it is obvious that q' returns the same results as q with respect to all streams S satisfying the constraint c.

3. If predicate elimination or join elimination applies to $H_c\sigma$ and q, there is a condition or an atom x in the body B_q of q which is implied by $H_c\sigma$. Because of ($\star$) and since implication is transitive, B_q without x implies x. Therefore, x can be omitted from B_q without effecting the semantics of q with respect to all streams S satisfying c.

□

The algorithm may not use trivial constraints because otherwise the third point in the above proof does not hold any more. (More exactly: B_c must subsume B_q without x in order to be able to drop x from B_q without effecting the semantics of q). Consider Example 32 illustrating how trivial constraints are processed by the algorithm and lead to false query transformations.

Example 32.
Let q be the query of the form $q(a) \leftarrow p(a)$. Let c be the trivial constraint of the form $p(a) \leftarrow p(a)$. The body of c subsumes the body of q with the empty unifier σ. The residue $H_c\sigma = H_c = p(a)$ holds for q with respect to all streams satisfying c. Therefore, the algorithm drops the only atom of the body of q and returns the modified query q' which is a ground fact $q(a) \leftarrow$. This is not true however.

Remember that the application of heuristics to a query q and its residue $H_q\sigma$ determines whether the residue contributes to the semantic optimization of the query. Even if no heuristic applies to them, $H_q\sigma$ can be appended to the body of q without changing its semantics as argued above.

Remember also that if $H_c\sigma$ is appended to the body of q, $H_c\sigma$ does not become a part of the query body in the sense that q returns answers if its original body B_q matches the stream, and not the modified body $B_q \wedge H_c\sigma$. As proven above, $H_c\sigma$ holds for all streams for which B_q holds, but the evaluation of $H_c\sigma$ as a part of the query body can prohibit early termination of q. To differentiate between B_q and $H_c\sigma$, the latter is written in curly braces. Consider Example 29 illustrating this.

Chapter 11

Conclusions

Summarizing this report, we briefly describe the main features of those parts of the approach which are completely elaborated at the moment. Compare the description of the entire approach in Section 1.1 with Figure 11.1. The results of this work are the following:

1. Event Stream Constraint Language (ESCL)

   ESCL is a novel declarative first-order logic language tailored to CEP.

   ESCL constraints are short but expressive logic formulas capturing causal, cardinality, temporal, data, and spatial constraints on events, application states, and data saved in a conventional database.

   ESCL has strong formal foundations. Its declarative semantics is a kind of Herbrand interpretation extended to deal with temporal relations and event and state identifiers. The model relation between an interpretation and an ESCL constraint is defined very similarly to the Tarski model relationship between an interpretation and a formula.

   Since ESCL is developed for the semantic optimization of CEP, the operational

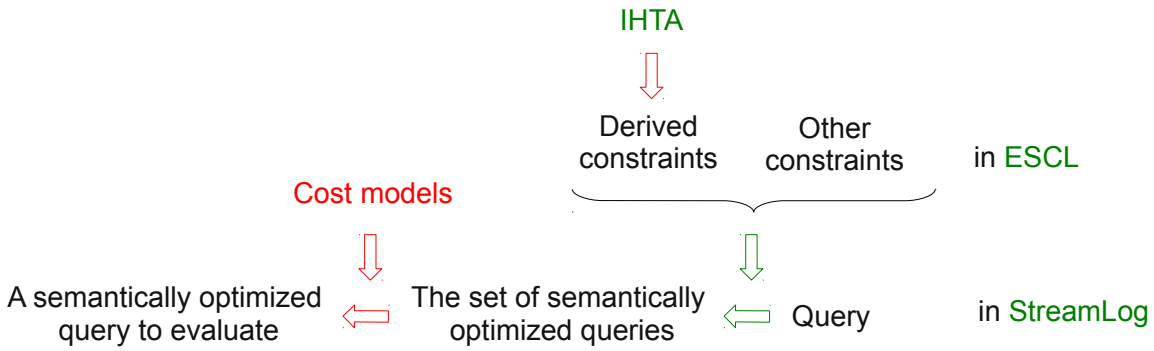

Figure 11.1: Overview of the Approach

semantics of the language is defined by the algorithm semantically rewriting CEP queries with respect to ESCL constraints. The algorithm is an adaption of the residue method [31] for semantic optimization of database systems to CEP. The algorithm is proven to terminate, to have exponential complexity in the worst case, and be correct with respect to the declarative semantics of ESCL. However, ESCL is not restricted to the semantic optimization of CEP. It could also be used for other purposes, for example, for stream verification.

Finally, ESCL is modularly defined and extensible.

2. Instantiating Hierarchical Timed Automata (IHTA)

   IHTA is a novel automata-based model of CEP application semantics which will be used as metadata for the semantic optimization of CEP. To the best of our knowledge, IHTA are the only kind of automata used in this way. But they are not restricted to this purpose. Like other timed automata, IHTA can be used for modeling dynamic event-driven systems and stream verification.

   Like all automata, IHTA capture the notion of state and state transitions very naturally. IHTA are independent from the language specifying their transition labels. However, the expressiveness of IHTA and the kinds of constraints captured by IHTA depend on the language specifying their transition labels.

   In this report, (a variant of) the event query language StreamLog is suggested as an example of such a language allowing access to event and state data, the current time point, the beginning and the end of event occurrence time, and the number of instances of IHTA. IHTA using StreamLog for formulation transition labels express multiple involved temporal, causal, cardinality, and data relations between events and states in a readable way. However, they do not express the entire application semantics, for example, spatial constraints are not expressed by IHTA using StreamLog at all. IHTA and StreamLog are designed to be comprehensive for the user, their expressiveness is limited for this reason. ESCL constraints complete IHTA with additional semantics.

   IHTA are non-deterministic and able to work with events with the same occurrence time.

   IHTA are hierarchical and, therefore, modular which implies abstraction, readability, (ex-) changeability, and reuse of the modules. In contrast to the existing hierarchical models, IHTA can represent an arbitrary number of concurrent processes as required in many CEP applications, like, for example, the online auction use case presented in Section 2.1.1. This is realized by instantiation of module specifications, i.e. of nested IHTA.

3. Propagation of ESCL cardinality constraints with respect to CEP queries

   Event-based systems work with large amount of event data continuously arriving on potentially infinite event streams. Under these circumstances, garbage collection is often indispensable. Hence, some base relations possibly became incomplete in the meantime. As a consequence, CEP views cannot be led back to their base events (states) without lost of results in some cases. Therefore, in order to semantically optimize CEP queries which are based on CEP views, constraints have to be defined for these views too and not only for their base events and states.

   To reduce the number of constraints which must be defined by the user, only the ECSL constraints for base events and states are manually specified and automatically propagated to CEP views with respect to the queries deriving the views. In this report, the propagation of ESCL cardinality constraints with respect to StreamLog queries is formally defined. But the approach is language independent in general. The propagation of ESCL causal, temporal, and data constraints is to be investigated.

   Propagation of ESCL cardinality constraints is defined in two steps: The first step considers the case in which the validity time intervals of cardinality constraints coincide with the evaluation time interval of the query with respect to which the constraints are propagated. This algorithm is also applicable to database constraints and queries since in database systems there is usually no dependency from time, in contrast to CEP. The second step treats the case in which the validity time intervals of cardinality constraints differ from the evaluation time interval of the query with respect to which the constraints are propagated. This algorithm is typical for CEP.

   Query evaluation time and constraint validity time can be specified by a tumbling window, a sliding window or a time interval. The time interval can be repeating or non-repeating. It covers (relative timer) events and states, an unbounded window, a now window, and a time interval the bounds of which are specified by functions.

   The cardinality specifications of propagated constraints may be complex, i.e. be an arbitrary nested conjunction or disjunction of simple cardinality specifications. The body of the query with respect to which the constraints are propagated may be an arbitrary nested conjunction or disjunction of literals.

   The base cases of cardinality constraint propagation are defined in the form of tables. The inductive cases are defined by algorithms. Each case is illustrated by examples.

4. Subsumption of CEP conjunctive queries

   Subsumption of CEP queries is required for multi-query (semantic) optimization, for the propagation of constraints which are rules, and for the determination of the

relevance of a constraint for a query which is the base of the algorithm semantically optimizing CEP queries.

In this report, the subsumption algorithm for Horn clauses proposed in [56] is adapted to CEP queries. This adaption is necessary because CEP queries are different from clauses, in particular temporal conditions, extended quantifiers, negative atomic queries, and (data or temporal) conditions are taken into consideration by the adapted algorithm deciding subsumption of ESCL or StreamLog conjunctive queries.

The termination and complexity of the adapted algorithm are investigated. Like the algorithm in [56], the adapted algorithm is polynomial in the best case and exponential in the worst case. For the considered core fragments of ESCL and StreamLog, the adapted algorithm is proven to be correct and complete with respect to the declarative semantics of the languages.

5. Semantic rewriting of CEP queries with respect to ESCL constraints

   The algorithm semantically rewriting CEP queries with respect to ESCL constraints is an adaption of the residue method semantically rewriting database queries with respect to database constraints which was proposed in [31]. The method is static and rather expensive. The method is proven to be exponential in the worst case in this report.

   Because of its complexity, the residue method is not wide-spread in database systems where static (semantic) query optimization must be efficient for the following reasons: Database queries are usually ad hoc, data is completely available when queries are put, and hence, all query answers are usually computed only once and expected (almost) immediately. However, static (semantic) optimization of CEP queries may be inefficient if it happens before events are available and queries can be evaluated. Besides, repeated evaluation of CEP queries over a long period of time is worth their expensive static semantic optimization. In other words, static semantic query optimization in CEP has greater potential than in database systems because querying a stream is fundamentally different from querying a database.

   The adapted algorithm is based on the same heuristics as the residue method. However, these heuristics are slightly modified to deal with temporal relations typical for CEP queries. The adapted algorithm and heuristics are illustrated by examples. The algorithm is proven to terminate, to have exponential complexity in the worst case, and to be correct with respect to the declarative semantics of ESCL and StreamLog.

Chapter 12

Future Work

This chapter gives an overview of the following future research directions which are based on this report:

1. Development of the algorithm transforming IHTA into a set of ESCL constraints (Section 12.1),
2. Development of the algorithm propagating causal, temporal, and data constraints with respect to CEP queries (Section 12.2),
3. Development of the algorithm deriving constraints from CEP queries independently from constraints (Section 12.3),
4. Elaboration of cost models according to which for each CEP query one of its alternatively semantically optimized queries is chosen to evaluate instead of the initial query (Section 12.4), and finally
5. Implementation and experimental evaluation of the approach, and development of the visual editor facilitating its use (Section 12.5).

12.1 Derivation of Constraints from IHTA

12.1.1 Motivation

IHTA (Chapter 6) are a presentation of constraints in a readable way. In particular, the constraints determined by the application workflow are expressed by the automata in a concise and comprehensive for the user way. However, the algorithm semantically rewriting CEP queries with respect to the application semantics (Chapter 10) works on metadata

expressed as a set of constraints, not automata. Therefore, IHTA will be transformed into a set of ESCL constraints. The transformation algorithm and the proof of its correctness with respect to the formal semantics of IHTA are subjects of future work. This section describes the intuition behind this transformation and gives some examples.

Consider the IHTA in Figure 6.1. Many ESCL constraints can be automatically derived from the automata. Indeed, it is known which, how many, when, in which order, and under which circumstances events will arrive and states will be reached. More concrete: All paths from a start state till an end state of an IHTA can be computed and the strongly connected components involved into these paths can be recognized. As a result the knowledge about:

1. The number,
2. The temporal order,
3. The causal dependencies, and finally
4. The dependencies between the data

of events and states is yielded. More involved constraints are specified in ESCL directly and not derived from IHTA. Examples of such constraints are given in Listing 7.4 and Listing 7.6. Spatial constraints are not expressed by IHTA. Remember that IHTA are designed to be readable and as a consequence their expressiveness is limited.

12.1.2 Cardinality Constrains

Consider the set of cardinality constraints in Listing 7.1. They all can be automatically derived from the IHTA *ItemOffer(auctionID(A), itemID(I))* in Figure 6.1 according to the following rules:

- If n is the upper bound, the lower bound, or the exact number of the events (states) matching the atom q for each path from a start till an end state of the IHTA, the cardinality constraint **IN ANY CASE** $\exists_{\theta n}\, q$ **END** is derived where $\theta \in \{\leq, \geq, =\}$ respectively. The constraints in Lines 33, 31, and 35 in Listing 7.1 are the respective examples.
- If both the lower bound n and the upper bound m of the number of events (states) matching the atom q in each path is known, these bounds are combined in a single cardinality constraint by means of logic $\wedge$, i.e. the constraint **IN ANY CASE** $\exists_{\geq n \wedge \leq m}\, q$ **END** is yielded.

- If different paths have different numbers of the events (states) matching the atom q, this knowledge is combined within a single cardinality constraint by means of logic $\vee$, see the example in Lines 9–11 in Listing 7.1.

12.1.3 Causality Constraints

Assume there is a state s in the IHTA such that all paths from s to an end state of the IHTA contain at least one event (state) matching the atom q. Let c be the cardinality specification of q from s till an end state of the IHTA. If s is not a start state of the IHTA, all event sequences S leading to s cause the events (states) matching q. The constraint **IF** S **THEN** $\exists_c\, q$ **END** is yielded. The constraint in Lines 37–71 in Listing 7.1 is a causality (and cardinality) constraint derived in this way.

12.1.4 Temporal Constraints

After all paths from a start to an end state of IHTA are analyzed, the order of events and states is known. In particular, if there is a strongly connected component in IHTA with the set of events and states S involved in it, then there is the set of events and states B appearing on a path before reaching S and there is the set of events and states A appearing on a path after leaving S forever. In this case each event (state) in B appears before each event (state) in S and A and each event (state) in S appears before each event (state) in A. This idea is exploited by the derivation of the temporal constraints in Listing 7.2.

12.1.5 Data Constraints

The class of data constraints which can be derived from the IHTA is the functional dependencies between the unique data attributes of nested IHTA differentiating between the instances of their respective IHTA. Remember that these data attributes are marked by the key word *unique* in the transition labels of IHTA. Consider, for example, the IHTA in Figure 6.1. The IHTA *BidderEnrollment(auctionID(A), enrollID(E))* and the IHTA *ItemOffer(auctionID(A), itemID(I))* are nested into the IHTA *Auction(auctionID(A))* where the values of the variables A, E, and I are unique, i.e. they identify the instances of their respective states. Because of these reasons, in all events and states of the IHTA *BidderEnrollment(auctionID(A), enrollID(E))* the value of the variable A is functionally dependent from the value of the variable E and, analogously, in all events and states of the IHTA *ItemOffer(auctionID(A), itemID(I))* the value of the variable A is functionally

dependent from the value of the variable I. Listing 7.5 shows some data constraints expressing the functional dependency of A from I. All the other constraints are derived from the IHTA analogously.

12.2 Propagation of Causal, Temporal, and Data Constraints

Analogously to the propagation of cardinality constraints (Chapter 8), causal, temporal, and data constraints will be propagated with respect to CEP queries. The StreamLog rule in Listing 12.1 will illustrate the intuition behind the propagation of these constraints. The rule identifies the items profitably sold during an auction, i.e. their selling price is higher than the double of their start price.

Listing 12.1: StreamLog rule identifying profitably sold items during an auction

```
1 auction(auctionID(A)):
2   profitablySoldItem(auctionID(A), itemID(I)) ←
3     itemDescription(auctionID(A), itemID(I), bidderID(B1), value(V1)) ∧
4     sell(auctionID(A), itemID(I), bidderID(B2), value(V2)) ∧
5     V2 > 2 · V1
```

12.2.1 Causal Constraints

Consider the IHTA in Figure 6.1. Whether there is at least one item description during an auction depends on whether at least two bidders are enrolled for the auction, i.e. *item description* events are caused by at least two *bidder enrollment* events. Whether an item is sold depends on whether there is at least one bid for it, i.e. *sell* events are caused by *bid* events. However, *profitablySoldItem* events are caused not only by *bidder enrollment* and *bid* events but also by the price difference of respective *item description* and *sell* events. Without the condition in Line 5 of Listing 12.1, the causal constraint could be derived for the *profitablySoldItem* events.

12.2.2 Temporal Constraints

Since the occurrence time of the derived event $profitablySoldItem$ comprises the occurrence time intervals of the events it was derived from (i.e., *itemDescription* and *sell*) and since description of an item always precedes its sell, all events and states happening

before *itemDescription* events precede *profitablySoldItems* events and all events and states following *sell* events happen after *profitablySoldItems* events.

12.2.3 Data Constraints

Since there is a functional dependency of the auction identifier from the item identifier in all *itemDescription* and *sell* events and since both respective attributes are carried by the derived *profitablySoldItem* events, this functional dependency holds also for the *profitablySoldItem* events.

12.3 Derivation of Constraints from CEP Queries

Some constraints can be derived from CEP queries independently from constraints. Consider the following example described in detail in Chapter 5.

Listing 12.2: StreamLog rule computing the number of bids per item during an auction

```
1 auction(auctionID(A)):
2   bidNumber(auctionID(A), itemID(I), number(count(A,I))) ←
3     bid(auctionID(A), itemID(I), bidderID(B), value(V))
```

The value of the attribute *number* of the derived events depends on the number of bids per item in each auction. The is exactly one *bidNumber* event for each item with bids during an auction. Therefore the following constraint can be derived:

Listing 12.3: Constraint derived from the rule in Listing 12.2.

```
1 WHILE auction(auctionID(A))
2 LET
3   IF ∃=k bid(auctionID(A), itemID(I), bidderID(B), value(V))
4      group−by {A,I}
5   THEN ∃=1 bidNumber(auctionID(A), itemID(I), number(N)) ∧ N = k
6   END
7 END
```

12.4 Cost Models

The algorithm semantically rewriting CEP queries with respect to constraints (Chapter 10) returns a set of alternatively semantically optimized queries for each original

query in general. Consider Example 31. Cost models are used to estimate the evaluation costs of these queries to choose the cheapest one of them to evaluate instead of the original query. The elaboration of these cost models as well as the algorithm choosing a query to evaluate are subjects for future work.

12.5 Implementation, Experimental Evaluation, and Visual Editor

Finally, the whole method will be implemented in Java and its profit will be experimentally demonstrated. A visual editor will be built to facilitate the use of the approach. The user will specify the application semantics by IHTA and ESCL constraints and put CEP queries to evaluate. IHTA will be automatically transformed into (normalized) sets of ESCL constraints. (The manually specified ESCL constraints will be also automatically normalized). The entire set of the constraints will be further used for the semantic rewriting of the CEP queries. The user will choose cost models out of the list of cost models supported by the system. According to these cost models, for each original query, a semantically optimized query with the lowest estimated cost will be chosen to evaluate instead of the initial query. Finally, the answers for the optimized queries will be computed and printed out.

Bibliography

[1] Yasmina Abdeddaïm, Eugene Asarin, and Oded Maler. Scheduling with Timed Automata. *Theor. Comput. Sci.*, 354:272–300, March 2006.

[2] Parosh Aziz Abdulla and Aletta Nylén. Timed Petri Nets and BQOs. In *Applications and Theory of Petri Nets*, volume 2075 of *Lecture Notes in Computer Science*, pages 53–70. Springer, 2001.

[3] James F. Allen. Maintaining Knowledge about Temporal Intervals. *Commun. ACM*, 26:832–843, November 1983.

[4] Rajeev Alur. Timed Automata. In *Computer Aided Verification*, volume 1633 of *Lecture Notes in Computer Science*, pages 8–22. Springer, 1999.

[5] Rajeev Alur and David L. Dill. Automata For Modeling Real-Time Systems. In *Proc. of the* 17^{th} *Int. Colloquium on Automata, Languages and Programming*, ICALP '90, pages 322–335. Springer, 1990.

[6] Rajeev Alur and David L. Dill. A Theory of Timed Automata. *Theor. Comput. Sci.*, 126:183–235, April 1994.

[7] Rajeev Alur, Sampath Kannan, and Mihalis Yannakakis. Communicating Hierarchical State Machines. In *Automata, Languages and Programming*, volume 1644 of *Lecture Notes in Computer Science*, pages 169–178. Springer, 1999.

[8] Rajeev Alur and P. Madhusudan. Decision Problems for Timed Automata: A Survey. In *Formal Methods for the Design of Real-Time Systems*, volume 3185 of *Lecture Notes in Computer Science*, pages 1–24. Springer, 2004.

[9] Rajeev Alur and Mihalis Yannakakis. Model Checking of Hierarchical State Machines. *SIGSOFT Softw. Eng. Notes*, 23:175–188, November 1998.

[10] Rajeev Alur and Mihalis Yannakakis. Model Checking of Message Sequence Charts. In *CONCUR'99 Concurrency Theory*, volume 1664 of *Lecture Notes in Computer Science*, pages 114–129. Springer, 1999.

[11] Arvind Arasu and Jennifer Widom. Resource Sharing in Continuous Sliding-Window Aggregates. In *Proc. Int. Conf. on Very Large Database*, pages 336–347. Morgan Kaufmann, 2004.

[12] Ron Avnur and Joseph M. Hellerstein. Eddies: Continuously Adaptive Query Processing. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 261–272. ACM, 2000.

[13] Shivnath Babu, Utkarsh Srivastava, and Jennifer Widom. Exploiting k-Constraints to Reduce Memory Overhead in Continuous Queries over Data Streams. *ACM Transactions on Database Systems*, 29(3):545–580, 2004.

[14] Leo Bachmair, Ta Chen, C. R. Ramakrishnan, and I. V. Ramakrishnan. Subsumption Algorithms Based on Search Trees. In *Trees in Algebra and Programming*, volume 1059 of *Lecture Notes in Computer Science*, pages 135–148. Springer, 1996.

[15] Danièle Beauquier. Muller Automata and Bi-infinite Words. In *Fundamentals of Computation Theory*, FCT '85, pages 36–43. Springer, 1985.

[16] Johan Bengtsson and Wang Yi. Timed Automata: Semantics, Algorithms and Tools. In Jörg Desel, Wolfgang Reisig, and Grzegorz Rozenberg, editors, *Lectures on Concurrency and Petri Nets*, volume 3098 of *Lecture Notes in Computer Science*, pages 87–124. Springer, 2004.

[17] Marco Bettelini, Nikolaus Seifert, and François Bry. Innovatives Sicherheitssystem für U-Bahn-Stationen. *Information Management und Consulting,*, 26:71–78, 2011.

[18] Marilyn Bohl and Maria Rynn. *Tools for Structured and Object-Oriented Design.* Prentice Hall, 2008.

[19] Benedikt Bollig and Dietrich Kuske. Distributed Muller Automata and Logics. *Information and Computation*, 206, 2008.

[20] Anthony J. Bonner and Michael Kifer. Transaction Logic Programming (or, A Logic of Procedural and Declarative Knowledge), 1995.

[21] Grady Booch, Robert Maksimchuk, Michael Engle, Bobbi Young, Jim Conallen, and Kelli Houston. *Object-Oriented Analysis and Design with Applications,* 3^{rd} *ed.* Addison-Wesley Professional, 2007.

[22] Grady Booch, James Rumbaugh, and Ivar Jacobson. *Unified Modeling Language User Guide,* 2^{nd} *ed.* Addison-Wesley Professional, 2005.

[23] Irina Botan, Gustavo Alonso, Peter M. Fischer, Donald Kossmann, and Nesime Tatbul. Flexible and Scalable Storage Management for Data-intensive Stream Processing. In *Proc. Int. Conf. on Extending Database Technology*, pages 934–945. ACM, 2009.

[24] Patricia Bouyer and François Laroussinie. *Model Checking Timed Automata*, pages 111–140. ISTE, 2010.

[25] Marius Bozga, Conrado Daws, Oded Maler, Alfredo Olivero, Stavros Tripakis, and Sergio Yovine. Kronos: A Model-checking Tool for Real-time Systems. In *Computer Aided Verification*, volume 1427 of *Lecture Notes in Computer Science*, pages 546–550. Springer, 1998.

[26] François Bry and Michael Eckert. Rule-Based Composite Event Queries: The Language $XChange^{EQ}$ and its Semantics. In *Proc. Int. Conf. on Web Reasoning and Rule Systems*, volume 4524 of *Lecture Notes in Computer Science*, 2007.

[27] François Bry and Michael Eckert. Temporal Order Optimizations of Incremental Joins for Composite Event Detection. In *Proc. Int. Conf. on Distributed Event-Based Systems*. ACM, 2007.

[28] François Bry and Michael Eckert. On Static Determination of Temporal Relevance for Incremental Evaluation of Complex Event Queries. In *Proc. Int. Conf. on Distributed Event-Based Systems*, volume 332, pages 289–300. ACM, 2008.

[29] François Bry, Norbert Eisinger, Thomas Eiter, Tim Furche, Georg Gottlob, Clemens Ley, Benedikt Linse, Reinhard Pichler, and Fang Wei. Foundations of Rule-Based Query Answering. In *Reasoning Web, Third International Summer School 2007*, volume 4636 of *LNCS*. Springer-Verlag, 2007.

[30] Julius Richard Büchi. On a Decision Method in Restricted Second Order Arithmetic. In *Proc. Int. Congress on Logic, Method, and Philosophy of Science*, pages 1–12. Standford University Press, 1960.

[31] Upen S. Chakravarthy, John Grant, and Jack Minker. Logic-based Approach to Semantic Query Optimization. volume 15, pages 162–207. ACM, 1990.

[32] Ashok K. Chandra, Dexter C. Kozen, and Larry J. Stockmeyer. Alternation. *J. ACM*, 28:114–133, January 1981.

[33] Ashok K. Chandra and Larry J. Stockmeyer. Alternation. In *Proc.* 17^{th} *IEEE Symp. on Foundations of Computer Science*, pages 98–108, 1976.

[34] Chin-Liang Chang and Richard Char-Tung Lee. *Symbolic Logic and Mechanical Theorem Proving*. Academic Press, 1973.

[35] Jianjun Chen, David J. DeWitt, and Jeffrey F. Naughton. Design and Evaluation of Alternative Selection Placement Strategies in Optimizing Continuous Queries. In *Proc. Int. Conf. on Data Engeneering*, pages 345–356, 2002.

[36] Jianjun Chen, David J. DeWitt, Feng Tian, and Yuan Wang. NiagaraCQ: A Scalable Continuous Query System for Internet Databases. *SIGMOD Rec.*, 29(2):379–390, 2000.

[37] Kwang Ting Cheng and A. S. Krishnakumar. Automatic Functional Test Generation Using the Extended Finite State Machine Model. In *Proc. of the* 30^{th} *Int. Design Automation Conf.*, DAC '93, pages 86–91. ACM, 1993.

[38] Qi Cheng, Jarek Gryz, Fred Koo, T. Y. Cliff Leung, Linqi Liu, Xiaoyan Qian, and K. Bernhard Schiefer. Implementation of Two Semantic Query Optimization Techniques in DB2 Universal Database. In *Proc. Int. Conf. on Very Large Data Bases*, pages 687–698, 1999.

[39] O. Coudert, C. Berthet, and J.C. Madre. Verification of Sequential Machines Based on Symbolic Execution. In *Proc. of the Workshop on Automatic Verification Methods for Finite State Systems*, 1989.

[40] Alexandre David. *Hierarchical Modeling and Analysis of Timed Systems*. PhD thesis, Uppsala University, November 2003.

[41] Alexandre David and M. Oliver Möller. From HUPPAAL to UPPAAL: A Translation from Hierarchical Timed Automata to Flat Timed Automata. Technical Report RS-01-11, BRICS, March 2001.

[42] Alexandre David and Yi Wang. Hierarchical Timed Automata for UPPAAL. In 10^{th} *Nirdic Workshop on Programming Theory*. Turku Centre for Computer Science, 1998.

[43] René David and Hassane Alla. Petri Nets for Modeling of Dynamic Systems: A Survey. *Automatica*, 30:175–202, February 1994.

[44] Yanlei Diao and Michael Franklin. Query Processing for High-volume XML Message Brokering. In *Proc. Int. Conf. on Very Large Data Bases*, pages 261–272. VLDB Endowment, 2003.

[45] Martin Dickhöfer and Thomas Wilke. Timed Alternating Tree Automata: The Automata-Theoretic Solution to the TCTL Model Checking Problem. In *Proc. of the* 26^{th} *Int. Colloquium on Automata, Languages and Programming*, pages 281–290. Springer, 1999.

[46] Cătălin Dima and Ruggero Lanotte. Distributed Time-Asynchronous Automata. In *Theoretical Aspects of Computing*, volume 4711 of *Lecture Notes in Computer Science*, pages 185–200. Springer, 2007.

[47] Luping Ding, Songting Chen, Elke A. Rundensteiner, Junichi Tatemura, Wang-Pin Hsiung, and K. Selcuk Candan. Runtime Semantic Query Optimization for Event Stream Processing. In *Proc. Int. Conf. on Data Engineering*, pages 676–685. IEEE Computer Society, 2008.

[48] Luping Ding, Nishant Mehta, Elke A. Rundensteiner, and George T. Heineman. Joining Punctuated Streams. In *Proc. Int. Conf. on Extending Database Technology*, volume 2992 of *Lecture Notes in Computer Science*, pages 587–604. Springer, 2004.

[49] Luping Ding, Elke A. Rundensteiner, and George T. Heineman. MJoin: A Metadata-aware Stream Join Operator. In *Proc. Int. Workshop on Distributed Event-Based Systems*, pages 1–8. ACM, 2003.

[50] Michael Eckert. *Complex Event Processing with XChange^{EQ}: Language Design, Formal Semantics and Incremental Evaluation for Querying Events.* Dissertation/Ph.D. thesis, Institute for Informatics, University of Munich, 2008.

[51] Michael Eckert, François Bry, Simon Brodt, Olga Poppe, and Steffen Hausmann. *A CEP Babelfish: Languages for Complex Event Processing and Querying Surveyed.* Springer, 2010. to appear.

[52] Michael Eckert, François Bry, Simon Brodt, Olga Poppe, and Steffen Hausmann. *Two Semantics for CEP, no Double Talk: Complex Event Relational Algebra (CERA) and its Application to XChange^{EQ}.* Springer, 2010. to appear.

[53] Norbert Eisinger. Subsumption and Connection Graphs. In *Proc. of the* 7^{th} *Int. Conf. on ArtificialIintelligence*, volume 1, pages 480–486. Morgan Kaufmann Publishers Inc., 1981.

[54] Franoise Fabret, H. Arno Jacobsen, Franois Llirbat, Joo Pereira, Jo Ao Pereira, Kenneth A. Ross, and Dennis Shasha. Filtering Algorithms and Implementation for Very Fast Publish/Subscribe Systems. volume 30, pages 115–126. ACM, 2001.

[55] Reiner Fehling. A Concept of Hierarchical Petri Nets with Building Blocks. In *Advances in Petri Nets*, volume 674 of *Lecture Notes in Computer Science*, pages 148–168. Springer, 1993.

[56] Stefano Ferilli, Nicola Di Mauro, Teresa M. A. Basile, and Floriana Esposito. A complete subsumption algorithm. In *Advances in Artificial Intelligence*, pages 1–13. Springer, 2003.

[57] Olivier Finkel. Undecidable Problems about Timed Automata. In *Formal Modeling and Analysis of Timed Systems*, volume 4202 of *Lecture Notes in Computer Science*, pages 187–199. Springer, 2006.

[58] Peter M. Fischer, Kyumars Sheykh Esmaili, and Renee J. Miller. Stream Schema: Providing and Exploiting Static Metadata for Data Stream Processing. In *Proc. Int. Conf. on Extending Database Technology*, volume 426 of *ACM Int. Conf. Proc.*, pages 207–218. ACM, 2010.

[59] N.A. Fountas, N.D. Hatziargyriou, and Valavanis K.P. Hierarchical Time-extended Petri Nets as a Generic Tool for Power System Restoration. volume 12, pages 837–843. IEEE Computer Society, 1997.

[60] Mark A. Fryman. *Quality and Process Improvement*. Cengage Learning, 2002.

[61] Sandro Giessl. Instantiating Hierarchical Timed Automata, Institute of Computer Science, LMU, Munich. Projektarbeit/project thesis, 2011.

[62] Lukasz Golab, Theodore Johnson, Nick Koudas, Divesh Srivastava, and David Toman. Optimizing Away Joins on Data Streams. In *Proc. Int. Workshop on Scalable Stream Processing System*, pages 48–57. ACM, 2008.

[63] Georg Gottlob and Alexander Leitsch. Fast Subsumption Algorithms. In *EUROCAL'85*, volume 204 of *Lecture Notes in Computer Science*, pages 64–77. Springer, 1985.

[64] Georg Gottlob and Alexander Leitsch. On the Efficiency of Subsumption Algorithms. *J. ACM*, 32:280–295, April 1985.

[65] Todd J. Green, Gerome Miklau, Makoto Onizuka, and Dan Suciu. Processing XML Streams with Deterministic Automata. In *Proc. Int. Conf. on Database Theory*, pages 173–189. Springer, 2002.

[66] Olga Grinchtein. *Learning of Timed Systems*. Dissertation/Ph.D. thesis, Uppsala University, 2008.

[67] Olga Grinchtein, Bengt Jonsson, and Martin Leucher. Learning of Event-recording Automata. In *Theoretical Computer Science*, volume 411, pages 4029–4054. Elsevier Science Publishers Ltd., 2010.

[68] Alexander Grosskopf, Gero Decker, and Mathias Weske. *The Process: Business Process Modeling using BPMN.* Meghan Kiffer Press, 2009.

[69] Hermann Gruber, Markus Holzer, Astrid Kiehn, and Barbara König. On Timed Automata with Discrete Time – Structural and Language Theoretical Characterization. In *Developments in Language Theory*, volume 3572 of *Lecture Notes in Computer Science*, pages 23–47. Springer, 2005.

[70] Ashish Kumar Gupta and Dan Suciu. Stream Processing of XPath Queries with Predicates. In *Proc. Int. Conf. on Management of Data*, pages 419–430. ACM Press, 2003.

[71] David Harel. Statecharts: A Visual Formalism for Complex Systems. volume 8. Elsevier Science Publishers Ltd., 1987.

[72] David Hemer, Robert Colvin, Ian J. Hayes, and Paul A. Strooper. Don't Care Non-determinism in Logic Program Refinement. *Electronic Notes in Theoretical Computer Science*, pages 101–121, 2002.

[73] Martijn Hendriks. *Model Checking Timed Automata: Techniques and Applications.* Dissertation/Ph.D. thesis, Radboud University Nijmegen, 2006.

[74] R. Hojatic, V. Singhal, and Robert K. Brayton. Edge-streett/edge-rabin automata environment for formal verification using language containment. Technical Report UCB/ERL M94/12, EECS Department, University of California, Berkeley, 1994.

[75] Theodore Johnson, Muthu S. Muthukrishnan, Vladislav Shkapenyuk, and Oliver Spatscheck. Query-aware Partitioning for Monitoring Massive Network Data Streams. In *Proc. Int. Conf. on Management of Data*, pages 1135–1146. ACM, 2008.

[76] Farrell Joyce. *Programming Logic and Design,* 5^{th} *ed. Comprehensive.* Thomson, 2008.

[77] Deepak Kapur and Paliath Narendran. NP-Completeness of the Set Unification and Matching Problems. In *Automated Deduction*, volume 230 of *Lecture Notes in Computer Science*, pages 489–495. Springer, 1986.

[78] Jörg-Uwe Kietz and Marcus Lübbe. An Efficient Subsumption Algorithm for Inductive Logic Programming. In *Proc. of the 11th Int. Conf. on Machine Learning*, pages 130–138. Morgan Kaufmann, 1994.

[79] Byeong Man Kim and Jung Wan Cho. A New Subsumption Method in the Connection Graph Proof Procedure. *Theoretical Computer Science*, 103:283–309, September 1992.

[80] Ekkart Kindler. On the Semantics of EPCs: A Framework for Resolving the Vicious Circle. Research report, Computer Science Department, University of Paderborn, Germany, 2006.

[81] Valerie King, Orna Kupferman, and Moshe Y. Vardi. On the Complexity of Parity Word Automata. In *Foundations of Software Science and Computation Structures*, volume 2030 of *Lecture Notes in Computer Science*, pages 276–286. Springer, 2001.

[82] Christoph Koch, Stefanie Scherzinger, Nicole Schweikardt, and Bernhard Stegmaier. FluXQuery: An Optimizing XQuery Processor for Streaming XML Data. In *Proc. Int. Conf. on Very Large Data Bases*, pages 1309–1312. VLDB Endowment, 2004.

[83] Christoph Koch, Stefanie Scherzinger, Nicole Schweikardt, and Bernhard Stegmaier. Schema-based Scheduling of Event Processors and Buffer Minimization for Queries on Structured Data Streams. In *Proc. Int. Conf. on Very Large Data Bases*, pages 228–239. VLDB Endowment, 2004.

[84] Robert Kowalski. A Proof Procedure Using Connection Graphs. *Association of Computing Machinery*, 22:572–595, October 1975.

[85] Sebastian Kupferschmid. *Directed Model Checking for Timed Automata.* Dissertation/Ph.D. thesis, Albert-Ludwigs University Freiburg, 2010.

[86] F. Laroussinie, N. Markey, and Ph. Schnoebelen. Model Checking Timed Automata with One or Two Clocks. In *Concurrency Theory*, volume 3170 of *Lecture Notes in Computer Science*, pages 387–441. Springer, 2004.

[87] Slawomir Lasota and Igor Walukiewicz. Alternating Timed Automata. *ACM Trans. Comput. Logic*, 9:1–27, April 2008.

[88] Jin Li, David Maier, Kristin Tufte, Vassilis Papadimos, and Peter A. Tucker. No Pane, no Gain: Efficient Evaluation of Sliding-Window Aggregates over Data Streams. *SIGMOD Record*, 34(1):39–44, 2005.

[89] Jiahong Liu, Quanyuan Wu, and Wei Liu. Temporal Restriction Query Optimization for Event Stream Processing. In *Advances in Web and Network Technologies, and Information Management*, volume 5731 of *Lecture Notes in Computer Science*, pages 25–35. Springer, 2010.

[90] Donald W. Loveland. *Automated Theorem Proving: A Logical Basis (Fundamental studies in Computer Science)*. Elsevier, 1978.

[91] Bertram Ludäscher, Pratik Mukhopadhyay, and Yannis Papakonstantinou. A Transducer-Based XML Query Processor. In *Proc. Int. Conf. on Very Large Data Bases*, pages 227–238. VLDB Endowment, 2002.

[92] Samuel Madden, Mehul Shah, Joseph M. Hellerstein, and Vijayshankar Raman. Continuously Adaptive Continuous Queries Over Streams. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 49–60. ACM, 2002.

[93] Jérôme Maloberti and Michèle Sebag. Theta-Subsumption in a Constraint Satisfaction Perspective. In *Proc. of the* 11^{th} *Int. Conf. on Inductive Logic Programming*, ILP '01, pages 164–178. Springer, 2001.

[94] Jérôme Maloberti and Michèle Sebag. Fast Theta-Subsumption with Constraint Satisfaction Algorithms. *Machne Learning*, 55:137–174, May 2004.

[95] Nenad Medvidovic, David S. Rosenblum, David F. Redmiles, and Jason E. Robbins. Modeling Software Architectures in the Unified Modeling Language. *ACM Trans. Softw. Eng. Methodol.*, 11:2–57, January 2002.

[96] H. Oswald, R. Esser, and R. Mattmann. An environment for specifying an executing hierarchical petri nets. In *Proc. of the* 12^{th} *Int. Conf. on Software Engineering*, pages 164–172. IEEE Computer Society Press, 1990.

[97] Christos Papadimitriou. *Alternation*, Section 16.2, pages 399–401. Addison Wesley, 1993.

[98] G. N. Paulley and G. K. Attaluri. Semantic Query Optimization in Object-Oriented Databases.

[99] Dominique Perrin and Jean-Eric Pin. *Infinite Words: Automata, Semigroups, Logic and Games*. Elsevier, 2004.

[100] James L. Peterson. Petri Nets. *ACM Comput. Surv.*, 9:223–252, September 1977.

[101] Paul Pettersson. *Modelling and Verification of Real-Time Systems Using Timed Automata: Theory and Practice.* Dissertation/Ph.D. thesis, Department of Computer Systems, Uppsala University, 1999.

[102] Nir Piterman. From Nondeterministic Büchi and Streett Automata to Deterministic Parity Automata. IEEE Computer Society, 2006.

[103] S. Ramaswamy and K.P. Valavanis. Hierarchical Time-extended Petri Nets (H-EPNs) Based Error Identification and Recovery for Multilevel Systems. volume 26, pages 164–175. IEEE Computer Society, 1996.

[104] Nick Russell, Wil M. P. van der Aalst, Arthur H. M. ter Hofstede, and Petia Wohed. On the Suitability of UML 2.0 Activity Diagrams for Business Process Modelling. In *Proc. of the* 3^{rd} *Asia-Pacific Conf. on Conceptual Modelling*, volume 53 of *APCCM*, pages 95–104. Australian Computer Society, Inc., 2006.

[105] Tobias Scheffer, Ralf Herbrich, and Fritz Wysotzki. Efficient Algorithms for theta-Subsumption. In *Proc. Int. Workshop on Inductive Logic Programming*, 1996.

[106] Tobias Scheffer, Ralf Herbrich, and Fritz Wysotzki. Efficient theta-Subsumption based on Graph Algorithms. In *Inductive Logic Programming*, volume 1314 of *Lecture Notes in Computer Science*, pages 212–228. Springer, 1997.

[107] Manfred Schmidt-Schauß. Implication of Clauses is Undecidable. *Theoretical Computer Science*, 59(3):287–296, 1988.

[108] Uwe Schöning. *Theoretische Informatik - kurzgefasst.* Spektrum Akademischer Verlag, 2003.

[109] Michael Sipser. *Alternation*, Section 10.3, pages 380–386. PWS Publishing, 2006.

[110] Maria Sorea. Bounded Model Checking for Timed Automata. In *Electronic Notes in Theoretical Computer Science*, volume 68. Elsevier Science Publishers Ltd., 2002.

[111] Rona B. Stillman. The Concept of Weak Substitution in Theorem-Proving. *J. ACM*, 20:648–667, October 1973.

[112] Hong Su, Elke A. Rundensteiner, and Murali Mani. Semantic Query Optimization in an Automata-Algebra Combined XQuery Engine over XML Streams. In *Proc. Int. Conf. on Very Large Data Bases*, pages 1293–1296. VLDB Endowment, 2004.

[113] Hong Su, Elke A. Rundensteiner, and Murali Mani. Semantic Query Optimization for XQuery over XML Streams. In *Proc. Int. Conf. on Very Large Data Bases*, pages 277–288. VLDB Endowment, 2005.

[114] Wolfgang Thomas. Automata on Infinite Objects. In *Handbook of Theoretical Computer Science*, pages 133–164. Elsevier, 1990.

[115] Salvatore La Torre and Margherita Napoli. Timed Tree Automata with an Application to Temporal Logic. *Acta Informatica*, 38:89–116, 2001.

[116] Peter A. Tucker, David Maier, Tim Sheard, and Paul Stephens. Using Production Schemas to Characterize Strategies for Querying over Data Streams. *IEEE Transactions on Knowledge and Data Engineering*, 19(9):1227–1240, 2007.

[117] W.M.P. van der Aalst. The Application of Petri Nets to Workflow Management. *Journal of Circuits, Systems and Computers*, 8(1):21–66, 1998.

[118] Sicco Verwer, Mathijs de Weerdt, and Cees Witteveen. An Algorithm for Learning Real-time Automata. In *Proc. of the* 18^{th} *Benelearn*, 2007.

[119] Sicco Verwer, Mathijs de Weerdt, and Cees Witteveen. One-Clock Deterministic Timed Automata Are Efficiently Identifiable in the Limit. In *Language and Automata Theory and Applications*, volume 5457 of *Lecture Notes in Computer Science*, pages 740–751. Springer, 2009.

[120] Sicco Ewout Verwer. *Efficient Identification of Timed Automata: Theory and Practice.* Dissertation/Ph.D. thesis, Delft University of Technology, 2010.

[121] Stratis D. Viglas and Jeffrey F. Naughton. Rate-Based Query Optimization for Streaming Information Sources. In *Proc. Int. Conf. on Management of Data*, pages 37–48. ACM, 2002.

[122] Jiacun Wang. *Timed Petri Nets: Theory and Application.* Springer, 1998.

[123] W.M.Zuberek. Timed Petri Nets: Definitions, Properties, and Applications. *Microelectronics Reliability*, 31:627–644, 1991.

[124] Mihalis Yannakakis. Hierarchical State Machines. In *Theoretical Computer Science: Exploiting new frontiers of theoretical informatics*, volume 1872 of *Lecture Notes in Computer Science*, pages 315–330. Springer, 2000.

[125] Sergio Yovine. Model Checking Timed Automata. In *Lectures on Embedded Systems*, volume 1494 of *Lecture Notes in Computer Science*, pages 114–152. Springer, 1998.

[126] Xudong Zhao and Yulin Feng. Automatic and Hierarchical Verification for Concurrent Systems. volume 5, pages 241–249. Springer, 1988.

Index