POLITECNICO
MILANO 1863

Department of Industrial and Information Engineering
Master degree in Computer Science and Engineering

Combining streaming events with static data in the Complex Event Processing tool T-Rex

Supervisors:
Prof. Giampaolo Cugola
Prof. Alessandro Margara

Thesis dissertation of:
Angelo Di Pilla (820336)

Academic year 2015 - 2016

ABSTRACT

Data management systems over the years evolved in two diametrically opposite fields of application: data storage and stream processing. Nowadays both the technologies are widespread and they are often required to cooperate toward a common goal. However the integration of the two is still in an early stage of development and usually custom solutions are required for each specific deployment.

The purpose of the thesis is to model and evaluate a native and general purpose integration of static data sources into the *Complex Event Processing* tool *T-Rex* [4]. To do so we extended and re-designed the T-Rex engine to integrate a static data source (namely a *SQLite* database) with the event streams that flow into the engine. This required to go through the T-Rex rule language extension, followed by an elaboration of the key algorithms to efficiently process this new language, i.e., to combine static and streaming data, concluding with a careful analysis of the performance of the new engine.

The project shows that events streams and data collections can be modeled with similar logical abstractions, simplifying the description of those problems that operate on the boundary of the two domains. At the same time the real-time performances of the original T-Rex implementation can be preserved, within reasonable limits.

SOMMARIO

I sistemi di data management nel corso degli anni si sono sviluppati in due campi di applicazione diametralmente opposti: immagazzinamento di dati e elaborazione in tempo reale di flussi di informazione. Al giorno d'oggi entrambe le tecnologie sono ampiamente diffuse e spesso è necessario che cooperino per il raggiungimento di comuni obiettivi. Tuttavia l'integrazione delle due è ancora in una fase iniziale di sviluppo e solitamente si rendono necessarie soluzioni personalizzate per specifico caso d'uso.

Lo scopo della tesi è di modellare e validare gli effetti di un'integrazione di sorgenti di dati statici nel tool di *Complex Event Processing T-Rex* [4]. Per raggiungere questo obiettivo presentiamo l'implementazione di un componente che permetta l'interazione con un database *SQLite*. In particolare descriviamo l'estensione del linguaggio di definizione di regole di T-Rex, spieghiamo i principali algoritmi e discutiamo le prestazioni ed i limiti analizzando i risultati di una serie di test.

Il progetto mostra come stream di eventi e collezioni di dati possano in effetti essere descritti con astrazioni logiche del tutto simili, permettendo di affrontare più facilmente problemi che operano sul confine tra i due domini. Inoltre i risultati dei test evidenziano come le prestazioni real-time dell'implementazione originale di T-Rex possano essere preservate sotto ragionevoli condizioni.

CONTENTS

LIST OF FIGURES

LISTINGS

INTRODUCTION

In the past Complex Event Processing (CEP) [10] was seen as a specialized product, applied to systems focused exclusively on streams and real-time processing and used only when the constraints where so strict that were impossible to satisfy with standard databases. Those requirements had the maximum priority and users were willing to sacrifice convenience to achieve the necessary speed.

Nowadays the increase in data production rate and the new trend of reactive and proactive systems is helping CEP and stream processing in general to gain popularity. In this new environment, requirements and architectures are often wider than just event handling and system integration is starting to get valued as much or even more than pure performance.

This thesis investigates the feasibility and limits of an interoperability between the T-Rex CEP engine and the SQLite database engine, showing that static data can find a natural fit into the *TESLA*[1] [7] rule definition language and that the performance of an embedded database can satisfy the requirements of a real time execution, especially if in combination with a cache mechanism. We also highlight how the response time of the external *DBMS* and the cache friendliness of the processed data remain strict requirements for a practical usability.

In particular my contribution were:

- Identification of ambiguities in some TESLA operators and proposal of refinements, supported by a formal definition of the syntax, which was previously presented mostly by examples.
- Rewrite of the T-Rex engine to make it more robust, extensible and accurate with respect to the specification.
- Design and formalization of a syntactic and semantic extension of the TESLA language to seamlessly combine event streams with static, relational data.
- Development of the integration of the T-Rex engine with the SQLite database to interpret the new rule language, and introduction of a caching layer to improve performance in accessing static data.

The rest of the thesis is organized as follows: the first chapter introduces the field of study, providing a basic overview of features and terminology related with Information Flow Processing and Complex Event Processing in particular. The second chapter presents TESLA and T-Rex, analyzing their strengths and weakness. The third chapter describes the way we extended the TESLA language to enable CEP

1 TESLA is the T-Rex rule specification language

rules that seamlessly combine static and streaming data. The fourth chapter explains the architecture of the system and the algorithms used. The fifth chapter studies, through a wide set of benchmarks, how the various design choices we made impact the performance of the resulting system, under various conditions. The last chapter draws the main conclusions and proposes future development.

1

CEP BACKGROUND

1.1 INTRODUCTION

Complex Event Processing (CEP) [10] consists in analysis and manipulation of streams of data, where each data item models an event occurring in an observed domain: starting from the primitive ones received as external input to the system, events are filtered, pattern matched, aggregated and combined into composite ones according to a given set of rules.

CEP represents the dual to the classical databases model, in which data are usually static or changing at a relatively slow pace and queries vary from time to time, depending on the information required at the moment. CEP instead is characterized by persistent rules applied to a continuous flow of new data.

Before stepping into more advanced topics, in this chapter I will make an overview of the field of study, from a historical contextualization to the general concepts and terminology.

1.2 HISTORY AND CONTEXT

The discipline was born in the '90s as an evolution of publish-subscribe systems, to satisfy the needs of expressing patterns of multiple events rather than simple topic or content filtering. From that starting point it focused on real-time applications (like IoT sensors, fraud and emergencies detection, stock markets analysis, transportation) and developed high level languages and operators to support common time related tasks.

In the meantime different approaches to Information Flow Processing (IFP) [6] arose from other research fields and complemented CEP requirements and features.

For example the database community developed *Data Stream Management Systems* (DSMSs) [1], which reduced everlasting streams to relational collections using windowing operators and manipulating them with continuous queries.

Moreover, recently *distributed stream processors* started getting a lot of interest and traction from big tech companies as a new paradigm to handle extremely parallelized and throughput focused stream manip-

ulations, in the same way *Hadoop MapReduce* [8] revolutionized batch processing.

Nowadays everything is slowly settling into a more unified and standardized ecosystem: tools are expanding their functionalities to match different needs of IFP and exploring the interaction with other systems and data sources.

1.3 FEATURES

As mentioned before, CEP engines are characterized by the execution of persistent rules in low-latency applications and one of the main aspects that differentiate CEP from other similar technologies is its focus on events patterns, opposed to single events or batches.

To ease rule development and optimization, CEP engines usually define a *Domain Specific Language* (DSL). These DSL are typically declarative and somehow inspired to SQL, meaning that they allow to describe the desired solution in terms of which data to retrieve and under which constraints.
There is a common base of operators and language constructs that can be found across most of the implementations, so let's try to review these building blocks and lay a terminology foundation for the chapters to come.

First of all it is necessary to define a sequence of events described as some kind of list of event types that have to be notified to activate the rule. That sequence is characterized by relationships of happen-before and other time constraints to delimit the eligibility to be part of the pattern. In practice, there are operators to define windows in terms of duration or number of events or delimited by the occurrence of two events that acts as boundaries.
In addition to temporal properties we need to apply filters based on content, so it is possible to write algebraic expressions, comparisons and possibly parameterization over the attributes of one or more events in the sequence.
Sometimes it is required the ability to iterate over a unknown number of events to detect a trend, some others to negate a predicate, so that the rule is fulfilled if there are no events that match the filters.
After a pattern is defined is important to declare selection policies, that are a way of expressing how many and which events satisfying the constraints should be picked for further processing. For example we might want to propagate only the most/least recent candidate.
Then we need to transform the data, combining information from multiple events or running aggregate functions.
Finally it may be desirable for an event notification to participate to a rule evaluation only once and then be discarded, waiting for new notifications. This practice is called event consumption and its effect and tunability can vary a lot from one implementation to another.

2

TESLA AND TREX

2.1 INTRODUCTION

TESLA [7] is a declarative strongly typed CEP language that, as some other alternatives, provides a comprehensive set of common operation on events (like filtering and parameterization, composition and pattern detection, negation and aggregation) and allows to control selection policies, time windows and event consumption. However, while the competitors rely on informal documentation that leaves room to ambiguities, TESLA's unprecedented characteristic is its aim to a complete semantic specification with *TRIO* [9], a first order temporal logic. The definition in advance of a precise behavior for each feature improves coherence in the development of engines based on TESLA and helps users to understand the language deeply with less empirical research. It is part of the purpose of the thesis to keep working on this track.

TESLA reference implementation is T-Rex [4], a CEP engine written in C++. It was initially designed around an algorithm called *Automata-based Incremental Processing* (AIP), which transforms each rule in a state machine that is spawned and activated by every incoming event until successful pattern detection or failure; AIP was fundamental to analyze the complexity bounds imposed by the language. Later T-Rex has been rewritten using a different algorithm called *Column-based Delayed Processing* (CDP) [5], that works accumulating events and processing them in batch as soon as a possible trigger is detected; this technique was found to be faster and easier to parallelize and offered the opportunity for a *CUDA* implementation.

While the specification of TESLA's semantic was a central topic of the very first paper, at the time of the writing the syntax was always described informally and through examples. This was identified as a risk of possible misunderstandings in the very foundation of the project, so as first thing in this chapter I will try to define the TESLA grammar in a more rigorous and hopefully clear way. Then I will summarize the previous writings about semantic and finally will highlight the inconsistencies between the semantics of TESLA and its actual implementation in the T-Rex engine, which aroused after system analysis and team discussion, followed by the clarifications or modifications proposed.

2.2 BNF GRAMMAR

Backus-Naur Form (BNF) is a notation for context-free grammars that allows to recursively define the composition of every single syntax feature. The components of this kind of writing are terminal symbols, that are simple strings possibly empty (ϵ), and non terminal ones, that are wrapped in angle brackets. Each non terminal is defined in a derivation rule and on the right hand side there will be the non terminal name, while on the left one there will be a sequence of symbols possibly separated by a vertical line (|) to imply choice between options.

BNF is also used, in a machine readable form, by parser generators like *ANTLR* (the one used by the project), so technically there is a definition already, but it had to go through some scarcely documented compromises to avoid parsing ambiguities and to face some practical need. The version presented here instead is meant for human comprehension and as a high level reference, so I will try make it as clear and self explanatory as possible, giving up the irrelevant details needed only for parsing purposes.

2.2.1 *Rule basic structure*

The outline of a rule is characterized by four main sections: definition of the derivate event, pattern of events, attribute assignment and event consumption.

⟨rule⟩ ⊨ ⟨define⟩ ⟨from⟩ ⟨where⟩ ⟨consuming⟩

2.2.2 *Define clause*

The definition of a complex event is characterized by the name of the soon to be generated tuple and by a list of attributes names and types.

⟨define⟩ ⊨ `define` ⟨capital identifier⟩ `(` ⟨attributes⟩ `)`

⟨attributes⟩ ⊨ ⟨attribute⟩ ⟨attributes tail⟩ | ϵ
⟨attributes tail⟩ ⊨ , ⟨attribute⟩ ⟨attributes tail⟩ | ϵ
⟨attribute⟩ ⊨ ⟨lower identifier⟩ : ⟨attribute type⟩
⟨attribute type⟩ ⊨ `int` | `float` | `bool` | `string`

2.2.3 *From clause*

From clause is the core of pattern detection and is made of a sequence of predicates on different events and aggregates.

⟨from⟩ ⊨ `from` ⟨predicate body⟩ ⟨predicates⟩

⟨predicates⟩ ⊨ `and` ⟨predicate⟩ ⟨predicates⟩ | ϵ
⟨predicate⟩ ⊨ ⟨event⟩ | ⟨aggregate⟩

2.2.4 *Where clause*

The where section is composed by a set of assignments of the attributes of the soon to be generated event using arbitrary expression over the data processed in the pattern detection phase.

⟨where⟩ ⊨ `where` ⟨assignments⟩ | ϵ

⟨assignments⟩ ⊨ ⟨assignment⟩ ⟨assignments tail⟩
⟨assignments tail⟩ ⊨ `,` ⟨assignment⟩ ⟨assignments tail⟩ | ϵ
⟨assignment⟩ ⊨ ⟨lower identifier⟩ `=` ⟨expression⟩

2.2.5 *Consuming clause*

Consumption policy is simply defined as a list of names of events, that took part in the from clause.

⟨consuming⟩ ⊨ `consuming` ⟨capital identifier⟩ ⟨capital identifiers⟩ | ϵ

2.2.6 *Predicate body*

The base of a predicate is made by a tuple constrained by a set of boolean expression over current event's data and parameters, possibly followed by an alias definition.

⟨predicate body⟩ ⊨ ⟨constrained tuple⟩ ⟨alias⟩
⟨constrained tuple⟩ ⊨ ⟨capital identifier⟩ `(` ⟨constraints⟩ `)`
⟨constraints⟩ ⊨ ⟨expression⟩ ⟨constraints tail⟩ | ϵ
⟨constraints tail⟩ ⊨ `,` ⟨expression⟩ ⟨constraints tail⟩ | ϵ
⟨alias⟩ ⊨ `as` ⟨capital identifier⟩ | ϵ

Note: in this formalization we use commas to separate constraint, opposed to AND as in the original papers, to better distinguish predicates vs. filters composition.

2.2.7 *Event*

An event predicate (except the trigger one that we can see in the $from$ clause) adds to tuple filtering a selection policy chosen between $each, not, first, last$ and constraints about the time window.

⟨event⟩ ⊨ ⟨event selection⟩ ⟨predicate body⟩ ⟨timing⟩
⟨event selection⟩ ⊨ `each` | `not` | `first` | `last`

2.2.8 *Aggregates*

TESLA has the common aggregators that can be found in DBMS and other CEP engines, but the list could be extended if needed. They are applied on a set of tuples filtered in a similar way to event predicates and the result can be used in an additional constraint for the event pattern.

⟨aggregate⟩ ⊨ ⟨aggregate body⟩ ⟨aggregate constraint⟩
⟨aggregate body⟩ ⊨ ⟨aggregator⟩ `(` ⟨aggregate inner⟩ `)`
⟨aggregate inner⟩ ⊨ ⟨constrained tuple⟩ ⟨attribute selection⟩ ⟨timing⟩
⟨aggregator⟩ ⊨ `AVG` | `SUM` | `MAX` | `MIN` | `COUNT`
⟨attribute selection⟩ ⊨ `.` ⟨lower identifier⟩ | ϵ
⟨aggregate constraint⟩ ⊨ ⟨binary operator⟩ ⟨expression⟩

2.2.9 *Timing*

Time constraint is imposed with two different type of window: one of given duration from a starting event, the other delimited by a couple of distinct events.

⟨timing⟩ ⊨ ⟨within⟩ | ⟨between⟩
⟨within⟩ ⊨ `within` ⟨time⟩ `from` ⟨capital identifier⟩
⟨between⟩ ⊨ `between` ⟨capital identifier⟩ `and` ⟨capital identifier⟩
⟨time⟩ ⊨ ⟨float⟩ ⟨time unit⟩
⟨time unit⟩ ⊨ `d` | `h` | `min` | `s` | `ms` | `us`

2.2.10 *Expressions and constraints*

Expressions in TESLA are common algebraic, boolean and string operations composed with each other and they can operate on immediate values, references to current tuple attributes or parameters.

⟨expression⟩ ⊨ ⟨parenthesization⟩ | ⟨operation⟩ | ⟨atom⟩
⟨parenthesization⟩ ⊨ `(` ⟨expression⟩ `)`
⟨operation⟩ ⊨ ⟨binary operation⟩ | ⟨unary operation⟩
⟨binary operation⟩ ⊨ ⟨expression⟩ ⟨binary operator⟩ ⟨expression⟩
⟨unary operation⟩ ⊨ ⟨unary operator⟩ ⟨expression⟩
⟨binary operator⟩ ⊨ *Common algebraic and comparison operators*
⟨unary operator⟩ ⊨ *Common unary operators*
⟨atom⟩ ⊨ ⟨identifier⟩ | ⟨parameter⟩ | ⟨immediate⟩
⟨identifier⟩ ⊨ ⟨qualifier⟩ ⟨lower identifier⟩
⟨qualifier⟩ ⊨ ⟨capital identifier⟩ `.` | ϵ

2.2.11 *Basic types*

These basic symbols help defining the rest of the BNF. We can notice the three types of identifiers: *capital* used for event names, *lower* used for tuple attributes and *parameter* obviously used for parameterization.

⟨capital identifier⟩ ⊨ *An identifier starting with an uppercase letter*
⟨lower identifier⟩ ⊨ *An identifier starting with a lowercase letter*
⟨parameter⟩ ⊨ $ ⟨lower identifier⟩
⟨capital identifiers⟩ ⊨ , ⟨capital identifier⟩ ⟨capital identifiers⟩ | ϵ
⟨lower identifiers⟩ ⊨ , ⟨lower identifier⟩ ⟨lower identifiers⟩ | ϵ
⟨immediate⟩ ⊨ *An immediate value, like a digit*
⟨float⟩ ⊨ *A floating point number*

2.3 SEMANTIC

2.3.1 *TRIO overview*

Created especially to address the needs of real-time systems specification, TRIO [9] is a formalism based on the extension of *First Order Logic* (FOL) with temporal operators that allow to describe properties in evolution and even to express distance and duration in time. This ability to reason quantitatively makes TRIO different from classical temporal logics and particularly suitable to deal with events and their occurrences.

The syntax is composed of the typical elements of a FOL, like variables, functions, predicates, propositional operators and quantifiers. In addition to those there are special temporal operators $Futr(A,t)$ and $Past(A,t)$ (plus derivatives). Moreover usual arithmetic functions, like $+$ and $-$, and common relational predicates, like $=$ and $<$, are assumed to be predefined at least for the temporal domain.

The semantic is again based on FOL, but some variables and predicates are time-dependent, meaning that their value changes in different instants. In the interpretation of a formula there is a present time, left implicit, that is the reference point for any temporal expression. The operator $Futr(A,t)$ is *true* if A holds t time units in the future and respectively $Past(A,t)$ is *true* if A holds t time units in the past. From those basic operators many others can be derived, but for the purpose of the thesis we will need just two of them:

$$Alw(A) = A \wedge \forall t > 0\ Futr(A,t) \wedge \forall t > 0\ Past(A,t)$$
$$WithinP(A,t_1,t_2) = \exists x\ (t_1 \leq x \leq t_1 + t_2 \wedge Past(A,x))$$

$Alw(A)$ intuitively means that A is always *true* in the past, present and future. While $WithinP(A,t_1,t_2)$ means that A is valid in some past instant within t_1 and $t_1 + t_2$.

2.3.2 *Basic concepts*

Labels and uniqueness of selection

First of all we have to introduce the concept of *label*: a unique global identifier, that makes possible to discern one event from another, even when they share the very same attributes and time. We assume that incoming events have been correctly labeled before entering the system, while to determine the label of a generated complex event we define the function $lab(r, s)$, where r is the rule that was triggered, s is the list of labels of the events that satisfied the pattern and the returned value is the label for the emitted event.
It is fundamental that *lab* will always generate a different label for each new event, so the function must be injective:

$$Alw(\forall\ r_1, s_1, r_2, s_2\ ((lab(r_1, s_1) = lab(r_2, s_2)) \leftrightarrow (r_1 = r_2 \wedge s_1 = s_2)))$$

At the same time we need to ensure the *uniqueness of selection*, which states that a rule r can be applied to the same list of events s only once. We notice that there is always a trigger event that happen at the same time of the generated one, so in a new time instant the trigger is necessarily changed. While during a single instant it's meaningless to generate more than one event from the very same list.

Event occurence

To describe the occurrence of an event we introduce the time-dependent predicate $Occurs(e, l)$, where e is the type of the event and l is the label, the predicate is *true* in the single instant of occurrence and $false$ in any other.

$$Alw(\forall e_1, e_2 \in E, \forall l \in L\ (Occurs(e_1, l) \wedge Occurs(e_2, l) \rightarrow e_1 = e_2))$$

$$\begin{aligned} &Alw(\forall e_1, e_2 \in E, \forall l \in L, \forall t > 0\ (Occurs(e_1, l) \rightarrow \\ &\neg Past(Occurs(e_2, l), t) \wedge \neg Futr(Occurs(e_2, l), t))) \end{aligned}$$

Where E is the set of event types and L the set of valid labels.

Time of occurence

We define a time-dependent function $time(l)$, where l is a label of an event, that returns (if any) the time of occurrence in the past with respect to the current instant.

$$Alw(\forall l \in L, \forall t > 0\ (time(l) = t \leftrightarrow Past(Occur(l), t)))$$

Attributes values

Finally to reason about the content of an event we define the function $attVal(l, n)$, where l is a label, n is a valid attribute name and the result is the value of the attribute. [1]

2.3.3 *Specification*

Event emission

As introduced during the syntax description, the $define$ statement declares the event to be generated and the rule is triggered as soon as a sequence of events satisfies the pattern of $from$ section.
The simplest rule is the one that produces a complex event CE for any notification of simple event SE.

$$define\ CE()\ from\ SE()\ \triangleq$$

$$Alw(\forall l(Occurs(CE,\ lab(\{l\})) \leftrightarrow Occurs(SE,\ l)))$$

The next step is being able to filter the trigger event, using basic operations on its attributes. So that CE is generated if and only if SE's attributes matches the all the constraints.

$$define\ CE()\ from\ SE(att_1\ op_1\ val_1,\ ...,\ att_n\ op_n\ val_n)\ \triangleq$$

$$Alw(\forall l(Occurs(CE,\ lab(\{l\})) \leftrightarrow (Occurs(SE,\ l) \wedge attVal(l,\ att_1)\ op_1\ val_1 \wedge\ ...\ \wedge attVal(l,\ att_n)\ op_n\ val_n)))$$

The constraints can be generalized from a single operator to generic expressions over tuple attributes and, as we will see, parameters. Each expression is mapped over a corresponding TRIO predicate that depends on the same arguments.

$$define\ CE()\ from\ SE(expr_1(att_1,\ ...,\ att_n),\ ...,\ expr_m(att_1,\ ...,\ att_n))\ \triangleq$$

$$Alw(\forall l(Occurs(CE,\ lab(\{l\})) \leftrightarrow (Occurs(SE,\ l) \wedge Pred_1(attVal(l,\ att_1),\ ...,\ attVal(l,\ att_n)) \wedge\ ...\ \wedge Pred_m(attVal(l,\ att_1),\ ...,\ attVal(l,\ att_n)))))$$

To assign values to CE attributes we use the $where$ clause and each TESLA expressions is mapped to a TRIO function f_i.

$$define\ CE(att_1,\ ...,\ att_n)\ from\ SE()\ where\ att_1 = expr_1(SE.att_1,\ ...,\ SE.att_m),\ ...,\ att_n = expr_n(SE.att_1,\ ...,\ SE.att_m)\ \triangleq$$

1 The concept is here simplified and doesn't take into account attributes types and domains, but for explanation purposes it is the right level of abstraction.

$$Alw(\forall l((Occurs(CE,\ lab(\{l\})) \leftrightarrow Occurs(SE,\ l))$$
$$\wedge (Occurs(CE,\ lab(\{l\})) \rightarrow\ attVal(lab(\{l\}),\ att_1) =$$
$$f_1(attVal(l,\ att_1),\ ...,\ attVal(l,\ att_m)) \wedge\ ... \wedge$$
$$attVal(lab(\{l\}),\ att_n) = f_n(attVal(l,\ att_1),\ ...,\ attVal(l,\ att_m)))))$$

Event composition

To capture patterns of multiple events, TESLA provides some composition operators, made of a selection policy (*each*, $first$, *last*, *not*) and a window (*within* or *between*).
The operator *within* defines a time range that goes from the occurrence of a specified event to a fixed amount of time in the past. The operator *between* defines a time range delimited by two different events.
The selection *each* emits an event for every occurrence inside the specified window.

$$define\ CE()\ from\ A()\ and\ each\ B()\ within\ x\ from\ A\ \triangleq$$

$$Alw(\forall l_0, l_1 \in L\ (Occurs(CE, lab(r, \{l_0, l_1\}))\ \leftrightarrow$$
$$(Occurs(A, l_0)\ \wedge\ WithinP(Occurs(B, l_1),\ time(l_0),\ x))))$$

The selection *last* emits a single event using the most recent occurrence in the window (ties are broken using an artificial total ordering on labels).

$$define\ CE()\ from\ A()\ and\ last\ B()\ within\ x\ from\ A\ \triangleq$$

$$Alw(\forall l_0, l_1 \in L\ (Occurs(CE, lab(r, \{l_0, l_1\}))\ \leftrightarrow$$
$$(Occurs(A, l_0)\ \wedge\ WithinP(Occurs(B, l_1), time(l_0), x)$$
$$\wedge\ \neg(\exists l_2 \in L, \exists t \in [time(l_0), time(l_1))\ Past(Occurs(B, l_2), t))$$
$$\wedge\ \neg\exists l_3 \in L\ (Past(Occurs(B, l_3), time(l_1))\ \wedge\ l_3 > l_1))))$$

The selection $first$ emits a single event using the least recent occurrence in the window (ties are broken using an artificial total ordering on labels).

$$define\ CE()\ from\ A()\ and\ first\ B()\ within\ x\ from\ A\ \triangleq$$

$$Alw(\forall l_0, l_1 \in L\ (Occurs(CE, lab(r, \{l_0, l_1\}))\ \leftrightarrow$$
$$(Occurs(A, l_0)\ \wedge\ WithinP(Occurs(B, l_1), time(l_0), x)$$
$$\wedge\ \neg(\exists l_2 \in L, \exists t \in (time(l_1), time(l_0) + x]\ Past(Occurs(B, l_2), t))$$
$$\wedge\ \neg\exists l_3 \in L\ (Past(Occurs(B, l_3), time(l_1))\ \wedge\ l_3 < l_1))))$$

The negation emits an event if there aren't occurrences in the window.

$$define\ CE()\ from\ A()\ and\ not\ B()\ within\ x\ from\ A\ \triangleq$$

$$Alw(\forall l_0, l_1 \in L\ (Occurs(CE, lab(r, \{l_0, l_1\}))\ \leftrightarrow$$
$$(Occurs(A, l_0)\ \wedge\ \neg WithinP(Occurs(B, l_1),\ time(l_0),\ x))))$$

The same principles apply with *between* operator:

$$define\ CE()\ from\ A()\ and\ each\ B()\ within\ x\ from\ A$$
$$and\ each\ C()\ between\ A\ and\ B\ \triangleq$$

$$Alw(\forall l_0, l_1, l_2 \in L\ (Occurs(CE, lab(r, \{l_0, l_1, l_2\}))\ \leftrightarrow$$
$$(Occurs(A, l_0)\ \wedge\ WithinP(Occurs(B, l_1),\ time(l_0),\ x)$$
$$\wedge\ WithinP(Occurs(C, l_2),\ time(l_0),\ time(l_1) - time(l_0)))))$$

Parameterization

Parameters, identified by the leading dollar sign, work as variables to interconnect filter expressions in different predicates inside a rule. In TRIO they can be represented simply by adding an additional constraint to the formula.

$$define\ CE()\ from\ A(att_0 = \$x)\ and$$
$$each\ B(att_2 < \$x)\ within\ x\ from\ A\ \triangleq$$

$$Alw(\forall l_0, l_1 \in L\ (Occurs(CE, lab(r, \{l_0, l_1\}))\ \leftrightarrow$$
$$(Occurs(A, l_0)\ \wedge\ WithinP(Occurs(B, l_1),\ time(l_0),\ x)$$
$$\wedge attVal(l_0, att_0) > attVal(l_1, att_1))))$$

Aggregates

TESLA provides the most common aggregates function, that compute a single result from a set of events in a window. They can be used for filtering and for value assignment.

$$AGGR(A.attr\ within\ x\ from\ B) = y\ \triangleq$$

$$Alw(\forall S(\forall s(s \in S\ \leftrightarrow \exists l \in L(s = \langle l,\ attVal(l, attr)\rangle$$
$$\wedge\ WithinP(Occurs(A, l),\ Time(B),\ x))) \rightarrow\ aggr(S) = y))$$

Event consumption

Event consumption is the ability to mark an event that previously matched in a pattern to be removed from the list of the one available to trigger the rule again. To describe this behavior in TRIO we introduce the predicate $Consumed(r, l)$, that should be $false$ until the rule r consumes the event labeled l and remain $true$ from then on.

$$Alw(\forall l \in L, \forall r \in R(Consumed(r, l) \rightarrow$$
$$\forall\ t > 0\ Futr(Consumed(r, l),\ t)))$$

$$Alw(\forall l \in L, \forall e \in E, \forall r \in R, \forall S\ ((\neg\exists t > 0\ (l \in S \wedge Past(Occurs(e, lab(r, S), t)))) \rightarrow \neg Consumed(r, l)))$$

We can use this predicate to describe rules with consuming clauses.

$$define\ CE()\ from\ A()\ and\ each\ B()$$
$$within\ x\ from\ A\ consuming\ B\ \triangleq$$

$$Alw(\forall l_0, l_1(Occurs(CE, lab(\{l_0, l_1\})) \leftrightarrow Occurs(A, l_0) \wedge$$
$$WithinP(Occurs(B, l_1), time(l_0), x) \wedge \neg Consumed(r, l_1)$$
$$Occurs(CE, lab(\{l_0, l_1\})) \rightarrow \forall t > 0\ Futr(Consumed(r, l_1), t)))$$

Hierarchies and iterations

Unlike other CEP languages, TESLA allows complex events, defined through rules, to be part of other detection patterns and not only to be used as subscription topics.

The straight forward consequence is the possibility to create a hierarchical structure. The composition improves code organization, reuse and maintainability, which are fundamental goals of any programming language, but it also makes possible progressive data refinement, that eases development in contexts characterized by different layers of information detail.

At the same time, complex events can be referenced in a recursive approach, that gives the power to express iterations without any additional language constructs. Iteration is necessary to analyze the evolution in time of the input flows and detect trends that could span over an undefined number of events.

$$define\ RepA(Times, Val)\ from\ A()$$
$$where\ Times = 1\ and\ Val = A.Val$$

$$define\ RepA(Times, Val)\ from\ A(\$x)$$
$$and\ last\ RepA(Val \leq \$x)\ within\ 3min\ from\ A$$
$$where\ Times = RepA.Times + 1\ and\ Val = \$x$$
$$consuming\ RepA$$

$$define\ B(Times)\ from\ RepA()$$
$$where\ Times = RepA.Times$$

This ability to create loops is powerful and tunable, but introduces the risk of non-termination typical of highly expressive languages.

2.4 AMBIGUITIES AND CLARIFICATIONS

During the analysis of the previously defined syntax, it appeared clear the presence of some ambiguities in the semantics of some specific TESLA operators and that, besides the additions of static data, a general renewal of the language was appropriate.

2.4.1 *Parameters*

In previous papers, parameterization was presented only through examples and there was no explicit definition of its expressivity. So it wasn't clear if parameters needed two separate phases of assignment and usage or they could appear as free variables in high level logical constraints.
The former option appeared to be the intended one and few more issues followed from that consideration, for example it wasn't stated if the order of definition and usage was relevant, there was no difference between assignment and comparison operators and parameters definition was mixed with selection predicates, without any syntax distinction.
We need the syntax to make everything more clear and intuitive by reordering clauses and separating parameters definitions.

2.4.2 *Event declaration*

TESLA was presented as a strongly typed language and so it was expected to prevent type incoherences before the actual rule evaluation. However simple events weren't declared beforehand and just appeared at runtime as input to the system, while complex event could be emitted by different rules possibly with different signatures. That lack of information made hard to check in advance.
More over it was impossible to assign a numerical id to an event type for more convenient interaction with external systems.
Some of these issues have been tackled informally in the implementation, but the solution adopted looked more like an unavoidable patch rather than a design choice. To fill those gaps we propose a new independent statement to declare the signature and the id of any event that is going to be processed by the system.

2.4.3 *Filtering*

The use of the keyword WHERE for attributes assignment can be counterintuitive, since it wrongly reminds of the SQL clause for rows filtering.
Moreover filtering was made in the FROM clause, mixed with the selection predicates, in a way that could easily became messy.

The language revision should improve keywords choice to facilitate the approach of newcomers and it should better separate selection and filtering to keep statements as clean as possible.

2.4.4 *Timestamp and selection*

During event composition, inclusion or exclusion of the present time instant can make a great difference, but the topic wasn't clearly discussed in previous writings.

In the original paper the predicate $WithinP(A, t_1, t_2)$ was defined using the symbol $\leq$, so the instants t_1 and t_2 were considered part of the range. However motivations and consequences of this decision weren't explained.

The exclusion of the present instant wouldn't allow hierarchical and recursive composition, which are needed to modularize rules when complexity increases and to have enough expressivity power to describe iterations.

Nevertheless the choice of closed ranges, which provides that kind of expressivity, bring some downsides too. For example it is possible to create paradoxes with events that should be generated if they are not and shouldn't be generated if they are, like:

$$define\ A\ from\ B\ and\ not\ A\ within\ X\ from\ B$$

At the same time it is impossible to write a rule like:

$$define\ B\ from\ A\ as\ A1\ and\ not\ A\ within\ X\ from\ A1$$

that sounds reasonable at a first look, but it wouldn't emit any event.

A practical solution is the one used by the reference implementation that breaks paradoxes introducing an implicit total ordering: it evaluates conditions in execution order and never looks back. The price paid is a runtime nondeterminism, which means that different processing order of logically concurrent events could lead to different outcomes.

This approach is the most reasonable found so far and should be considered the standard, but the users should be aware of the implications and topic should remain open to further discussion.

2.4.5 *Consuming*

Event consumption was always described in combination with *each* selection policy, while the use of *first* or *last* wasn't really clarified. In particular, when the first (resp. last) event satisfying the pattern was already marked as consumed, it wasn't specified if the match should be considered terminated, without complex event emission, or the processor should just look for the next one that is not consumed. The reference implementation adopted the latter approach and we decided to take it as the standard.

3

LANGUAGE EXTENSION

3.1 INTRODUCTION

Without a connection to a DBMS, whenever it is required to access some data outside the characteristic event payload, it is unavoidable to fall into cumbersome workarounds: one option is to attach the needed information to each event, the other it to simulate a database table with an everlasting events window filled with the entire dataset.

For example let's say that we are processing information coming from a public transport network, we receive an event of a bus stuck in a traffic jam, the characteristic payload is the id of the bus and the amount of time it will delay its ride. If we would like to notify every bus stop on that line, we would first need to know on which line the bus is traveling.
We could repeat the line information on every delay event. However one issue it that all the information would have to be provided by custom software from outside the engine, which implies technical expertise and effort. Moreover if we would like to know something else, like the capacity of the bus or the name of the driver, we would have to expand even more the payload and, every time a new field is required, we would have to restart the system to load the new event structure. Finally if there were more than an event originating from that bus, we would need to attach the same extras to each and every one of them.
An alternative is to create a stream containing an event for each row in the dataset and use predicates with extremely broad windows that allow to interact with those entries at any time. This allows a better separation and it eases the introduction of new information sources. However it loses a lot of efficiency, making the engine almost unusable, because of its lack of optimizations and indexes.

That said and before getting into the details of the presented solution, let's have an overview of the requirement and goals that influenced the design and development.
The foundation is to keep it simple and natural, while preserving the baseline performances of a real-time process. We would also like to have the maximum expressivity and tunability of the system so that it can be adapted to any context and situation. From a devel-

oper point of view we want the code to be extensible to new storages and optimizations, while remaining organized and small enough for maintenance. Last but not least we want to continue to have a formal definition of every aspect of the language.

In the following paragraphs I will propose some modifications to TESLA to mitigate the issues highlighted at the end of the previous chapter and an extension to express joins and filters over heterogeneous data sources. As before there will be two main sections: the first focused on syntax changes, the second concerning the semantic interpretation and specification.

3.2 SYNTAX EXTENSION

3.2.1 *Tuple declaration*

This is an entirely new instruction that aims to provide all the required information for rules type-checking. In the beginning it is specified if the tuple is an event or a data row. The main part is the name of the tuple and the attributes types. At the end there is the assignment of a numerical id.

⟨declaration⟩ ⊨ ⟨declare⟩ ⟨capital identifier⟩ `(` ⟨attributes⟩ `)` ⟨with id⟩

⟨declare⟩ ⊨ `declare` | `declare fact`

⟨attributes⟩ ⊨ ⟨attribute⟩ ⟨attributes tail⟩ | ϵ

⟨attributes tail⟩ ⊨ `,` ⟨attribute⟩ ⟨attributes tail⟩ | ϵ

⟨attribute⟩ ⊨ ⟨lower identifier⟩ `:` ⟨attribute type⟩

⟨attribute type⟩ ⊨ `int` | `float` | `bool` | `string`

⟨with id⟩ ⊨ `with id` ⟨digits⟩

3.2.2 *Basic rule structure*

The outline of the rule changes significantly in shape, but not in meaning. There are still four section, but they are renamed and reordered. Pattern detection clause at the beginning, then filtering, followed by definition and assignment, at the end event consumption.

⟨rule⟩ ⊨ ⟨from⟩ ⟨where⟩ ⟨emit⟩ ⟨consuming⟩

3.2.3 *From clause*

The *from* clause is moved at the beginning of the rule since, since the following sections rely on the tuples matched during this phase. As before it's composed by a sequence of predicates, that now can refer to static data.

⟨from⟩ ⊨ `from` ⟨predicate body⟩ ⟨predicates⟩

$$\begin{aligned} \langle \text{predicates} \rangle &\models \texttt{and}\ \langle \text{predicate} \rangle\ \langle \text{predicates} \rangle \mid \epsilon \\ \langle \text{predicate} \rangle &\models \langle \text{event} \rangle \mid \langle \text{aggregate} \rangle \mid \langle \text{static} \rangle \end{aligned}$$

3.2.4 *Where clause*

The keyword *where* loses its previous meaning of assignment and gets closer to the SQL-like interpretation. The clause is composed by a sequence of boolean expression that filter the propagation of the event when the *from* pattern is satisfied.

$$\langle \text{where} \rangle \models \texttt{where}\ \langle \text{filters} \rangle \mid \epsilon$$

$$\begin{aligned} \langle \text{filters} \rangle &\models \langle \text{expression} \rangle\ \langle \text{filters tail} \rangle \mid \epsilon \\ \langle \text{filters tail} \rangle &\models \texttt{and}\ \langle \text{expression} \rangle\ \langle \text{filters tail} \rangle \mid \epsilon \end{aligned}$$

3.2.5 *Emit clause*

The old clause *define* loses part its meaning because of event declaration, that now specify in advance all the events attributes, so it is joined to the old *where* in the *emit* clause. This section is composed by the name of the complex event and a chain of variable assignment.

$$\langle \text{emit} \rangle \models \texttt{emit}\ \langle \text{capital identifier} \rangle\ \langle \text{evaluations} \rangle$$

$$\begin{aligned} \langle \text{evaluations} \rangle &\models \texttt{(}\ \langle \text{evaluation} \rangle\ \langle \text{evaluations tail} \rangle\ \texttt{)} \mid \epsilon \\ \langle \text{evaluations tail} \rangle &\models \texttt{,}\ \langle \text{evaluation} \rangle\ \langle \text{evaluations tail} \rangle \mid \epsilon \\ \langle \text{evaluation} \rangle &\models \langle \text{lower identifier} \rangle\ \texttt{=}\ \langle \text{expression} \rangle \end{aligned}$$

3.2.6 *Consuming clause*

Event consumption preserves its position and syntax.

$$\langle \text{consuming} \rangle \models \texttt{consuming}\ \langle \text{capital identifier} \rangle\ \langle \text{capital identifiers} \rangle \mid \epsilon$$

3.2.7 *Predicate body*

To satisfy the goal of simplicity, we tried to reach a data format unification as tuples. That may not seem important, but it helps handling events and static data as similarly as possible. The predicate body,

which describes tuple filtering, parameter assignment and alias definition, is one of main example this uniformity.

⟨predicate body⟩ ⊨ ⟨capital identifier⟩ ⟨assignments⟩ ⟨constraints⟩ ⟨alias⟩
⟨assignments⟩ ⊨ `[` ⟨assignment⟩ ⟨assignments tail⟩ `]` | ϵ
⟨assignments tail⟩ ⊨ `,` ⟨assignment⟩ ⟨assignments tail⟩ | ϵ
⟨assignment⟩ ⊨ ⟨parameter⟩ `=` ⟨expression⟩
⟨constraints⟩ ⊨ `(` ⟨expression⟩ ⟨constraints tail⟩ `)` | ϵ
⟨constraints tail⟩ ⊨ `,` ⟨expression⟩ ⟨constraints tail⟩ | ϵ
⟨alias⟩ ⊨ `as` ⟨capital identifier⟩ | ϵ

3.2.8 *Event*

Starting from the common predicate body, events are characterized by windowing information and four types of selection, where *first* and *last* are based on the implicit ordering given by notification timestamps.

⟨event⟩ ⊨ ⟨event selection⟩ ⟨predicate body⟩ ⟨timing⟩
⟨event selection⟩ ⊨ `each` | `not` | `first` | `last`

3.2.9 *Aggregates*

Aggregates can be applied to both a window of events or to a collection of static data, with almost no difference.
Moreover in this statement it isn't possible any more to express comparisons and constraints, instead we can assign the resulting value to a parameter and then use it in the where clause.

⟨aggregate⟩ ⊨ ⟨aggregate assignment⟩ ⟨aggregate body⟩
⟨aggregate assignment⟩ ⊨ ⟨parameter⟩ `=` | ϵ
⟨aggregate body⟩ ⊨ ⟨aggregator⟩ `(` ⟨constrained tuple⟩ ⟨aggregate timing⟩ `)`
⟨aggregator⟩ ⊨ `AVG` | `SUM` | `MAX` | `MIN` | `COUNT`
⟨constrained tuple⟩ ⊨ ⟨capital identifier⟩ `(` ⟨constraints⟩ `)` ⟨attribute selection⟩
⟨aggregate timing⟩ ⊨ ⟨timing⟩ | ϵ
⟨attribute selection⟩ ⊨ `.` ⟨lower identifier⟩ | ϵ

3.2.10 *Static*

Static predicates are really similar to event predicates, as we mentioned before, they share the core of parameters assignment and filtering, but also the selection policies. There is a catch though: static tuples don't have timestamps, so no need for windowing and no natural ordering. The lack of an order implies that whenever we want to

use the operators *first* and *last* we have to describe how to sort data using their attributes in a SQL-like fashion.

⟨static⟩ ⊨ ⟨unordered static⟩ | ⟨ordered static⟩
⟨unordered static⟩ ⊨ ⟨unordered selection⟩ ⟨predicate body⟩
⟨unordered selection⟩ ⊨ `each` | `not`
⟨ordered static⟩ ⊨ ⟨ordered selection⟩ ⟨predicate body⟩ ⟨ordered by⟩
⟨ordered selection⟩ ⊨ `first` | `last`
⟨ordered by⟩ ⊨ `ordered by` ⟨ordering⟩ ⟨orderings⟩
⟨ordering⟩ ⊨ ⟨lower identifier⟩ ⟨order⟩
⟨orderings⟩ ⊨ `,` ⟨ordering⟩ ⟨orderings⟩
⟨order⟩ ⊨ `asc` | `desc`

3.2.11 *Timing*

⟨timing⟩ ⊨ ⟨within⟩ | ⟨between⟩
⟨within⟩ ⊨ `within` ⟨time⟩ `from` ⟨capital identifier⟩
⟨between⟩ ⊨ `between` ⟨capital identifier⟩ `and` ⟨capital identifier⟩
⟨time⟩ ⊨ ⟨float⟩ ⟨time unit⟩
⟨time unit⟩ ⊨ `d` | `h` | `min` | `s` | `ms` | `us`

3.2.12 *Expressions and constraints*

⟨expression⟩ ⊨ ⟨parenthesization⟩ | ⟨operation⟩ | ⟨atom⟩
⟨parenthesization⟩ ⊨ `(` ⟨expression⟩ `)`
⟨operation⟩ ⊨ ⟨binary operation⟩ | ⟨unary operation⟩
⟨binary operation⟩ ⊨ ⟨expression⟩ ⟨binary operator⟩ ⟨expression⟩
⟨unary operation⟩ ⊨ ⟨unary operator⟩ ⟨expression⟩
⟨binary operator⟩ ⊨ *Common algebraic and comparison operators*
⟨unary operator⟩ ⊨ *Common unary operators*
⟨atom⟩ ⊨ ⟨identifier⟩ | ⟨parameter⟩ | ⟨immediate⟩
⟨identifier⟩ ⊨ ⟨qualifier⟩ ⟨lower identifier⟩
⟨qualifier⟩ ⊨ ⟨capital identifier⟩ `.` | ϵ

3.2.13 *Basic types*

⟨capital identifier⟩ ⊨ *An identifier starting with an uppercase letter*
⟨lower identifier⟩ ⊨ *An identifier starting with a lowercase letter*
⟨parameter⟩ ⊨ `$` ⟨lower identifier⟩
⟨capital identifiers⟩ ⊨ `,` ⟨capital identifier⟩ ⟨capital identifiers⟩ | ϵ
⟨lower identifiers⟩ ⊨ `,` ⟨lower identifier⟩ ⟨lower identifiers⟩ | ϵ
⟨immediate⟩ ⊨ *An immediate value, like a digit*
⟨float⟩ ⊨ *A floating point number*

3.3 SEMANTIC OF RULES

3.3.1 *Extension of basic concepts*

In the previous chapter we introduced the concept of *label*, the TRIO function $lab(Rule, Labels)$, the predicate $Occurs(Type, Label)$ and the function $attVal(Label, Attribute)$.
Before we continue further, we have to adjust those concepts to take into account static data.

Labels and uniqueness of selection

We can extend the meaning of labels to identify data too, so that a label can be associated to an event notification or to a tuple and only one of them.
In the same way we do for simple events, we assume the static collections to have been already labeled externally and so the function *lab* remains unchanged, making no distinction between what a label is associated to.
The *uniqueness of selection* remains valid, because a trigger event is still required and that assures at least a different label for every new execution.

Definition of predicate ThereIs

The predicate $Occurs(Type, Label)$ is not applicable, since datasets remain unchanged in every time instant, so we define the predicate $ThereIs(Type, Label)$ with a similar meaning: it associates a label to a single tuple type.

$$Alw(\forall s_1, s_2 \in S, \forall l \in L\ ((ThereIs(s_1, l) \wedge ThereIs(s_2, l)) \rightarrow s_1 = s_2))$$

If the label is associated to static data it cannot be tied to an event.

$$Alw(\forall s \in S, \forall e \in E, \forall l \in L\ (ThereIs(s, l) \rightarrow Alw(\neg Occurs(e, l))))$$

Differently from *Occurs*, this predicate is time independent, so if it is satisfied in one instant it has to be true in any other.

$$Alw(\forall s \in S, \forall l \in L\ (ThereIs(s, l) \rightarrow Alw(ThereIs(s, l))))$$

Where L is the set of all labels, E the set of all event types and S the set of all static data types.

Extension of predicate attVal

The function *attVal* doesn't need to be syntactically modified, but should be extended to return attributes values of static tuples as well.

3.3.2 *Specification*

Event composition

Since the first predicate has to be a trigger event, it is pointless to make examples that do not include at least an additional predicate. So the base case is a complex event CE generated every time the event SE is notified and repeated for each entry found in the dataset SD.

$$from\ SE\ and\ each\ SD\ emit\ CE \triangleq$$

$$Alw(\forall l_0, l_1 \in L\ (Occurs(CE, lab(r, \{l_0, l_1\}) \leftrightarrow (Occurs(SE, l_0) \wedge ThereIs(SD, l_1))))$$

Using *not* operator, a complex event is emitted only if there are no acceptable entries in the collection SD.

$$from\ SE\ and\ not\ SD\ emit\ CE \triangleq$$

$$Alw(\forall l_0 \in L\ (Occurs(CE, lab(r, \{l_0\}) \leftrightarrow (Occurs(SE, l_0) \wedge \neg\exists l_1 ThereIs(SD, l_1))))$$

When we use the operators $first$ or $last$, we have to define an ordering so that the concept of coming first or last acquire its meaning. We assume single attributes to be comparable. The choice between the keywords *asc* and *desc* determine the direction of the comparison operator. Ties are broken using the label to impose a total ordering.

$$from\ SE\ and\ first\ SD\ order\ by\ att_1\ asc\ emit\ CE \triangleq$$

$$\begin{aligned} &Alw(\forall l_0, l_1 \in L\ (Occurs(CE, lab(r, \{l_0, l_1\})) \leftrightarrow \\ &(Occurs(SE, l_0) \wedge ThereIs(SD, l_1) \wedge \\ &(\neg\exists l_2 \in L\ (ThereIs(SD, l_2) \wedge ((attVal(l_2, att_1) < attVal(l_1, att_1)) \\ &\vee (attVal(l_2, att_1) = attVal(l_1, att_1)) \wedge (l_2 < l_1)))))) \end{aligned}$$

When there are multiple attributes in the ordering clause, they are checked in with lexicographical priority as in SQL.

$$from\ SE\ and\ first\ SD\ order\ by\ att_1\ asc,\ att_2\ desc\ emit\ CE \triangleq$$

$$\begin{aligned} &Alw(\forall l_0, l_1 \in L\ (Occurs(CE, lab(r, \{l_0, l_1\})) \leftrightarrow \\ &(Occurs(SE, l_0) \wedge ThereIs(SD, l_1) \wedge \\ &(\neg\exists l_2 \in L\ (ThereIs(SD, l_2) \wedge ((attVal(l_2, att_1) < attVal(l_1, att_1)) \\ &\vee (attVal(l_2, att_1) = attVal(l_1, att_1)) \wedge \\ &((attVal(l_2, att_2) > attVal(l_1, att_2)) \vee \\ &(attVal(l_2, att_2) = attVal(l_1, att_2)) \wedge (l_2 < l_1)))))))) \end{aligned}$$

Filtering in static data selection is the same as in event selection.

$$from\ SE\ and\ each\ SD(expr(att_1))\ emit\ CE\ \triangleq$$

$$Alw(\forall l_0, l_1 \in \text{Ł}\ (Occurs(CE, lab(r, \{l_0, l_1\}))\ \leftrightarrow \\ (Occurs(SE, l_0)\ \wedge\ ThereIs(SD, l_1)\ \wedge\ Pred(attVal(l_1, att_1)))))$$

Parameterization

Parameters have the new assignment syntax, but semantically preserve their meaning and expressivity.

$$from\ SE[\$par = att_i]()\ and\ each\ SD(att_j == \$par)\ emit\ CE\ \triangleq$$

$$Alw(\forall l_0, l_1 \in L\ (Occurs(CE, lab(r, \{l_0, l_1\}))\ \leftrightarrow\ (Occurs(SE, l_0) \\ \wedge\ ThereIs(SD, l_1)\ \wedge\ (attVal(l_0, att_i)\ =\ attVal(l_1, att_j)))))$$

Aggregates

To perform filtering using aggregates values now we have to make use of parameters assignment and the *where* clause.

$$from\ SE\ and\ \$par = AGGR(SD.att_1) \\ where\ expr(\$par)\ emit\ CE\ \triangleq$$

$$Alw(\forall l_0 \in L\ (Occurs(CE, lab(r, \{l_0\}))\ \leftrightarrow\ (Occurs(SE, l_0) \\ \wedge\ \forall S\ (\forall s\ (s \in S\ \leftrightarrow\ \exists l \in L\ (s\ =\ \langle l,\ attVal(l, attr_1)\rangle\ \wedge \\ ThereIs(SD, l_1)))\ \rightarrow\ Pred(aggr(S))))))$$

Additional remarks

Also attribute assignment in the *emit* clause has to make use of parameters: they are now the only notation to refer to information from another predicate or clause.

Moreover, while with the selection policy *each* may not be noticed, there is a semantic difference between filtering in the selection predicate or filtering in the where clause. The distinction appear in combination with *first* and *last*: if filtering in the selection, the engine will search for the first (resp. last) tuple matching the constraints, if filtering inside the *where* statement, the engine will retrieve the absolute first (resp. last) tuple and then apply the filter possibly preventing the event generation.

4

IMPLEMENTATION

4.1 INTRODUCTION

Given the real-time nature of a CEP engine and the extensive uptime typical of a server, a thorough implementation is of maximum importance.

In this chapter I will explain how the project is structured and how it evolved in time, highlighting the choices made to find a balance between performance and convenience.

4.2 OVERVIEW

The T-Rex project, in its entirety, is composed of the engine library, a server, a client and an HTTP proxy. The engine, written in C++ and CUDA, is the foundation of the whole system and contains the business logic and the computationally intensive tasks. The server, written in C++ as well, links the core library and provides a network interface on top of it, receiving and dispatching packets for rules and events. The client, written in Java, allows full interaction with the CEP server via TCP/IP sockets: from publish-subscribe, to rule registration, using a TESLA parser created with ANTLR. The proxy, written in JavaScript for NodeJS, exposes an HTTP interface for publishing and subscribing, but at the moment doesn't implement rule parsing and registration. For the purpose of the thesis we will focus on the library, since its is the only one relevant in terms of feasibility and performance of static data integration.

At the beginning of the collaboration the T-Rex engine was a reasonably complex and well performing piece of software, although it showed several signs of its age. It was crafted over different iterations starting from 2010 and at that time C++ was still shaped according to its 12 years old original standardization. The very next year the C++11 standard came along, beginning an incredible period of renovation for both the language and the libraries, and the adoption of these new paradigms have the potential of a great improvement of safety and extensibility.

In particular the broad usage of dynamically allocated objects required exceptional caution during refactoring, because any minimal

oversight could lead to leaks or attempts to access freed memory. In T-Rex it was handled with manual reference counting, which is now discouraged in favor of `shared_ptr`, that uses RAII to relieve the programmer from the responsibility of updating the count.
Similarly a big part of the execution relied on the combination of type unions, enums and switches to describe and process TESLA expressions. The access to the wrong type or the absence of a switch case can cause bugs that aren't detected by the compiler and lead to unexpected runtime failures. In the upcoming C++17 the type *variant* will be added to the standard library and it uses the power of template metaprogramming to prevent those issues at compile time.
Moreover threading utilities and patterns are continuously evolving and they offer new levels of abstraction that reduce the needs of synchronization and locking, improving performances and preventing data races and deadlocks.
In addition to those safety benefits, there are several small improvements in terms of comprehensibility: like replacing array pointers with vectors, avoiding output arguments now that compilers handle efficiently the return statement, abandoning the cumbersome naming (inherited from the C tradition) for a wise usage of namespaces, adopting foreach loops and making use of idiomatic std functions.

These innovations highlighted the weak points of the original implementation and a renewal of the code base was deemed to be a collateral requirement of the broad modifications that the library was going to face. So, after an initial attempt of progressive refactoring, I proceeded with a complete rewrite of the engine in *Rust*, a new language which offers the aforementioned advantages and even more.

4.3 RUST

Rust was born around 2010 as a side project of a Mozilla engineer and later backed by the company. The first pre-alpha release was reached in 2012 and the project hit version 1.0 in May 2015. So the language is pretty young and still missing some of its planned features, but many signs suggest that it may be on the right path.
First of all, while in the past there were several radical changes, after they reached the release 1.0 they committed to stability and backward compatibility, adopting a well defined workflow based on reference proposals. However the project has kept evolving quickly, with a release train model over 3 months windows.
Moreover it has raised a lot of interest within the community and there is a strong traction from a big company, ensuring continuity. Finally it is already used in production by Mozilla itself, Dropbox and Coursera among the others.

Rust aims to be a safe and practical language for system programming, with particular attention to concurrency. Its syntax, modern

and expressive, combines the best aspects of imperative, object oriented and functional programming, offering the right level of abstraction for many different task.
The type-system, inspired by Haskell, is based on the concept of trait, which describe a property or a behavior of an object. Traits achieve their maximum utility in combination with generics, allowing code reuse while imposing clear restrictions on the applied types.
Rust also implements some features typical of higher level languages, like advanced union types (enums in Rust jargon), tuples, type inference, destructuring and pattern matching.
However the most prominent and peculiar characteristic is its unprecedented approach to safety with the concept of data ownership. Each object is tied to a single owner at a time, ownership can be transferred via assignment, return or function arguments and the content is moved and no longer accessible from the previous variable. If we want to access data without loosing ownership, it's possible to borrow multiple immutable references or a single mutable one, in the meantime the variable is considered blocked in something conceptually similar to a read-write lock. Every reference is bounded by the lifetime of the referee and can't outlive the owner. In this way is always clear who is responsible for the resource and when the owner goes out of scope the object can be safely dropped.
All these constraints and other minor ones are statically checked by the compiler, which gives strong guarantees of memory safety with no runtime overhead.

Last but not least it worth mentioning that there is a growing ecosystem of tools and utilities, that ease setup and development. For example: rustup.rs is an automated setup and update script, cargo is a modern and simple package manager, build tool and documentation generator, crates.io is the official repository of open source libraries, rustfmt is a customizable code beautifier and racer is a code completion utility. It's also thanks to this simplicity of bootstrap and distribution, that I was persuaded to adopt this new technology.

4.4 ARCHITECTURE

The T-Rex rewrite, *T-Rex2* from now on, is composed by three crates (Rust jargon for packages): `tesla`, that contains basic engine interfaces and language structures, `trex`, that is the implementation of those traits, and `benches`, a set of executables to test performances under simulated workload (on which we will focus in next chapter).

4.4.1 *Tesla*

In tesla package I extracted all the aspects that aren't implementation related, so that it could work as minimal contact point between

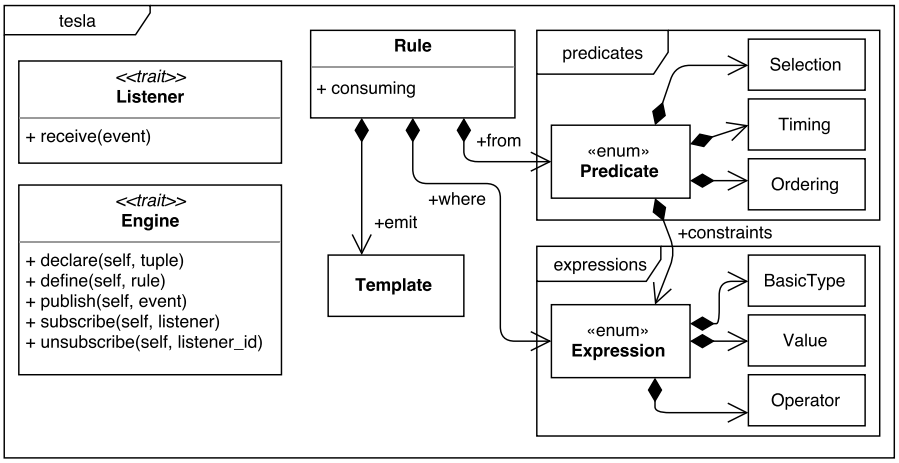

Figure 4.1: Tesla crate overview

components like the library and the server. The core of the package is made of two traits: `Engine`, that defines methods for publish-subscribe, tuple declaration and rule definition, and `Subscriber`, that is used for event reception. Starting from these entry points the rest of the data structures just follows from the information required: in particular we have two sub-packages `predicates` and `expressions`, the first contains all the tools to describe patterns of events and static data, the latter contains the *Abstract Sytax Tree* (AST) of algebraic and comparison expressions. They both heavily leverage rust native union type, giving them a terse design compared to the possible equivalent with C++ `variant`.

4.4.2 *TRex*

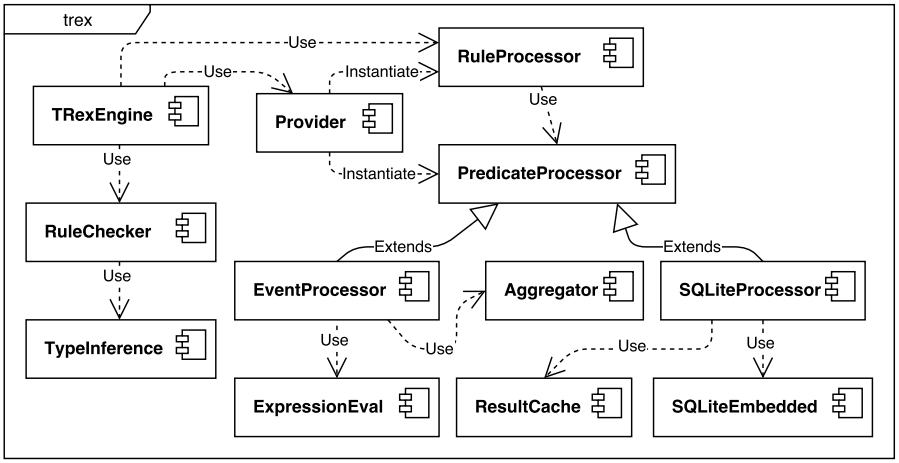

Figure 4.2: TRex crate overview

The structure of `trex` crate closely recalls the previous implementation, with some simplification and new abstractions.

The core is the `TRex` struct, that implements the `Engine` trait and acts as a coordinator between the different components and the external world. The rest of the code can be divided in two main functionalities: the rule definition and the event processing.

The rule definition procedure is basically a sequence of configurations, that does not affect the performance of execution. When a new rule is received, `TRex` invokes the validation module and types, inferred from the collected tuple declarations, are checked to be used properly in expressions and assignments. If the rule passes the examination, a factory pattern is used to instantiate a dedicated `RuleProcessor`. Then the processor is indexed by the events that participate to the definition of the rule, so that is going to be activated only by those notifications that are relevant to its evaluation. In the original project the index was more sophisticated than a single hash-map and had a mechanism to filter the matches by checking those predicate constraints that only depend on immediate values. In practice the alternative appears to work well enough and additional optimization could be added if needed.

Event handling, instead, is the core of the business logic. Event notifications are sent to the `TRex` instance through the `publish` method, presented in pseudo-code in listing 4.1. First the event is forwarded to the registered subscribers, then the index of rule processors is accessed and for each match a task is scheduled to execute in a thread pool, at this point the method is blocked waiting for tasks completion, finally if the evaluation does not produce new events the execution is complete otherwise the new events are submitted recursively to `publish` and the execution repeated. Since there is the assumption that incoming events are presented in time order and the blocking code prevent the next event from being processed, the time order for the generated complex events is guaranteed.

```
1  function publish(event):
2    for_each subscriber in subscribers:
3      subscriber.notify(event)
4    done
5
6    let processors = index.find_by(event.id)
7    for_each processor in processors:
8      thread_pool.execute { processor.process(event) }
9    done
10
11   let events = thread_pool.collect_results()
12   for_each event in events:
13         self.publish(event)
14   done
15 end
```

Listing 4.1: TRex publish method

Going deeper, the `RuleProcessor`, which correspond to `StackRule` class in C++, is the coordinator of a sequence of `PredicatesProcessor`

and its main method (in listing 4.2) is a generalized version of the Column-based Delayed Processing (CDP) algorithm that, as mentioned in chapter 2, is at the core of T-Rex library itself.
On event reception `RuleProcessor` is responsible of dispatching the notification to the predicates and, if the trigger is satisfied, of propagating the chain of evaluation. The evaluation is based on a list of objects called `PartialResult`, each them is a representation of sequence of tuples (events or static data) that are compatible with the pattern of predicates has been so far verified. Every step in the propagation consumes the list of partial results, filters it and enriches it with the predicate inner information, with the result that the list can be shrunk, expanded or dropped altogether. Once the evaluation is complete and successful, the `RuleProcessor` verifies the last constraints of the where clause and builds the events to emit from a template, possibly handling the consumption of some of the events used in the process.

```
function process(event) -> [event] :
  for_each predicate in predicates_processors:
    predicate.process(event)
  done

  results = []
  if trigger.is_satisfied(event):
    time = event.time;
    for_each predicate in predicates_processors:
      time = predicate.remove_old(time)
    done

    partial_results = []
    for_each predicate in predicates_processors:
      partial_results =
        predicate.evaluate(partial_results)
    done

    for_each partial_result in partial_results:
      if where_clause.is_satisfied(partial_result):
        results.push(template.fill_with(partial_result)
        mark_for_consumption(partial_result)
      fi
    done
  fi

  return results
end
```

Listing 4.2: RuleProcessor process method

In the previous version there were, clearly, only event based predicates and their different types (in term of event selection, aggregates and negations) were often coupled with business logic and in particular `StackRule` had to handle all of them explicitly, with specific functions and separate collections. T-Rex2 introduces a new abstraction, that is the aforementioned trait `PredicateProcessor` which

is implemented by any component that acts as a predicate evaluator. In this way everything is handled uniformly: event and static processors are instantiated through a provider, for further decoupling, and kept into a unique list. Every time that a notification arrives all the elements of the list are notified and each of them decide how to handle the information. Similarly, during evaluation, the only mean of communication is through `PartialResult` and the nature of each step remains encapsulated.
This design choice makes it much easier to experiment with new data sources and hopefully this modularity will facilitate future development of custom components for different DBMS.

Currently the system runs two implementations of predicate processor: `EventProcessor`, which supports the previous functionalities and completes the CDP algorithm as in the original paper [5], and `SQLiteProcessor`, that is the test implementation to interact with a relational database.
The `EventProcessor`, at every notification, filters the received event by ID and checking the constraint that depend only on immediate values, if everything matches the event is stored into a time ordered vector, which is periodically cleaned from old or consumed entries. When the evaluation starts and a `PartialResult` is received, the processor scans the events and propagates those that match every parameterized constraint.
The `SQLiteProcessor`, instead, doesn't interact with event notifications and it's activated only during the evaluation phase, when it queries the database to find suitable tuples, as we will see in the following section.

An example of rule processor is shown in figure 4.3, where the rule is composed in sequence by a trigger, two event predicates and a static predicate. During the execution the two event processors have collected a stack of events, instead the SQLite processor has direct access to a DB table.

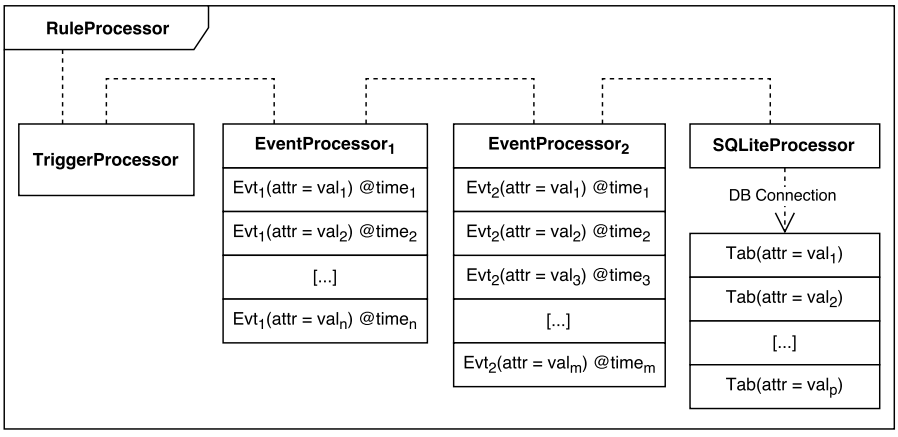

Figure 4.3: Processors chain example

4.5 SQLITE MODULE

The SQLite module contains two main components: a translator and an executor.

At creation time the predicate is fed to the translator and TESLA syntax is mapped permanently to a SQL statement using the following simple conversion rules.
The basic example is composed of an each predicate with no constraints nor parameters and, as we can see, the statement can be translated to a simple SQL select.

$$\begin{aligned} &\textit{each SD} \triangleq \\ &\textit{SELECT } 1 \textit{ FROM SD;} \end{aligned}$$

When the predicate features a negation, the operator `NOT EXIST` can be used to evaluate a subquery and return whether there are values in the result set or not.

$$\begin{aligned} &\textit{not SD} \triangleq \\ &\textit{SELECT NOT EXIST } (\textit{SELECT } 1 \textit{ FROM SD}); \end{aligned}$$

The selection policy *first* does not have an implicit absolute ordering to work with, so it has to define one. The order clause is directly mapped to the SQL one.

$$\begin{aligned} &\textit{first SD order by } attr_1 \textit{ ASC, } \ldots, \textit{ } attr_n \textit{ DESC} \triangleq \\ &\textit{SELECT id FROM SD} \\ &\textit{ORDER BY } attr_1 \textit{ ASC, } \ldots, \textit{ } attr_n \textit{ DESC LIMIT } 1; \end{aligned}$$

For *last* the same considerations made for *first* hold, except for the mapping of the order clause, which is inverted.

$$\begin{aligned} &\textit{last SD order by } attr_1 \textit{ ASC, } \ldots, \textit{ } attr_n \textit{ DESC} \triangleq \\ &\textit{SELECT id FROM SD} \\ &\textit{ORDER BY } attr_1 \textit{ DESC, } \ldots, \textit{ } attr_n \textit{ ASC LIMIT } 1; \end{aligned}$$

The use of constraints inside tuple brackets find its correspondence in the SQL where clause.

$$\begin{aligned} &\textit{each SD}(attr_1 \textit{ op } val_1, \ldots, attr_n \textit{ op } val_n) \triangleq \\ &\textit{SELECT id FROM SD} \\ &\textit{WHERE } attr_1 \textit{ op } val_1 \textit{ AND } \ldots \textit{ AND } attr_n \textit{ op } val_n; \end{aligned}$$

The definition of a parameter it's processed adding its value to the query return column.

$$\begin{aligned} &\textit{each SD}[\$param = attr_i, \ldots] \triangleq \\ &\textit{SELECT } attr_i \textit{ as param, } \ldots \textit{ FROM SD;} \end{aligned}$$

Finally the most common aggregation functions can be found in both the languages, so a translation is immediate.

$$\$param \; = \; AGGR(SD.attr_1) \; \triangleq$$
$$SELECT \; AGGR(attr_1) \; as \; param \; FROM \; SD$$

The prepared statement produced is then stored for later use.

When the evaluation start, the executor populates the query with the parameters contained in `PartialResult` and execute it through `rusqulite` library (that is a wrapper of C SQLite embedded API) and the result are processed and forwarded in the chain of evaluation.

4.6 CACHE

To improve performance of data retrieval and try to keep the speed of execution above the constraints of real-time, a caching layer was added between the executor and the database.

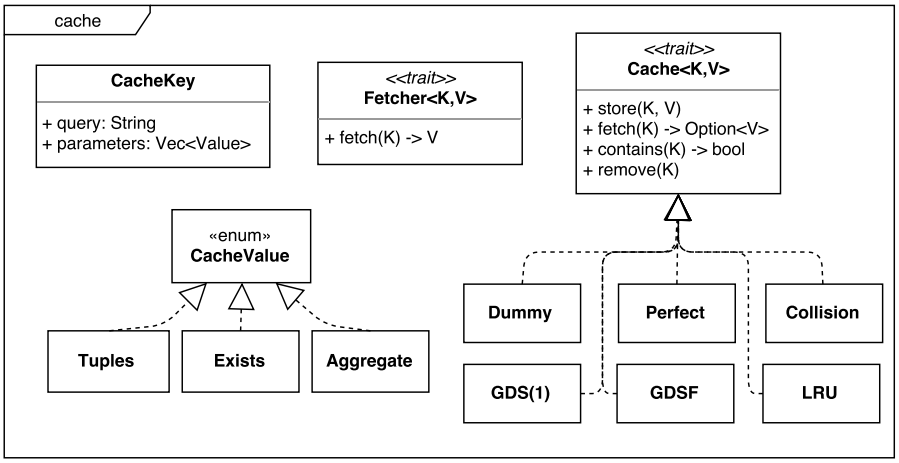

Figure 4.4: Cache module overview

The key of the cache is made by the combination of a SQL statement and the parameters with which it was populated. The key is intuitively unique for each result set[1] and it is also descriptive of the interrogation itself. So, instead of contacting the cache and database directly, the `SQLiteProcessor` will construct a key and delegate the task to a `Fetcher`. This new component will attempt a lookup in the cache and if the result is missing will fall back to the DB, hiding the complexity of cache update and multi-threaded synchronization. The cache value is the result of a previously executed query and, depending from the type of the predicate, it can be a list of tuples or the result of an aggregate function or a flag of existence.

The system is integrated with several cache algorithms that can be activated and configured through the engine construction arguments.

1 For the purpose of the thesis the data are considered immutable and cache invalidation is omitted

Each of them, as can be seen in figure 4.4, implements `Cache` trait allowing to easily replace one implementation with another.

4.6.1 *Simple Caches*

We can categorize the first three algorithm as simple caches, because they have basic storage approaches and do not leverage the context information to chose what to preserve and what to discard.

DUMMY The first implementation is a dummy cache that doesn't store anything and it's useful for testing the system in condition of 100% misses. The methods `store` and `remove` are just empty, while `contains` will always return false and `fetch` will always `None`.

PERFECT The perfect cache is the opposite of the dummy and will store everything, without capacity limits. Its purpose is to test the maximum theoretical benefits of a cache. The methods simply forward the calls to those of an hashmap.

COLLISION The collision cache is basically an hashmap with a reduced key space and a fixed capacity. The methods `contains`, `fetch` and `remove` act in the same way a normal map would. However the insertion possibly removes a single entry with the same hash value.

4.6.2 *Complex Caches*

The last three algorithms, instead, store metadata about the entries and exploit the context information to provide better chance to preserve useful entries in their limited available capacity.

```
1 function store(key, entry):
2   insert entry
3   while memory usage > capacity
4     remove the entry with least priority
5   done
6 end
```

Listing 4.3: Complex cache insertion

All the algorithms presented, as most of the caches in general, are characterized by an insertion method that operates, as shown in listing 4.3, by storing the new element to cache and removing the lowest priority element until the memory usage is below the allowed capacity. The difference between the three implementations is only the formula they use to compute priorities. All the other methods behave

as in a standard map, except that `fetch` and `contains` function internally update the priority of the accessed element.

LRU-SIZE *Least Recently Used* is a family of caches that keep track of the order of access and prioritize time locality. In its usual implementation is meant for fixed size entries and, since in our case results sets can vary significantly in length we adopted the LRU-size variant. LRU-size has a maximum capacity set in terms of object sizes and, every time a value is inserted, multiple entries can be evicted to make room for the new one, keeping a controlled memory footprint. The cache was implemented on top of a `LinkedHashMap`, that is an hash map whose entries are connected one another in a linked list, having $O(1)$ complexity for each operation.

GDS(1) *Greedy-Dual Size(1)* [2] is a cache developed in the context of web browsers and proxies. GDS(1) rewards small entries, so that in the same capacity will fit more values, increasing the probability of a hit. The priority is computed with the following formula:

$$priority := 1 \ / \ size + clock$$

Where *clock* is an aging factor to avoid stagnation of entries that aren't relevant anymore and it's updated monotonically as the priority value of the last discarded entry.

GDSF *Greedy-Dual Size Frequency* [3], an evolution of GDS(1), is the current champion among the web caching algorithms and implemented, for example, by Squid caching proxy. It tries to consider a combination of how costly it is to obtain the entry, the size it occupies and the frequency it was accessed so far, with the following formula:

$$priority := cost * frequency \ / \ size + clock$$

Where *clock* is the same aging factor seen for GDS(1).

Both GDS(1) and GDSF can be implemented using a expecially crafted combination of a binary heap and an hash map, with amortized $O(1)$ complexity and worst case of $O(log(N))$. However, for simplicity, it has been implemented using a high level combination of a B-tree set and an hash map, with complexity $O(log(N))$.

5

EVALUATION

5.1 INTRODUCTION

CEP engines are used in a variety of scenarios, each of which has different requirements and settings, and that makes performance evaluation a true challenge. In fact there is not a universally agreed methodology of measurement and there are neither a reference workload, recorded from a real execution, nor a standard emulator, to generate a synthetic one.

The complexity is given by the high number of parameter that characterize the execution of the application and in particular by the different ways in which they interact.
In a real world scenario most of those variables are bound to the environment and very specific: the inputs are correlated one each other and together deliver a meaningful information, at the same time the rules are manually tailored to the particular task.
On the opposite side, during a general purpose evaluation, it is necessary to control the behavior of the inputs and the complexity of the rules, while preserving the stability of the execution.

In this chapter I will present the results of a selected number of test cases, explaining why each configuration was chosen and how it could impact on a real world application.

5.2 ENVIRONMENT

Processor:	Intel Core i7-4770 @ 3.40GHz 4 Cores, 8 Threads
RAM size:	16GB
Operating System:	Debian GNU/Linux 8.6 (jessie) Kernel 3.16.0-4-amd64
C++ Compiler:	G++ 4.9.2
Rust Compiler:	1.14.0-nightly (2016-11-05)

5.3 GENERAL PERFORMANCES

First of all, setting temporarily aside the introduction of static data, we will compare the previous T-Rex engine with the rewrite T-Rex2. This will show that the new implementation is indeed correct and efficient, mitigating the risk of a bias in the tests due to a poor realization. In the meantime we will gradually introduce the key points of the evaluation process and the general environment setup.

5.3.1 *Characteristic variables*

We start from the fundamental variables that characterize a CEP evaluation, in particular explaining their on the computation.

- The frequency of events and the size of predicate window are two of the most characteristic control variables and, in combination one with each other, they regulate the amount of events retained by the system. So higher frequencies and wider windows imply that more events will be associated with every predicate processor: those events have to be searched at each iteration of the system and directly concur in rules satisfaction.
- The number of rules intuitively impact the computational requirements increasing the number of step needed to process an incoming event. However this aspect is strongly related with the number of declared event types and since the latter influence the probability of a rule being activated by the next random event. So a high number of rules composed by many different event types, may be activated just one at a time and stress the system way less than the half of the rules with fewer event declarations.
- The number of predicates per rule and the presence of constraints on them influence the probability of a rule to be satisfied. In combination with the selection policy, measured in terms of probability of being *each*, *first* or *last*, they determine how many new events will be generated by each rule.
- Finally we mention the two most important output variables: the drop rate and the time of completion. The drop rate analyzes the system in a latency oriented fashion and it is measured as the percentage of discarded events: we feed the engine through a buffer queue of finite length and if it is not emptied fast enough the exceeding events are lost. On the other hand the completion time is more throughput oriented and it is obtained using an unbounded queue and waiting for the system to process every single event.

5.3.2 *Base rule*

As for the rule definition, the challenge is to create a model that is simple enough to predictably work under all the circumstances examined and rich enough to be interesting and customizable. We found the following rule to be representative of the different aspects we care about, while being easy to extend.

$$
\begin{aligned}
&\textit{declare } SE_0(x:\ int) \textit{ with id } 0\\
&\textit{declare } SE_1(x:\ int) \textit{ with id } 1\\
&\textit{declare } \ldots\\
&\textit{declare } SE_i(x:\ int) \textit{ with id } i\\
&\textit{declare } CE() \textit{ with id } i+1
\end{aligned}
$$

$$
\begin{aligned}
&from\ SE_0(x == 1)\\
&\textit{and each } SE_1(x == 1) \textit{ within } \tau_1\ from\ SE_0\\
&\textit{and } \ldots\\
&\textit{and each } SE_i(x == 1) \textit{ within } \tau_i\ from\ SE_{i-1}\\
&\textit{emit } CE
\end{aligned}
$$

The rule is composed of a linear chain of simple events from SE_0 to SE_i, where every predicate has a temporal dependency with the previous one and the number of predicates can be extended at will. Each constraint is composed of a single equality to the immediate value 1, this configuration allows to simply control the match through the choice of event values. The time window is parameterized by τ_i and, while the example shows only the use of *each* selection policy, they can be varied with a given probability.
Exactly $i+1$ tuples were declared, so that each one can only appear in a specific position in the rule, improving the predictability of the system.

5.3.3 *Workload*

For the benchmark we configured the previous rule with $i = 2$ and τ_i as a random random value uniformly distributed in the range $\tau_{avg} - 1s$ and $\tau_{avg} + 1s$.
The probability of choosing *each*, *first* or *last* selection policies was usually fixed to 100% of *each* to maximize the load of the system.
Regarding the rest of the system, 65 group of 4 event types were declared and 650 rules were instantiated, 10 identical for each group of events, so that every time a sequence was satisfied ten rules would fire. The events were generated uniformly across the different declarations, with all the attributes x set to 1 to satisfy the constraints, and they were emitted at a tunable frequency. The number of events fed

to the system was $60 * freq$, that is the quantity emitted in a minute at the given frequency. The length of the queue was bound as $freq/10$ when measuring the drop rate and left unbound when measuring execution time.

5.3.4 *T-Rex and T-Rex2 comparison*

For the first comparison we set the predicates window to at an average of 10s and varied the frequencies from 600 to 4000 events per second, so that at the minimum both handle all the events, with a drop rate of 0% and at the end of the scale most of them are lost.
Figure 5.1 shows that the drop rate of the rewrite is lower at every frequency and similarly figure 5.2, that plot execution times in second on logarithmic scale, shows that at frequencies where the engines would loose events the speed of the rewrite is almost constantly four times higher.

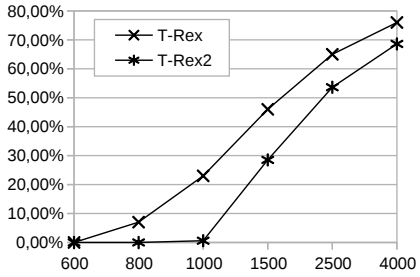

Figure 5.1: T-Rex vs T-Rex2 Frequencies and drop

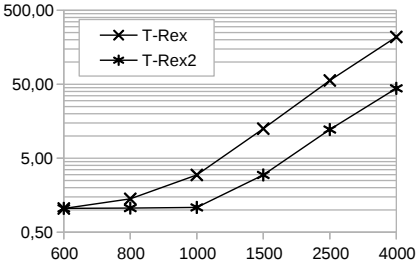

Figure 5.2: T-Rex vs T-Rex2 Frequencies and time

To better explore the domain, for the second comparison we pick a middle frequency, 1500, and vary the average windows size, in a range from 3 to 12 seconds.
The results, shown in figure 5.3 and 5.4, are noticeably similar to the first measurement and confirm the performance gain.
We can also observe how in both the test case the drop rate and the corresponding execution time are closely correlated and most of the time it is possible to switch from one to the other without loosing the qualitative interpretation.

The results demonstrate the efficiency of T-Rex2, with improvements that can be considered beyond the expectations, considering that the rewrite closely followed the architecture of its predecessor. The most likely and relevant motivation is a slightly improved parallelism that take advantage of all the available processing units: they both had 8 workers threads (plus few other to deal with coordination and event publishing), but the old implementation distribute the jobs statically using rule ids, while the rewrite allocate them dynamically with a thread pool.

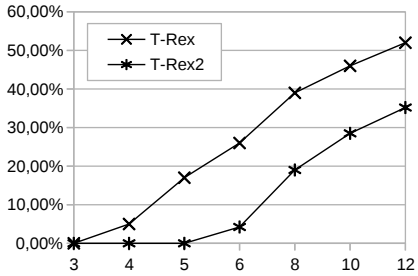

Figure 5.3: T-Rex vs T-Rex2 Windows and drop

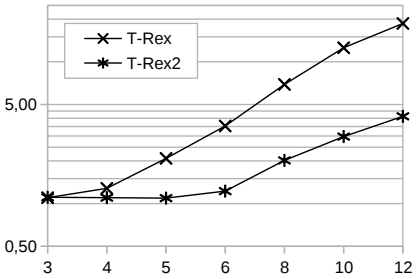

Figure 5.4: T-Rex vs T-Rex2 Windows and time

5.4 STATIC DATA AND CACHE

In this section we are going to explain how the execution must be adapted to evaluate the interaction between events and persistent data and to test the performance gains obtained from the cache.

5.4.1 *Additional variables*

We must refine the domain of the problem, since the new expressivity inevitably causes the expansion of the control variables with the one needed to describe the properties of the database and the cache layer. In particular a database is a complex technology and have plenty of settings and characteristic, but the fundamental ones are: the number of rows, the number of columns, the distribution of the values, the latency of the connection and the presence of indexes. While the integration into the CEP execution is characterized by the selection policy, the constraints on the values, the number of results and the frequency of invocation.
As for the cache, there is the choice of the algorithm among the many presented, the size in terms of memory occupation and finally the possibility to share the storage or split it among the different users. Moreover there are environmental factors that influence the results like the domain and the probabilistic distribution of the values. Finally miss rate and hit and miss time are the characteristic metrics of performance.

5.4.2 *Rule adaptation*

The previous TESLA rule was kept as a template and extended with a static predicate with the following configuration:

$$
\begin{aligned}
&\textit{declare } SE_0(x:\ int) \textit{ with id } 0\\
&\textit{declare } SE_1(x:\ int) \textit{ with id } 1\\
&\textit{declare } \ldots\\
&\textit{declare } SE_i(x:\ int,\ y:\ int,\ z:\ int) \textit{ with id } i\\
&\textit{declare } SE_{i+1}(col_0:\ int,\ \ldots,\ col_j:\ int) \textit{ with id } i+1\\
&\textit{declare fact } SD(col_0:\ int,\ \ldots,\ col_j:\ int) \textit{ with id } i+2\\
&\textit{declare } CE() \textit{ with id } i+3
\end{aligned}
$$

$$
\begin{aligned}
&\textit{from } SE_0(x == 1)\\
&\textit{and each } SE_1(x == 1) \textit{ within } \tau_1 \textit{ from } SE_0\\
&\textit{and } \ldots\\
&\textit{and each } SE_i[\$p_1 = y,\ \$p_2 = z](x == 1) \textit{ within } \tau_i \textit{ from } SE_{i-1}\\
&\textit{and each } SD[\$c_0 = col_0](col_1 >= \$p_1,\ col_1 < \$p_2)\\
&\textit{emit } CE(x = \$c_0)
\end{aligned}
$$

There is a new declaration SD that describes the static data collection and each of its attributes is mapped to a column of the table.
The event SE_i acquire a special role, it has a second and third attribute, that are referenced in the static predicate and they work as lower and upper bound, controlling the result of the query.
Another declaration was introduced, SE_{i+1}, that, as we will see, can be used as a replacement of SD whenever we want to reduce the access to the DB while keeping the same level of produced events.
Finally it is possible, through a configuration, to add a parametric dependency between the static predicate and other predicates in the rule: this allow to expand the domain of the cache key stressing the cache algorithms.

5.4.3 *Workload*

First of all, a database table is created with 2 columns (an incremental index and a numerical payload) and populated with a given number of rows. In particular, for each entry the aforementioned payload is randomly generated in a range between $-1/2 * \#rows$ and $+1/2 * \#rows$: having a uniform distribution of values, but in a non sequential order, helps to avoid possible bias in term of DBMS architecture or file read.
The declaration were changed to 100 groups of 5, plus one single declaration for the database table. The rule defined are 1000, preserving

the ratio of 10 identical rule for each declaration group. Database indexing can be switched on or off.
During the event generation, $SE_i.y$ (the second attribute of the events of type SE_i) is set to a random value, sampled from a selectable probabilistic distribution, while $SE_i.z$ is set as $SE_i.y + \Delta$, so that they act properly as lower and upper bound. Where Δ is generated pseudo-randomly with an average of 10.
What is not mentioned here is unchanged from the previous definition of the workload in section 5.3.3.

5.4.4 *Table simulation and database*

Even without a native support for DBMSs, it was already possible to emulate the access to a persistent data collection, as we recall from section 3.1. So it seems appropriate to verify the integration with SQLite to be consistently faster than preexisting alternative.

So, to define two executions with the very same semantic, we applied the algorithm for DB population to produce events with the same payload of the corresponding table and we emitted all of them at the beginning of the execution. At the same time we adapted the static predicate in each rule extract the same data from the simulated event stream (using an unlimited window, to have the data available at any time).

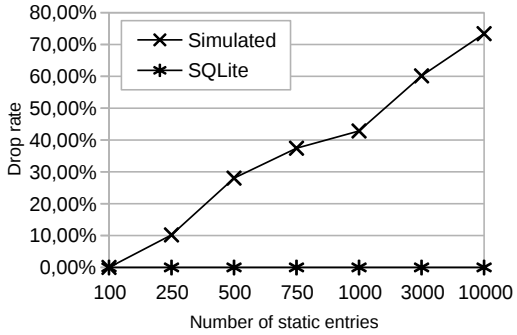

Figure 5.5: Database vs simulation

We execute the programs with a small window of 2s average and low frequency of 800 events per second, varying the number of static data entries from 100 to 10000. As we can see from figure 5.5, the table simulation fully handle the load only with a very small collection and performances rapidly deteriorate as soon the number of static entries grow. On the other hand, T-Rex2 does not lose events even with the highest configuration considered. It is worth noting that, actually, thanks to the benefits of the database index, the performance of T-Rex2 are virtually not influenced by the size of the table (tested up to 10 million rows).

The results show that a native support is clearly an important improvement to the previous possibilities, but at the same time they give us little information about the limits of T-Rex2.

5.4.5 *Performances contextualization*

In the following benchmark we contextualize the measures collected through a comparison of the results of three different configuration that have the same rate of event production, meaning that given the same number of input events they will output a similar number of complex events.
Two out of thee different competitors share exactly the same workload described in section 5.4.3, one uses the dummy cache and the other uses the perfect one. The remaining execution does not have static predicates and is constructed in this way: each predicate of type SD is replaced with one of type SE_{i+1}, with no constraints and unlimited window, and then events of type SE_{i+1} are generated in the same number as the average length of the database result set (10 in this case). In this way the behavior of the system is closely simulated. The test is run with an average predicate window of 6 seconds, 1 million of table rows (where used) and a frequency of events that vary from 600 to 4000 events per second.

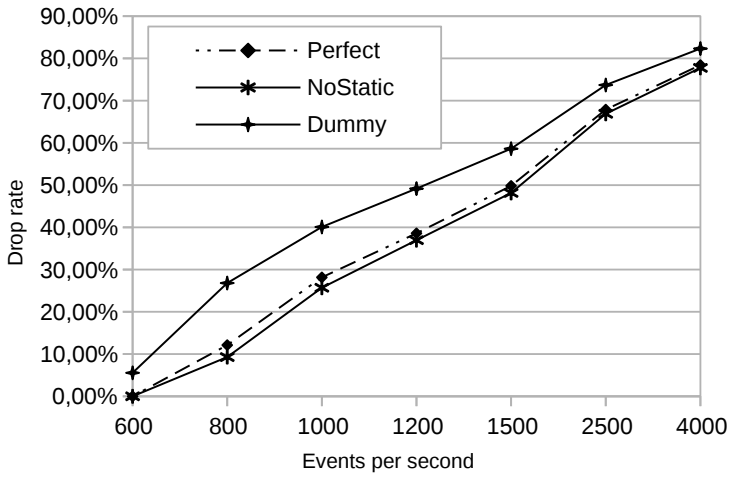

Figure 5.6: Perfect cache - Frequency and Drop

The results in figure 5.6 shows that that T-Rex2 integrated with SQLite in combination with a perfect cache (dashed line in the middle) can almost keep up with an execution that does not require access to the database (lowest line), loosing just around 2% of events more than the competitor. At the same time the execution with the dummy cache draw a lower limit for the worst case performance: so, even in presence of non cache-friendly data or using less perform-

ing caches, the system is still capable of handling a remarkable load, likely enough to satisfy the requirements of many real applications.

5.4.6 *Cache algorithms*

Now that we have defined some term of comparison and showed the results of the best and worst cache, in the following paragraphs we are going to analyze the performance of all the cache algorithms presented in section 4.6 to see how they compare one with the others.

For the execution we set a frequency of 2000 event per second and 2 seconds window, with the event payload sampled from a normal distribution $\mathcal{N}(\mu = 0,\ \sigma = 30)$ and making the static query dependent from the last two predicates (expanding the query parameterization space, making harder to cache enough values). Then we vary the cache size, measured as the number of the rows contained in each cache object, from 500 to 6000.

We first analyze the miss rate of each algorithm, see figure 5.7, and then compare it to the corresponding drop rate to evaluate the impact on the global execution, see figure 5.8.

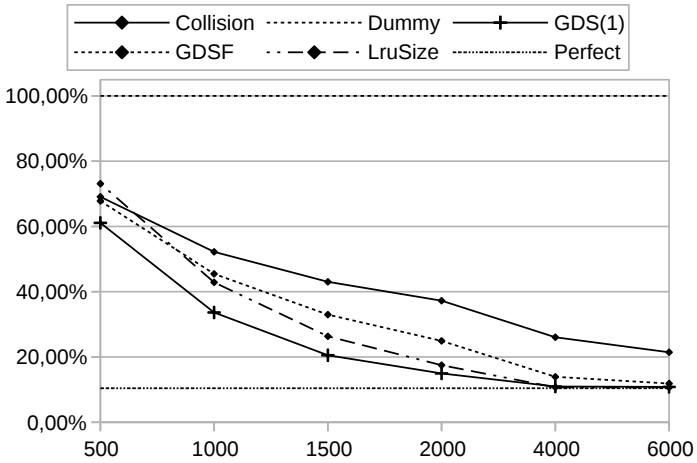

Figure 5.7: Cache comparison - Frequency and Miss

The dummy cache once again shows the worst case performances with 100% miss rate. While the perfect cache sets the upper limit for the maximum efficiency achievable, with a 10% of miss rate. They are shown on the charts as a horizontal line respectively at the top and at the bottom of the plots. At the end of the execution the perfect cache contained about 12000 entries, that is the size of the key space.

The collision cache, that is built on a very basic replacement policy, had a poor performance and was exceeded by every other algorithm. However it was unexpected to see the GDSF cache, that is the de facto standard and champion in HTTP caching, to be clearly left behind in the comparison. My intuition is that its measure of frequency does not work well with the repeated access patterns of a CEP engine

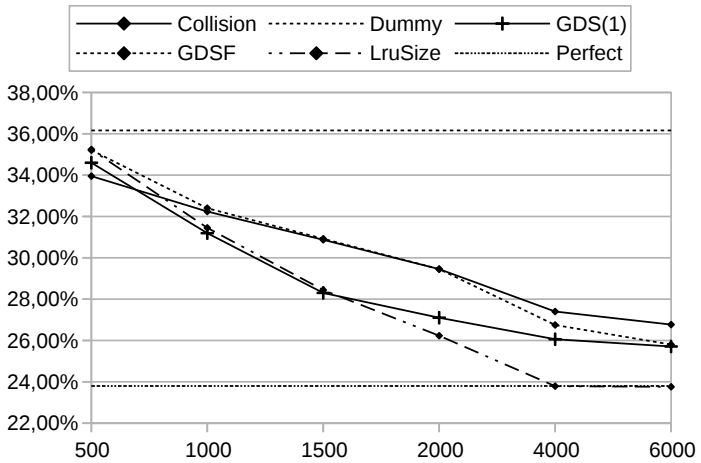

Figure 5.8: Cache comparison - Frequency and Drop

would require a more precise tuning.
The second best is the LRU-size, that proves itself to be a good all-round solution and the first choice for any caching purpose. The winner of the comparison is the GDS(1) cache, that choosing smaller entries is able to maximize the memory usage. We can also notice, that both the top two algorithms reach the level of optimality of the perfect cache with a size between 2000 and 4000 entries, that means 3 to 6 time less than the actual key space.

Comparing the results with the plot of system wide performances, we show that the sole minimization of the miss rate it is not sufficient affect the global execution. The most notable example is how GDS(1), champion of the previous category, fails to keep up with LRU. The phenomenon is explained by the difference in term of hit time: in fact GDS(1) does have more hits (between 5% and 10% more), but with a cost of almost the double of the LRU ($\sim 1200vs$ compared to $\sim 650vs$ of access time). The reason behind this slow down is likely due to the suboptimal GDS(1) implementation.

5.4.7 *Shared or per predicate*

When the cache is allocated as a single storage accessed by different predicates, if they request similar data they will share the space and the cache loading cost. Also if there is some process that has more benefits from caching, an advanced algorithm could reward it with more space in memory. However there is always the risk that a single component would hog the entire space, filling it with garbage and degrading the system wide performance.
A cache per predicate, instead, would be much smaller, but it could better adapt to the specific pattern or distribution of the data.
However we found out to be another the most relevant factor decision: sharing the cache across multiple threads require constant syn-

chronization. In the implementation the cache is wrapped with a mutex that soon became the bottleneck of the entire parallelization.

5.4.8 *Data distribution*

First of all we need to clarify that having a discrete and not too dense domain of values is a fundamental requirement, otherwise even in the smallest range there would be to many elements to hope for a reasonable match in the cache. So for example floating point numbers usually not adapt to take part to the cache key.
The effectiveness of a cache is based on assumptions on the processed data, for example time locality, meaning that a value requested recently is likely to be requested again. Each assumption may fit better or worse depending from the actual data distribution.

The system was built to offer a choice about the distribution of the input event values. So it is possible to experimented with gaussian, exponential and uniform distributions tuning the parameters to spread or narrow the range of the samples.
However we found out that this is only partially relevant, in fact there is a limit to the variability that can be introduced by a single event: because of the nature of the system, only a restricted number of events is queued in a processor and that is the maximum expression of the distribution. Conversely sequences of evaluation are executed repeatedly and the same data is requested many time during a single system iteration, so the benefit of the cache prevail over any possible distribution.
The situation changes when we make the static predicate dependent from more than one event at a time: in this case the probability distributions combine with each other and generate a wider and more diverse output, challenging the capacity of the cache.

CONCLUSIONS

In this dissertation we presented an extension to the TESLA language to describe the interaction between static data collections and event streams, showing that is possible to have a powerful abstraction without compromising the simple and declarative nature of TESLA language or its formal definition.

We showed how the T-Rex engine, operating in connection with an SQLite database, can handle queries over tables of millions of rows within the strict latency requirements of a real-time system and how, in presence of discrete and recurring query parameters, the use of a cache can make the engine perform almost as fast as if it was handling a similar load of pure events. In particular we demonstrated that the component developed outperforms the workarounds that were previously needed to emulate the access to persistent information.

However we verified that the response time of the database, even with the use of caches, has a critical impact on the entire system and the problem is emphasized by the guaranties of time order execution enforced in T-Rex with a blocking architecture. This has been found to be a risky point of failure, because a delay of even a single call to the database may cause the engine to stall. So for future development we suggest the investigation of a fine grained model of concurrency and the possibility to configure the desired level of guaranties.

We also observed that the TESLA language allows multiple formulation of equivalent rules, which may produce different execution plans. At the moment the responsibility of choosing the most efficient alternative is left to the programmer, but this task could be automated introducing a step of rule rewriting.

In conclusion we think that the integration of SQLite into the T-Rex engine was a successful example of the proposed interoperabiliy between CEP tools and DBMSs. So we believe that the presented implementation is worth expanding with new adapters for other data sources. However, as for now, the introduction of so many changes to the project will require a period of stabilization and reintegration of all the existing modules, like the parser and the gpu processor.

BIBLIOGRAPHY

[1] Brian Babcock et al. "Models and issues in data stream systems". In: *Proceedings of the twenty-first ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*. ACM. 2002, pp. 1–16.

[2] Pei Cao and Sandy Irani. "Cost-Aware WWW Proxy Caching Algorithms." In: *Usenix symposium on internet technologies and systems*. Vol. 12. 97. 1997, pp. 193–206.

[3] Ludmila Cherkasova. *Improving WWW proxies performance with greedy-dual-size-frequency caching policy*. Hewlett-Packard Laboratories, 1998.

[4] Gianpaolo Cugola and Alessandro Margara. "Complex event processing with T-REX". In: *Journal of Systems and Software* 85.8 (2012), pp. 1709–1728.

[5] Gianpaolo Cugola and Alessandro Margara. "Low latency complex event processing on parallel hardware". In: *Journal of Parallel and Distributed Computing* 72.2 (2012), pp. 205–218.

[6] Gianpaolo Cugola and Alessandro Margara. "Processing flows of information: From data stream to complex event processing". In: *ACM Computing Surveys (CSUR)* 44.3 (2012), p. 15.

[7] Gianpaolo Cugola and Alessandro Margara. "TESLA: a formally defined event specification language". In: *Proceedings of the Fourth ACM International Conference on Distributed Event-Based Systems*. ACM. 2010, pp. 50–61.

[8] Jeffrey Dean and Sanjay Ghemawat. "MapReduce: simplified data processing on large clusters". In: *Communications of the ACM* 51.1 (2008), pp. 107–113.

[9] Carlo Ghezzi, Dino Mandrioli, and Angelo Morzenti. "TRIO: A logic language for executable specifications of real-time systems". In: *Journal of Systems and software* 12.2 (1990), pp. 107–123.

[10] David Luckham. *The power of events*. Vol. 204. Addison-Wesley Reading, 2002.