Towards Temporal Correctness of Event Processing

Elior Malul

Technion - Computer Science Department - M.Sc. Thesis MSC-2013-24 - 2013

Towards Temporal Correctness of Event Processing

Research Thesis

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science

Elior Malul

Submitted to the Senate of
the Technion — Israel Institute of Technology
Tevet 5773 Haifa December 2012

The research thesis was done under the supervision of Prof. Josef Gil and Dr. Opher Etzion in the Computer Science Department.

The generous financial support of the Technion is gratefully acknowledged.

Contents

Abstract **1**

Abbreviations and Notations **3**

1 Introduction **4**

2 Related Work **7**

3 Motivating scenarios **10**
3.1 Terminology 10
3.2 The fair auction scenario 12
3.3 The adaptive toll scenario 14

4 Correctness Guards **17**
4.1 Definitions 17
4.2 Fairness guard 19
4.3 Nested derivation guard 20
4.4 Consecutive derivation guard 21
4.5 Discussion 22

5 Language support **24**
5.1 Declaration of correctness guards 24
5.2 Extensions to event processing execution languages 26
5.3 Deployment of correctness guards 27

6 Implementation algorithm **31**
6.1 Variation on the linear two-phase commit algorithm 31
6.2 An alternative approach 35

6.3 Performance comparison 35
6.4 Summary 38

7 Temporal assertion language (Tal) 39
7.1 Introduction and motivation 39
7.2 The expressive power of TAL 41
7.3 Grammar 42
7.4 Examples 43
7.5 TAL code generation 44
7.6 Summary 47

8 Summary 49

List of Figures

2.1 Types of temporal windows 7

3.1 The fair auction scenario 13
3.2 EPN for the adaptive toll scenario 15

5.1 Graphic tool for inserting a nested derivation correctness guard into an EPN 25

6.1 The buffering and the two-phase commit algorithms compared to a naïve implementation by their throughput 37
6.2 The buffering and the two-phase commit algorithms compared to a naïve implementation by their latency 37

7.1 EPN with V-shape topology, to which two paths have different derivation policies 40
7.2 The mutual exclusion problem, a correctness issue that cannot be solved by TAL 41
7.3 The abstract context free grammar of TAL, a declarative temporal assertion language. 43

Abstract

Although event processing is considered an emerging technology, empirical studies show that most enterprises ability to use such systems as ranging from poor to fair. Moreover, many of them indicated the skill issue as a barrier to adoption. One of the major contributors to this observation is the inherent difficulty in enforcing temporal correctness using current languages and tools, as also addressed by [18, 7, 22]. This difficulty can lead to incorrect outcomes or to work being hacked around these issues for a particular event processing system. Such workarounds create a substantial entry barrier in the development of event processing applications.

We propose a solution that uses high level programming constructs, (which much like query languages such as SPADE [15], abstracts away the excruciating details of the middleware), for defining and enforcing the requirements of what we call *temporal correctness guards*. We demonstrate this approach by defining three temporal correctness guards that correspond to frequently used correctness requirements. One of the three guards we define is the *fairness guard*, which guarantees that events are processed in their logical order, regardless to the path in which they were processed in. This is vital for applications in which fairness must be kept with respect to the order; examples of such application include bids, stock purchases, and computerized games. We present two scenarios that demonstrate these correctness guards and formally define each guard. Since the correctness guards are model-independent, we also define the extension needed for a specific event processing model to support these guards. The scenarios include the run-time algorithm we use to enforce the guards within that model. The proposed solution opens up the opportunity to develop correctness abstractions, which have the potential to significantly reduce the skill barrier for developing event processing applications. The main contribution of this

thesis, is that it pinpoints potential correctness problems in event processing application, that we think to be common. Furthermore, this thesis formalize these problems, as well as offer a solution to deal with those problems. We then continue by generalizing the solutions into runtime artifacts to enforce the guards. Finally, we define a domain specific language, that allows one to define new guards.

Abbreviations and Notations

EPA	—	Event processing agent
EPN	—	Event processing network
Lat.	—	Latency
msec	—	Milli-second
Through.	—	Throughput

Chapter 1

Introduction

Despite the fact that event processing is considered an emerging technology, empirical studies done by analysts [33] indicate that most enterprises view the level of their skills and abilities to use such systems as ranging from poor to fair. Many of them indicated the skill issue as barrier to adoption. For example, one barrier is the difficulty inherent in creating applications that are consistent with the requirements and intuitions of the users. Because many of these requirements relate to the notion of time [18, 7, 22], it is especially difficult to take them into consideration.

The processing itself is done within the physical time of the system, yet events may come with their own logical occurrence time. This problem is manifested in the question of whether the output of an event processing application (derived event) is consistent with the intentions of the users and the way they understood the requirements of the application. These difficulties are quite common in application development and can either lead to results inconsistent with the intuition of the application's users, or to the need to find "workarounds" by hacking into a particular system's logic or by creating code that bypasses the system. As an example, in Section 3, Listing 1, we describe a concrete trace of a simple scenario that was naively implemented using one of the event processing products. The scenario deals with fairness in bids and clearly produces results that are intuitively incorrect. The fact that workaround is required to obtain correct results raises the bar of the skills required from the developers, and harms the usability gains achieved from the use of higher level languages in this area.

Drilling down to the root cause of this problem reveals that it stems from the highly temporal nature of event processing. This temporal nature is realized in two ways:

1. Temporal event processing patterns where the order of events affect results, such as detecting a sequence of events or detecting trends on the progress of events with time. In these cases, the timing in which events logically occurred from the users' view is significant for obtaining correct results consistent with users' intuition.

2. Events are processed within time windows of various types (temporal contexts). The inclusion or exclusion of an event from a certain window may change the results. But due to race conditions between the time that events are processed and the time in which a particular window is opened or closed, the results may again turn out to be non-intuitive to the users of the application.

In many of the current systems, temporal race conditions are created that might generate incorrect results. Moreover, the developer has to identify that such a problem occurred and deal with it in a manual, sometimes tricky way.

The issue of temporal correctness in event processing systems has been addressed before, both by [18] and by [22]. The prior solves the temporal correctness problem by using the *buffering technique* also used by [1]. The former offers three strategies of dealing with temporal correctness, all different from our own. The first two are also derivatives of buffering technique, where as the third involves a compensation mechanism for incorrect results. The pros and cons of these strategies in comparison to our own, is further discussed in Section 2.

This thesis focuses on the **temporal correctness of event processing**, in an effort to contribute to this problem. Our aim is to give system designers the means to define explicit temporal correctness guards that are automatically compiled into constructs in the lower level language. These correctness guards would be enforced at run-time and eliminate the temporal fallacies.

The rest of this thesis addresses the different aspects of this study. Chapter 2 starts with related work. In Section 3 we demonstrate the problem,

using two motivating scenarios. In one of them, we show that naïve implementation in current languages yields incorrect results. We demonstrate our solution approach in Chapter 4 by defining an initial set of requirements that are typical to many applications and can be demonstrated through the scenarios we selected. In Chapter 5, we describe some extensions required to the current event processing languages to support such requirements, based on the event processing model described in [14]. We also describe the automatic compilation from the definition of the correctness guards to the execution language. In Chapter 6, we introduce a run-time algorithm that can be used to support these correctness guards for that model. Chapter 7 describes TAL, a domain-specific language, that aims to supply the means to define and deploy yet new correctness guards. Finally, in Chapter 8, we present our conclusions and briefly address other facets of correctness issues, such as the extensions required to support more correctness guards.

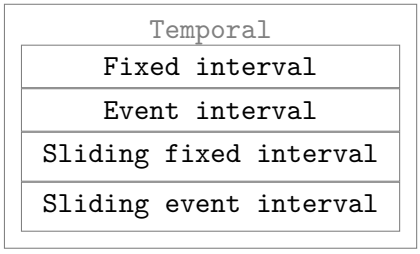
The main contribution of this thesis is to the correctness of event processing applications. First we identify and formalize some correctness issues we identified, in event processing application. We then offer solutions to those problems, which are later generalized into a construct named correctness guard. There are more correctness issues out there, ones that are not covered by this thesis, some of them could be addressed correctness guards. For that end we devised a domain specific language named TAL, that enables one to develop new correctness guards.

Chapter 2

Related Work

The notion of temporal context and various kinds of time windows is described by Etzion et al. [13] as shown in Figure 2.1. Time windows may

Figure 2.1 Types of temporal windows



be fixed based on the occurrence of events, which start and end the time windows. Sliding windows may be based on time (overlapping or non-overlapping) or based on the number of event instances. While there are many varieties on the semantic level, there is no work on ensuring that all events that are semantically classified to be inside a time window are actually treated as such by the run-time engine. The notion of punctuation in stream processing [30] enriches event streams with events that denote the end of a sub-stream. While the semantics are similar to that of the terminator in the event interval [13], this work assumes that streams are totally ordered and that events are processed in order. However, these assumptions may not hold in most event processing applications. Part of the problem is the case where events that are emitted by various external producers arrive out of order. A summary of approaches for out-of-order handling of input

events is summarized in [22, 35, 12]. Additional discussions about the order issue appear in specific areas such as [20] on sensor network data. There are practical challenges in maintaining the order of events coming from external sources. However, even if we assume that the correct order of events flowing into the system is preserved, it does not guarantee that the run-time engine processes them and their descendants in the correct logical order, as we demonstrate in Section 3.

Our solution approach based on guards is inspired by the notion of guarded commands introduced by Dijkstra [10]. High-level definitions of correctness criteria and their automated enforcement were also introduced within several domains, such as the BPM domain [31], the product line software [19], and in transactional memory systems [27]. The idea of abstracting away timing constraints was used in the synchronous dataflow paradigm (languages like Esterel [5]). It can also be found in other approaches to synchronous computations over time-varying data such as simulation languages, (e.g., Simulink [25]), and in functional reactive programming (FRP) [34]. The main difference lies in the level of programmability. In our approach, temporal consistency is programmable and not fixed within the system, since the notion of temporal correctness may vary among different applications. For example, different applications may view events that occur simultaneously with the time window termination as being inside or outside the scope of this window. Three solutions for that were offered by Claypool et al. [22]. The first is the traditional buffering technique (referred by them as K-slack). Although it is simple to implement and intuitive, the buffering technique requires one to commit to a certain threshold time $\bar{t}$, which is the maximum delay for an event. Identifying such a threshold may be tricky, since if is made too short, there is a risks incorrect results. On the other hand, if it is made too long ,it will cause latency delays. The second solution offered was a modification for the buffering technique. In this solution, $\bar{t}$ is set to ∞. In order to avoid the indefinite accumulation of states, a partial order guarantee (POG) is embedded in each event. A POG data comes in the form of an ordered pair $\langle E, t\rangle$, which guarantees that there will be no event of type E with timestamp less then t. Thus, when $\langle E_0, t_0\rangle$ is detected by the system, all the events of type E_0 with timestamp earlier then t_0 may be purged.

The downside of that solution is that the generation of the POG re-

quires the system designer to know much about the nature of the data of its application, rather then just restricting his knowledge to the logic of it. Moreover, not always such POGs can be generated. The third offered solution is called the "aggressive strategy". In the aggressive strategy, events are being derived based on the current knowledge. If in some point in the future, that knowledge turns out to be incorrect, a "compensation" event is derived, to notify system agent that previous events where falsely generated.

The upside of this approach is that latency times are reduced to minimum, however, the downside is that sometimes, compensation is not an option, (e.g., it would not be acceptable to generate an event that launches a rocket).

Although this work focuses on temporal correctness, there are other semantic difficulties within the state of the practice such as ensuring deterministic behavior [3] and the performance implications of application complexity [24]. These issues are beyond the scope of this paper.

Data guards has also been used in the domain of system-correctness, both of traditional RDBMS systems [28], and that of event processing systems [17]. The main difference between these papers and this thesis, is that while they are trying to guarantee result correctness in the presence node-failures. The correctness problem we face are not only subjected to system failures, they occur also when all the nodes comprising the system are fully functional.

Chapter 3

Motivating scenarios

In this section, we provide two scenarios to demonstrate the correctness requirements and the problems that exist in current systems. We start by defining the terminology used in our work and then detail two scenarios: the fair auction scenario and the adaptive toll scenario.

3.1 Terminology

In this section, we briefly describe the terminology used throughout this paper. Our work is based on the concept of the *event processing network* (*EPN*) that was introduced by Luckham et al. [23], and refined by Etzion et al. [14]; Figures 3.1 and 3.2 describe scenarios that use the notation of EPN. An EPN consists of a collection of *event processing agents* (*EPAs*) and events that flow between the EPAs. A collection of events of the same type that flows to or from an EPA is known as an *event stream*. An EPA instance receives one or more events as input and derives one or more events as output. We refer to events that are emitted by external producers as *raw events* and to events that are generated by an EPA as *derived events*. Throughout the paper, we use the following EPA categories: *filter*, in which events are filtered in and out based on user-defined Boolean statements; *enrich*, in which the content of a given event is enriched by a database or other external store; *aggregator*, in which one event is derived as an aggregation of a collection of events; *composer*, which composes events from

different types[1]; and *pattern matcher*, which indicates whether a *pattern* (e.g., sequence, conjunction, absence, and trend) within a collection of events has occurred. All the events flowing through the system are typed, such that each event is an instance of a single event type. An event has two temporal properties [12]: *detection time*, which denotes the time in which the event reached an entry point of the processing system; and *occurrence time*, which denotes the real time at which an event occurs, as reported by the event source.

Each EPA has instances; each instance is responsible for processing a certain collections of events. The specification of such a collection is known as *context* [13]. A context may have several dimensions. We focus on two dimensions: *segmentation context* and *temporal context*. Segmentation context divides events among EPA instances by a given criteria. For example, the events related to specific customer are processed within a distinct EPA instance. Temporal context, also known as a *temporal window* (or just *window*), can either be a single window or a sliding window. A single window starts either with the occurrence of a certain event or at a predefined point in time. It ends when an event occurs or within a time offset from the start. A sliding window consists of a collection of windows with fixed length or with a fixed number of events of certain types. Windows may or may not overlap. Since we assume that event occurs in a specific point in time, then for each window, an event either occurs within the window or outside the window boundaries. A context often comprises these two dimensions—partition by a certain segment, (e.g., a customer ID), as well as a certain time window (e.g., a non-overlapping sliding window where the duration of each window instance is one hour). Several types of policies also tune the semantics of the various event processing functions. In this paper, we refer to two of these policies: the *evaluation policy*[2] and the *order policy*. The evaluation policy may be either *immediate*, in which computation is done whenever a match is detected, or *deferred*, in which computation is done at the end of the window. The order policy determines whether the order of events for processing purposes is driven by the *detection time* value or the *occurrence time* value of the events. More details about the terminology and

[1]Composer is similar to "stream join" in the stream models terminology, *e.g.,* [16].

[2]The evaluation policies terminology is taken from coupling modes in active databases [26]

term definitions may be found in [14]. Next, we present the two scenarios.

3.2 The fair auction scenario[3]

A computerized auction system (Figure 3.1) enables people to bid on items and guarantees fairness in the bids. A specific bid starts with a `bid start` event and ends either by an explicit `bid end` event or, if one was not emitted, by a predetermined time offset (e.g., ten hours). An auction management site is also a credit-granting institute that provides credit to customers. The bidder can rely on such credit while bidding, but naturally this entails an approval process before the actual bid is issued. Thus, two types of bid events exist: `cash bid` (E_1) and `credit bid` (E_2). Events of type `credit bid` are enriched by A_1, to determine the credit status, and then filtered by A_2 to determine whether the credit has been authorized. An `enriched credit bid` (E_3) is derived during the enrichment process. The filtered-in event is an instance of the `confirmed credit bid` (E_4) event type, which has the same structure of E_3. The *determine winner* EPA (A_4) gets all the bids in the two paths and determines the winner.

The fairness ground rules that determine the winner are:

1. All bidders that issued a bid within the validity interval participate in the bid.

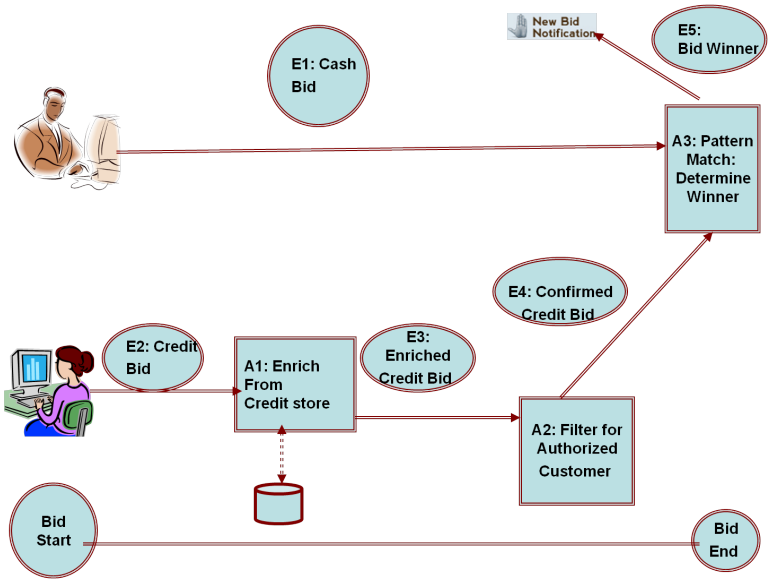
2. The highest bid wins. In the case of a tie between bids, the *first* accepted bid wins the auction.

To illustrate the difficulty of implementing these ground rules in current systems, we experimented with some of the existing systems. In Listing 1, we show some results obtained by using one of the event processing products. (We don't identity the product since this behavior is typical for many products using naïve implementation.) As shown, if we apply our ground rules over the input, the winner should be bidder 56, who issued the highest bid (5). Among those who issued the second price (4), the first one is bidder 2, and the second is 29. However, looking at the results, we see that bidder 29 has won.

The result exposes two problems:

[3]This scenario was inspired by an example discussed in the book Event Processing in Action [14], Chapter 11, but has been slightly modified.

Figure 3.1 The fair auction scenario



1. Bidder 2 issued a credit bid a short time before bidder 29 issued a cash bid. Since the credit bid had an approval path before getting into the determine winner EPA, it appeared to the EPA as if it occurred later.

2. Bidder 56 issued a credit bid a short time before the bid interval ended. However, an internal race condition caused the bid end event to occur before this event was processed, thus it did not arrive in time to participate in the winner calculation for this auction.

We made two significant observations during this exercise:

1. The implementation was done in a naïve way, by simply implementing this scenario in the most intuitive way available in the system we implemented. These issues have workarounds, but they are not simple and require in depth familiarity with the way a particular system works.

2. This scenario is relatively simple since both A_1 and A_2 are stateless EPAs with a single input event and a single output event.

In Section 3.3, we address an additional scenario with other common consistency requirements.

Listing 1 Trace of the bid example, implemented in existing event processing language, demonstrating faulty output due to temporal race condition.

```
1 ===Input Bids===
  Bid Start 12:55:00
3 credit bid id=2,occurrence time=12:55:32,price=4
  cash bid id=29,occurrence time=12:55:33,price=4
5 cash bid id=33,occurrence time=12:55:34,price=3
  credit bid id=66,occurrence time=12:55:36,price=4
7 cash bid id=56,occurrence time=12:55:59,price=5
  Bid End 12:56:00
9
  ===Winning Bid===
11 cash bid id=29,occurrence time=12:55:33,price=4
```

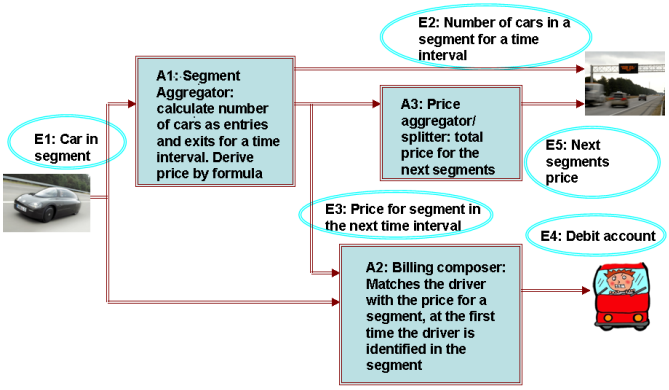
3.3 The adaptive toll scenario[4]

Some toll roads use billing systems that employ adaptive tolls according to traffic density; the cost of driving a certain road segment may vary. Such a system works by partitioning a road into segments. Each segment spans between two highway exits, such that a driver may decide to continue on to the next segment or exit the highway. At the beginning of each segment a sensor identifies the car. The price is re-evaluated every ten minutes and the applicable price is the price that was set at the end of the previous ten-minute interval, before the driver entered a certain segment. Electronic boards are exhibited in each segment and on all routes leading to the highway to display highway statistics such as traffic load per segment, prices of upcoming segments, etc. The billing formula is a function of the number of cars within a ten-minute interval, which is calculated at the end of the interval and applies to the next ten-minute interval. A billing invoice is issued separately for each segment. The above scenario is illustrated in Figure 3.2.

Events of types `car entered` (E_1), designate that a car with a certain ID has entered a certain road segment. We assume that the road consists of a collection of road segments, $\langle s_1, \ldots, s_n \rangle$, and that the temporal dimension is a non-overlapping sliding window with a duration of ten minutes, whose

[4]This scenario was inspired by the "linear road benchmark" scenario introduced in Richard Tibbetts' MSc Thesis [2]; it was significantly modified to fit the points we wish to highlight.

Figure 3.2 EPN for the adaptive toll scenario



individual windows are designated as $\langle w_1, \ldots, w_n \rangle$. The EPA *segment aggregator* (A_1) operates within a composite context of a segmentation context, which partitions the event space by road segments. For example, s_i and a window member within a sliding temporal window of ten minutes duration such as w_j. This EPA receives `car entered` events as input whenever a car is detected as entering a segment and serves as an aggregator by creating two derived events: `car count` (E_2), and `segment price` (E_3). Events of type `car count` report the number of cars that entered the segment s_i in some window w_j. Events of type `segment price`, (which are calculated at the end of w_j), report the price of driving through segment s_i, in the time interval represented by w_{j+1}.

The EPA *billing composer* (A_2) also operates in the context of a segment and that of a window. A_2 derives events of type `debit account` (E_4), immediately upon detection of car entered segment events. The `segment price` event used for this composition is the one that refers to the previous window (w_{j-1}) for the same segment s_i.

The EPA *price aggregator* (A_3) takes all `segment price` (E_3) events of all road segments as an input and creates a derived `next segments prices` (E_5) event for each segment $s_i \in \{s_1, \ldots, s_n\}$. These events hold the prices for all segments between s_i and s_n (the end of the road), during the time interval represented by w_j. The entire array is created once and then the appropriate sub-array is broadcast to all electronic boards preceding segment S_i. Implementing this scenario also has some pitfalls that stem from race

conditions between events and window boundaries. In a typical case, an E_3 event (`segment price`) is derived from the agent A_1, and serves as input to the agent A_3, which are both defined in contexts that have the same temporal dimension (non-overlapping sliding window of ten minutes). An E_3 event is derived at the end of the window, but it is too late to be included in the agent instance A_3, which operates within that window. Thus, the agent A_3 gets incorrect results, since some of its input moves to the next window. Now that we have provided some insight into the types of correctness problems, we move on to discuss correctness guards in a systematic way.

Chapter 4

Correctness Guards

In this section we define three correctness guards that are common in the applications we observed and can be demonstrated by the two scenarios presented in Section 3. There is no single correctness guard that can heal all illnesses, but rather a set of guards from which a system designer can choose to apply for a specific application. Using these guards will ease the task of creating temporally correct applications.

In Section 4.1 we define constructs that are later used to formally define three correctness guards: the *fairness guard* in Section 4.2, the *nested derivation guard* in Section 4.3, and the *consecutive derivation guard* in Section 4.4. We then complete the section with a short discussion in Section 4.5.

4.1 Definitions

$e \in In(\alpha)$ denotes that the event e is the input of the EPA instance α. Similarly, $e \in Out(\alpha)$ denotes that the event e was produced by the EPA instance α.

We say that the derivation of event e' was *triggered* by another event e, if the following properites apply to e and e':

1. If e wouldn't have been detected, e' wouldn't have been generated.

2. e is the maximal event (with respect to its timestamp), among all the events for wich 1 applies.

Definition I: an event e is said to be the *deriver* of event e' with respect to an EPA instance α, (and e' to be the derivative of e), denoted by $e \triangleright_\alpha e'$ if the following applies:

- $e \in In(\alpha)$,
- $e' \in Out(\alpha)$,
- e is the event that *triggered* the creation of e' by α.

In the application shown in Figure 3.2, the evaluation policy of A_1 is deferred. Hence, context terminators derive the events of type E_3 produced by A_1. If the evaluation policy of A_1 was immediate, every event of type E_1 would have been the deriver of an event of type E_3. $\triangleright$ is therefore a family of injective functions whose domain and range is of type Event. Each EPA instance has its own derivation function; we refer to the derivation function of a given EPA instance α, by sub-indexing the derivation sign: $\triangleright_\alpha$

Definition II: Let $\Pi = \langle A_1, \ldots, A_n \rangle$ be an ordered set of distinct EPAs, all part of some EPN. The order between the EPAs is set by a topological order induced by the containing EPN. We say that $\pi = \langle \alpha_1, \ldots, \alpha_n \rangle$ is a *dependency path* of type Π, if $\alpha_1, \ldots, \alpha_n$ are EPA instances of type $A_1, \ldots, A_n$ respectively.

Definition III: An event e' is said to be the *indirect derivative* of another event e, (and e is said to be *indirect deriver* of e') with respect to the dependency path $\pi = \langle \alpha_1, \ldots, \alpha_n \rangle$, if there is a set of events $\langle e_1, \ldots, e_{n-1} \rangle$ for which e_1 is the derivative of e with respect to α_1, which in turn derives e_2 with respect to α_2 etc., until e' is derived by e_{n-1} with respect to α_n.

More formally, $e \triangleright *_{\langle \alpha_1, \ldots, \alpha_n \rangle} e'$, if $e \triangleright_{\alpha_n} \circ \triangleright_{\alpha_{n-1}} \circ \ldots \circ \triangleright_{\alpha_1} e'$ applies. For example, each context terminator event for EPA A_1 (Figure 3.2) has an event of type `debit account` (E_4) that indirectly derives it along a dependency path of type $\langle A_1, A_2 \rangle$.

Definition IV: Let α be an EPA instance and c be a context instance of type C. An event e is *associated to c by* α, (denoted by $e \in_\alpha c$), if α processes e within the context of c.

In the example shown by Figure 3.2, Let us assume the existence of a temporal context w, which spans between 10:00 am and 10:10 am. We shall denote the fact that `car in segment` event (e) is processed by a_1, (an EPA instance of A_1), in the context of w by $e \in_{a_1} w$. Next, we define the three correctness guards: the *fairness guard*, demonstrated by the fair auction scenario, the *consecutive window guard* demonstrated by the adaptive toll scenario, and the *nested derivation guard* demonstrated by both scenarios in different fashions.

4.2 Fairness guard

The fairness guard aims to provide the means for system developers to ensure a specific order by which input events are processed by an EPA instance. This is useful for EPAs that are order sensitive (i.e., the correctness of their output depends on the order of their inputs).

Example: In the fair auction scenario (Section 3.1), the guard requires that if a `credit bid` event has been emitted before a `cash bid` event, then its derived `enriched credit bid` event should win in the case of a tie for the highest bid. Assume that a bid ends with a `bid end` event at 11:00. A `cash bid` event for customer C_1 is issued at 10:45:01 for the amount of 15,000$, and a `credit bid` event for customer C_2 is issued at 10:44:57, also for the amount of 15,000$. The corresponding `enriched credit bid` event is created at 10:45:03, and the filter EPA (A_2) approves it and creates a `confirmed credit bid` event at 10:45:10. At 11:00 (the bid end), it is determined that 15,000$ was the highest bid amount with two bidders: C_1 and C_2. The system should chose bidder C_2 to be the winner, although its direct input to the determine winner EPA was created *after* the event related to C_1.

Formal definition: Let π and ψ be dependency paths of types Π and Ψ, respectively. Let E' and F' be the out event types of the last EPAs of Π and Ψ, respectively. And, let E and F be the input types of the first EPAs of Π and Ψ, respectively. In the formal fairness assertion shown in Guard 1, e' and f' are events of types E' and F', while e and f are events of type E and F. e' is the indirect derivative of e with respect to the dependency path π and f' is an indirect derivative of f with respect to the dependency path ψ.

We use the order relation $\ll$ between event instances. The interpretation

Guard 1 $Fairness(\Pi, \Psi)$ which asserts order preserving between two given dependency paths.

$$e \triangleright *_{\pi} e' \wedge f \triangleright *_{\psi} f' \wedge e \ll f \Rightarrow e' \ll f'$$

of this order relation is application specific (i.e., $e \ll f$ either means e is *detected* before f or that e *occurred* before f). The guard definition uses the second interpretation.

When the fairness guard shown above is deployed for an EPN, the run-time ensures that the right side of the assertion will hold every time the left side holds.

4.3 Nested derivation guard

The next two guards handle the cases in which one or more EPAs are context sensitive, so their behavior is influenced by the context under which they operate. Attaching this guard to an EPA ensures that its derived events will be processed within the same temporal context (or a co-ending temporal context) as of its derivers. This requirement is used to support cases in which events are derived at the end of a temporal window and should be processed in the context of that window by some EPA.

Example: in Figure 3.2, E_2 events, which are calculated for each road-segment at the end of each temporal window, sum the total number of cars that entered a certain segment, in a certain time-frame. Let us assume the event (e) that was calculated at the end of window w =(10:00 – 10:10). e will most likely be detected by A_2 *after* w ends (say at 10:10:02). Nevertheless, A_2 must calculate a `debit account` event (p) within the context of w since that is the time-frame in which the car entered the road segment. To support such logic, we suggest the *nested derivation guard*.

Formal definition: we define two auxiliary functions; $start : C \rightarrow T$ and $end : C \rightarrow T$. These associate the beginning and ending timestamps of a given temporal context (window), where T is an unbounded set of

time constants. Guard 2 shows a correctness assertion in which α_1 and α_2 are EPA instances of type A_1 and A_2, respectively, with associated context window instances c_1 and c_2, respectively. We demand that α_1 will be directly connected to α_2, so the guard will be applicable. Events e and e' are of type E and E', where $e \in In(\alpha_1)$, $e' \in Out(\alpha_1)$, $e' \in In(\alpha_2)$, and $e \triangleright_{\alpha_1} e'$.

Guard 2 $NestedDerivation(A_1, A_2)$ which asserts nested context association between events derived by α_1 and those derived by α_2.

$$e \in_{\alpha_1} c_1 \wedge e \triangleright_{\alpha_1} e' \wedge end(c_1) = end(c_2) \Rightarrow e' \in_{\alpha_2} c_2$$

Given this assertion, the runtime ensures that for every two events e, e' where e is the direct deriver of e', the two events will be processed in co-ending contexts.

4.4 Consecutive derivation guard

This guard is closely related to nested derivation guard, but it states that a derived event should *not* be processed in the same context as its deriver, rather in its consecutive context.

Example: Let w_i be (10:00–10:10), and p_i the price calculated for some road segment based on the traffic load noticed on w_i. p_i should be used by A_3 (Figure 3.2) in the context of w_i's successor, (w_{i+1}=(10:10–10:20)), as mentioned in the adaptive toll scenario, since cars that entered a certain section within w_{i+1}'s time frame will see the price p_i as the price for the segment. To enforce that logic, the *consecutive derivation guard* can be defined for `price for segment` events emitted by A_2 to A_3.

Formal definition: first, we need to define an auxiliary function that returns a window w', which is the successor of a given window w of the same domain. More formally: $succ : W \rightarrow W$, where $succ(w) = w' \Leftrightarrow w' > w \wedge (\forall w''.\ w'' > w \Rightarrow w'' \geq w')$. The complete assertion is shown by Guard 3, in which A_1 and A_2 are directly connected EPA types. Both EPAs have the same temporal context type C, whose temporal dimension is of type sliding window. c_1 is the context associated to α_1, and c_2 is associated to α_2. α_1

and α_2 are EPA instances of A_1 and A_2, respectively. e and e' are event instances of types E and E', respectively, in which e is the direct deriver of e' with respect to α_1.

Guard 3 $ConsecutiveDerivation(A_1, A_2)$ which asserts that a derived event e' will not be processed in a co-ending context c_2 by α_2.

$$e \in_{\alpha_1} c_1 \wedge e \triangleright_{\alpha_1} e' \wedge end(c_1) = end(c_2) \Rightarrow e' \in_{\alpha_2} succ(c_2)$$

When the above guard is attached to an EPA, the runtime enforces the right-hand side of the assertion any time the left one holds.

4.5 Discussion

The three correctness guards have the potential to solve temporal correctness issues in event processing applications, based on samples of applications we surveyed. As such, they represent concrete and vital requirements as designated by system designers but are not supported in that form by the current generation of event processing systems and products. The common denominator among these requirements is that they are all related to the temporal properties of derived events. These properties are realized by the classification of events to temporal contexts and by the relative ordering of derived events with other events (raw or derived) that are processed together with these events. Enabling system designers to satisfy these correctness guards and have them enforced by the event processing runtime environment without extra programming, will create a quantum leap in the usability of such systems, and a proof of concept to support additional requirements of this sort. Current products have fixed assumptions and do not easily enable setting these correctness guards as application design decisions. We demonstrate the notion of correctness guards through these three guards, but one should be aware to the fact that those guards cannot be applied to solve all the correctness problems in event processing applications. Worst, we don't even know what those are; one such correctness problem we have been able to identify is the mutual exclusion problem (Figure 7.2). The reason we have chose to solve these three problems, is that they are the ones we identified in application we surveyed as crucial.

A discussion about extending additional guards appears in Section 8. In the next section, we show how to extend existing event processing languages with constructs that are vital to supporting these requirements.

Chapter 5

Language support

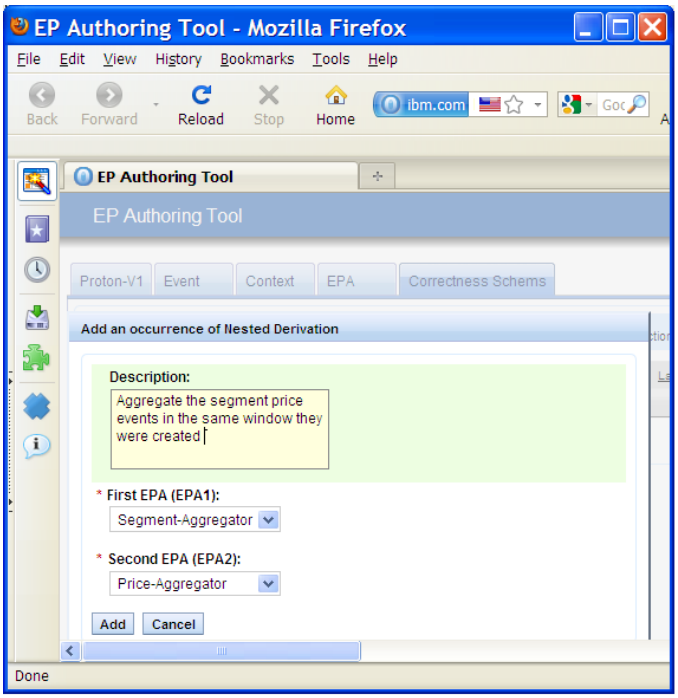
There are two levels of language abstractions that can be supported to specify correctness guards such as those discussed in Section 4. The higher level abstraction is to define the correctness guards as a semantic abstraction that closely reflects the requirements stated in Section 4. We describe these abstractions in Section 5.1. In Section 5.2 we describe an extension to a common event processing execution language and in Section 5.3 we show the mapping between the higher level abstractions to the lower level execution languages.

5.1 Declaration of correctness guards

Each correctness guard is represented by a parametric template, as formally defined in Section 4. We envision a library of correctness guards that can be used by system developers to ensure the correctness of their event processing applications. Those guards are not architecture-specific and thus could be deployed for any event processing language that supports the model presented in [14], as well as the language extensions presented in this section. The screen-shot in Figure 5.1 shows the declaration of the nested derivation guard (Section 4.3) on the EPN of the adaptive toll application (Section 3.3).

In the following section, we show extensions to execution-oriented languages that enable the deployment of the guards discussed in Section 5.1. We then discuss the mapping between the two levels of abstraction.

Figure 5.1 Graphic tool for inserting a nested derivation correctness guard into an EPN



5.2 Extensions to event processing execution languages

In this section, we describe two linguistic extensions to event processing execution languages that are required to map the requirements definitions to existing languages. The two extensions required are a policy-based assignment of *occurrence time* for derived events and a policy-based semantics of *edge points* in temporal context.

The first extension concentrates on setting an *event time-assignment policy*. For raw events, timestamps are typically assigned by the event producer. The occurrence time semantics for derived events does not have a universal interpretation that is applicable in all cases [12]. Thus, we propose a linguistic extension that enables the system designer to specify the timestamp assignment policy for derived event. While various possibilities exist, we propose three policies:

1. *Detection time based policy*—the timestamp assigned to the event is the time in which it was detected by an EPA instance. This policy is the one most commonly used by systems today.

2. *Occurrence time based policy*—the timestamp is copied from the deriver event to the derived event.

3. *End of window*—the timestamp of the event is set to be the timestamp of the context terminator in which the event was processed.

The second linguistic extension enables the system designer to define whether each edge of the temporal context is *open* or *closed* (i.e., whether the temporal context includes events whose occurrence times are equal to the window start or end times). The following example illustrates this: Let w be a time window whose start and end timestamps are T_s and T_e, respectively. Table 5.1 shows the different inclusion options of an event e whose occurrence time is τ, by w.

The type of edge points (e.g., inclusive, exclusive), used by temporal contexts are significant, since in many scenarios the same timestamp is used to open or close more than one temporal window. For example, this could be every ten minutes as shown by the adaptive toll scenario (Section 3.3).

Also, when EPAs derive an event as a result of context termination, the derived event and context terminator have the same timestamp.

5.3 Deployment of correctness guards

In this section, we explain how the two linguistic extensions of Section 5.2 are used to automatically compile a correctness guard into the lower level language. First, we assume that all the event processing agents operate according to the occurrence time order policy (i.e., event ordering is done by comparing their occurrence time). Second, we show how the linguistic extensions can be used to ensure that each given guard holds. Guard map 1 shows how a $fairness(\pi, \psi)$ guard is deployed into the language by assigning *timestamp assignment policies* to derived events along the dependency paths π and ψ, to which the fairness guard should be applied.

Guard map 1 Fairness guard

```
function MAP_FAIRNESS(π, ψ)
    for e : derived event ∈ ψ ∪ π do
        timestamp_policy (e) := occurrence_time
    end for
end function
```

Let us demonstrate how Guard map 1 actually satisfies the fairness guard of Section 4.2 by using the fair auction scenario (Figure 3.1). In that scenario, fairness should be applied between two dependency paths: the "cash-bid path", which starts with the `cash bid` (E_1) event producer and ends with EPA A_3; and the second "credit-bid path", which starts at the `credit bid` (E_2) event producer and also ends with A_3. Guard map 1 ensures that when A_1 assigns a timestamp to events of type `enriched credit bid` (E_3), it assigns the occurrence time of its deriver event of type `credit bid` (E_2). This will also be the case in the derivation of `confirmed credit bid` events (E_4) by A_2. This way, when EPA A_3 needs to apply the tie-breaker described in Section 3.2, it actually compares the occurrence times of a `credit bid` event and that of a `cash bid` event, hence satisfying the fairness guard.

The mapping of the nested derivation guard (Section 4.3) on an EPA A that processes events of type E is shown in Guard map 2.

Consider the example given in Section 4.3 for this guard, in which the

Guard map 2 Nested derivation guard

function MAP_NESTED_DERIVATION(A, E)
  $\forall e : E$, timestamp_policy (e) := end_of_window
  window_start_policy(A) := opened
  window_end_policy(A) := closed
end function

`next segments price` events, which accumulate the prices for all segments in w_j, should accumulate all `segment price` events that were created within the time interval w_j. This is achieved by

1. Setting the occurrence time of `segment price` events to be equal to the w_j termination time, and by

2. Setting the time window w_j of the price aggregator, to have open start times and closed end times.

This way, all the segment price events are processed within w_j allowing for correct results according to the guard.

Guard map 3 shows how the *consecutiveDerivation*(A, E) is mapped into an application by ensuring that derived events of type E, whose occurrence times equal to the window end time, are not processed in the same window but in a consecutive one.

Guard map 3 Consecutive derivation guard

function MAP_CONSECUTIVE_DERIVATION(A, E)
  $\forall e : E$, timestamp_policy (e) := end_of_window
  window_start_policy(A) := closed
  window_end_policy(A) := open
end function

Consider the example given for this guard in Section 4.4, in which `segment price` events that refer to w_j are used to derive `debit account` events within w_{j+1}. By setting the windows of the `billing composer` EPA (A_2) to have a closed start and an open end, `segment price` events created at the end of the window would be processed by the billing composer in the w_j's consecutive window.

Now that we have addressed the compile-time issues, showing how correctness guards can be defined in high-level constructs, and then mapped to

low-level constructs, we turn to the run-time issues. In the next section, we describe how these requirements are manifested in run-time in our specific system.

Table 5.1: Inclusion of event with variable edge properties

Window start	Window end	Inclusion condition
Open	Open	$T_s < \tau < T_e$
Open	Closed	$T_s < \tau \leq T_e$
Closed	Open	$T_s \leq \tau < T_e$
Closed	Closed	$T_s \leq \tau \leq T_e$

Chapter 6

Implementation algorithm

Adding linguistic constructs is a first step. The complementary step is to provide run-time support for the low-level language extensions that were introduced in Section 5. In this section, we address the enforcement of these constructs at run-time. We propose an algorithm that is based on a variation of the linear two-phase commit protocol [36]. The algorithm is set in the EPA level; whenever an EPA has more than one instance, the EPA entity is assumed to manage all of its instances (e.g., hold in its queue all the event instances relevant to all its EPA instances). Another alternative algorithm, based on the buffering technique [1] is described in general terms only.

For the sake of simplicity, we assume that the EPN has no cycles (i.e., there are no two EPAs with circular dependency: $\nexists \alpha, \beta.\ e \triangleright *_{\langle \alpha,\ldots,\beta \rangle} e' \wedge f \triangleright *_{\langle \beta,\ldots,\alpha \rangle} f'$), where α, β are the first and last EPAs of two dependency paths.

6.1 Variation on the linear two-phase commit algorithm

The algorithm is a variation of the linear two-phase commit algorithm. When the function `is_safe` (Algorithm 1), receives an EPA A and an event E as parameters, it ensures the "safe processing" of any e such that e is an instance of E, by A. `is_safe` waits until it ensures there is no event e' that needs to be processed before e. While the EPA waits it may process other events that are safe to be processed before e (i.e., their occurrence time is earlier than the occurrence time of e).

Algorithm 1 The `is_safe` algorithm, which ensures the safety of event derivation

```
1: initialization:∀α : EPA, latest_α := −∞
2: function IS_SAFE(A, e)
3:     if latest_A ≥ ts(e) then
4:         return safe
5:     end if
6:     for all B ∈ predecessor(A) do
7:         is_safe(B,e)
8:     end for
9:     while ∃e′ ∈ Q. ts(e′) < ts(e) do
10:        wait
11:    end while
12:    latest_A := ts(e)
13:    return safe
14: end function
```

The function $ts : E \rightarrow T$ (line 3) returns the timestamp of a given event. Q (line 9) is the event queue of EPA A to which events are queued before they are processed by A. Events are dequeued from Q only after their processing phase ends.

If an EPA expects to get its incoming events in the order of their creation, then it should activate the algorithm for every incoming event it processes. Otherwise, the EPA internal logic can compensate for processing events out-of-order and it should invoke `is_safe` in the following cases:

1. When it needs to close a temporal window w.
2. When an event arrives before its context start event.
3. When the EPA operates in immediate mode.

The invocation of `is_safe` is needed in the first case, since an earlier event e may have an occurrence time that entitles e to be included in the closing window w. The invocation in the second and third cases is done to ensure the correct association of events to their temporal context. To that end, we need to take a closer look at the event order in the event queue. The function `insert_event` (Listing 2) is used insert a given event e into the queue (Q) of some EPA.

Listing 2 Inserting a given event in its proper position in the event processing queue

```
function INSERT_EVENT(e, Q)
    if ∄e′ ∈ Q. ts(e′) = ts(e) then
        insert e to Q by ts(e)
        return
    end if
    if e′ is context_initiator then
        if window_start_policy(A) = closed then
            put e before e′
        else
            put e after e′
        end if
    else if e′ is context_terminator then
        if window_end_policy(A) = closed then
            put e after e′
        else
            put e before e′
        end if
    else
        put e after e′
    end if
end function
```

By combining the methodology of Listing 2, and the activation of `is_safe` (Algorithm 1), we ensure that events will be processed within the boundaries of their correct temporal context.

To prove the correctness of Algorithm 1, we must prove Claim 1.

Claim 1 *If EPA A returns a* `safe` *response, (lines 4, 13), in time t_{safe}, then A will not receive or send event e_i in time $t_i > t_{safe}$, with occurrence time $t_{oi} < t$.*

The claim is proved by induction according to the topological order of the EPAs in the EPN direct acyclic graph.

Proof Induction base: The induction base is proved by noticing that the first EPA A_1 in the topology order can only get raw events. It is assumed that raw events arrive in their correct chronological order, and since timestamp t is earlier than the current time, no raw event with an occurrence

time earlier than t can arrive at the point of the `is_safe` activation. The algorithm returns "`safe`" in line 4 and in line 13. Upon the first activation of the algorithm on A_1, the condition on line 2 will be evaluated to `false`. Therefore, the algorithm will return from line 13. In line 10, A_1 waits until all the events in its processing queue (Q) have timestamps that are bigger than t. Since the timestamps of derived events are always bigger or equal to the timestamps of their derivers, A_1 will not derive events with occurrence times earlier than t, after the function return in line 13. Upon later activations, the function may return from line 4. When this is the case, it means a previous activation returned from line 13 with a timestamp $t' > t$; thus, all derived events will have timestamps bigger or equal to t', which also satisfies the safety requirement.

Induction step: Let A_i be the i^{th} EPA in the topological order induced by some EPN. The validity of line 4 is correct by the same justifications made in the induction base. Accordingly, when the algorithm reaches line 11, all A_i's predecessors will not derive events with timestamps earlier that t—assuming the communication times are negligible and that detection times in all EPAs are synchronized. A_i can also get raw events, but since we assume that raw events arrive in their correct chronological order, and since time t is earlier than the current time, no raw event with occurrence time earlier than t will arrive A_i. This implies that after A_i gets a safe answer from all the preceding nodes in the topological order, it will not get events with timestamps than earlier or equal to t. In line 10, A_i waits for events with timestamps $< t$ to finish their processing phase. Since by all derivation policies the timestamp of the derived event is greater than or equal to the one of its deriver, A_i **will not derive** events with timestamps greater than t.

In the worst case, every activation of the algorithm at a distinct time t might be propagated to all the agents in the EPN, thus add $O(|EPN|)$ overhead to the execution time. To handle this pathological scenario we have identified scenarios in which the backwards propagation (line 7) of Algorithm 1 is not needed. This fact is based on the following distinction: if an EPA is not a "time-assigning" node (i.e., its timestamp assigning policy (Section 5.2) is detection-time based), it always derives events in chronological order. Therefore, we sign EPAs as "time-assigning" in compile time and prevent the propagation of Algorithm 1 beyond nodes that are signed

as “time-assigning”.

6.2 An alternative approach

Another algorithm that can be applied as alternative to Algorithm 1, is a variation of the buffering technique, proposed by Widom et al. [1], named the *buffering technique*. Originally designed to deal with cases in which raw events arrive out-of-order, raw events are not immediately processed at the moment they are discovered. Rather, they are buffered into an input buffer, in which events are ordered by their occurrence time. Events from the input buffer are processed only after a certain predefined time constant (τ) elapses from the moment they are discovered. (i.e., an event that was discovered in time t will be processed in time $t + \tau$). This approach can be adjusted to handle derived events rather than raw events, by computing a time constant for each EPA, based on the computation upper bound of the longest path leading to it. To achieve that, we would need to have the ability to assign a computation upper bound for each EPA, so the upper bound of a computation path will be the sum of the upper bounds of the EPA that composes it.

The quandary is, how do we statically compute a computation upper bound for an EPA. Even if we overlook the fact that such a boundary may not even exist, (by merely relaying on the average processing time plus k standard deviations), we will always be in the dilemma between performance, and correctness. If we assign the upper bound to be too big, we will increase the latency of the each path. On the other hand if we will assign it to be too small, we could risk that the computation time will exceed the computed upper bound, thus compromising the correctness of the result.

6.3 Performance comparison

As an attempt to asses the performance of the two algorithms above, we conducted an experiment, implemented by undergraduate students. In this experiment, a generic event processing framework was developed in Java. Based on this framework, an application based on the fair auction scenario (Figure 3.1) was implemented in three flavors:

1. The first, we named the "naïve" version, was implemented without any correctness mechanism.

2. The second flavor was enhanced by the two-phase correctness mechanism (Section 6.1).

3. Finally, the third flavor was implemented with the buffering algorithm (Section 6.2).

We ran the three flavors of applications on the same data set, and compared their performance by two criteria; latency an throughput. Latency was defined as the average time span between the point in time in which an event was detected by an EPA, until the point in time that event was derived by that EPA. Throughput was measured as the number of events processed by all the EPAs in the system in average, per second. Table 6.1 shows the results of the experiment.

Table 6.1: Performance comparison between the naïve approach, two-phase commit and buffering algorithms, implemented on the fair auction scenario.

	naïve		Buffering Algorithm		Two-phase commit	
Bids/ auction	Through.	Lat. (msec)	Through.	Lat. (msec)	Through	Lat. (msec)
10	117.06	10.2	4.48	223.67	14.92	67.42
20	119.5	9.45	1.11	904.14	15.68	64.01
50	107.14	9.62	0.92	1090	15.98	62.56
100	130.83	8.04	0.67	1498.80	13.60	73.54
200	171.87	5.97	1.18	870.26	10.49	105.51

The buffering algorithm was configured with $\tau_{max} = 20_{ms}$ as maximum delay threshold. This time was hand-tuned, to ensure that the results of the bid will be correct. As expected, both algorithm presents overheads both in terms of throughput, and of latency. In both criteria (as can be seen in Figures 6.1, 6.2), the two-phase commit proved to be superior over the buffering algorithm. Moreover, whereas the buffering algorithm is simple and therefore we believe its implementation could not be dramatically improved, the implementation of the two-phase commit algorithm which produced the results may be implemented more carefully, so that thread utilization will increase, (e.g., by using thread-pools, choosing synchronization mechanism more carefully, etc.).

Figure 6.1 The buffering and the two-phase commit algorithms compared to a naïve implementation by their throughput

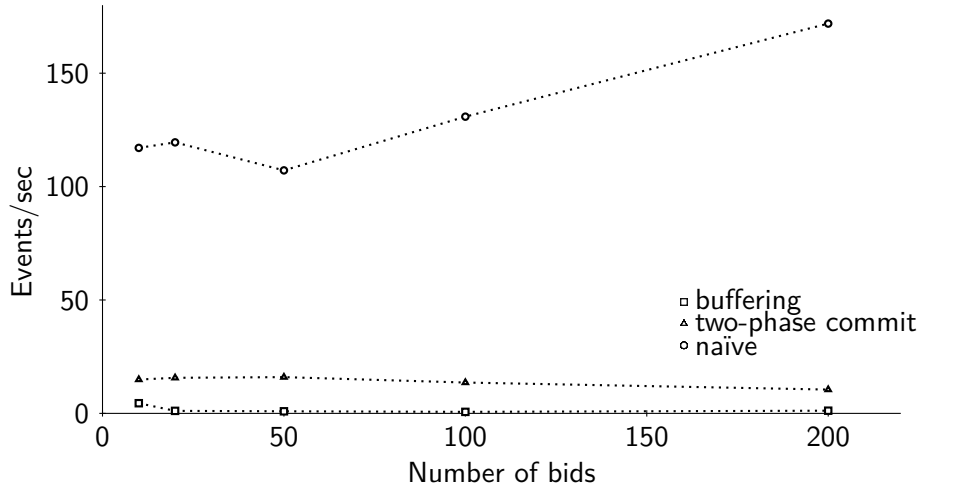
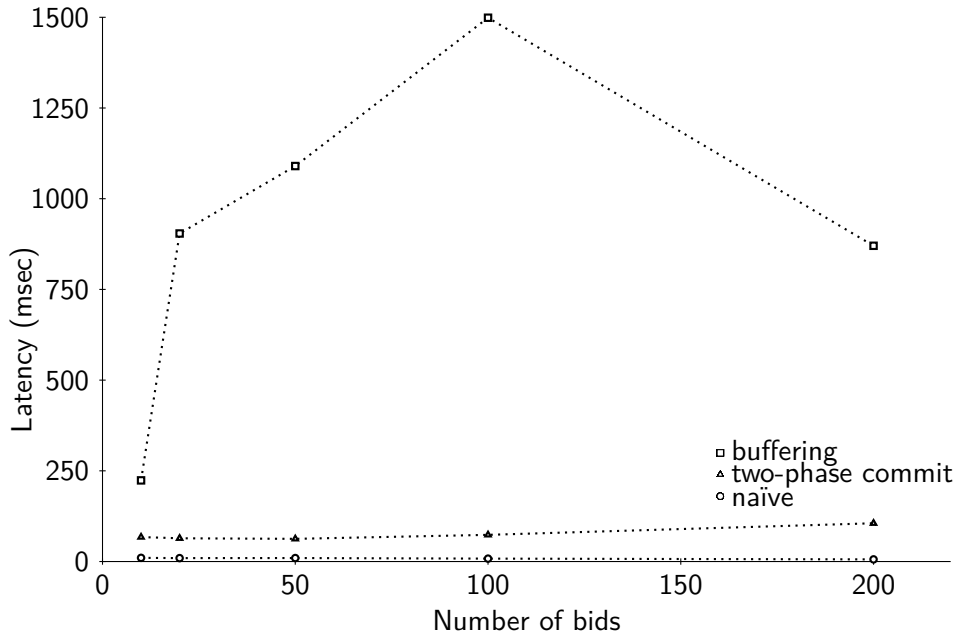


Figure 6.2 The buffering and the two-phase commit algorithms compared to a naïve implementation by their latency



6.4 Summary

Correctness guards can be automatically deployed into event processing applications, so that the timing in which events are derived will not change expected result of the application. This is achieved by installing "valves" on certain EPAs, that will run the `is_safe` algorithm prior to the derivation of each event. We have compared the performance of our solution to the buffering technique proposed other event processing languages such as CQL [1], and seen significant improvements both in terms of throughput ($\tilde{4}$–10X), and both in terms of shorter latency. However, the compression to the same code that operated without any correctness mechanism showed up to 10X slowdown in terms of throughput, and a similar degradation in terms of latency. Since using correctness mechanism is barring a toll beside it, not all applications should be using them all the time. In some cases, an occasional faulty output is a price the developer is willing to pay, as long as his application has good performance.

Chapter 7

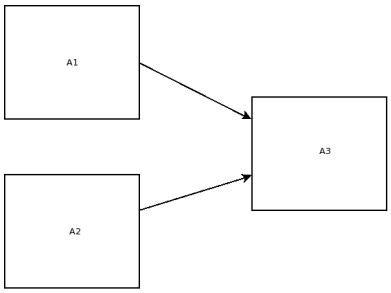
Temporal assertion language (Tal)

7.1 Introduction and motivation

The motivation behind Tal, stems from the fact that correctness guards defined in Section 4, cannot solve all the correctness issues that may be found in event processing applications. Some correctness problems (other then the one we showed in Section 3), can be solved by our generalized definition of correctness guards. Tal enables its users to customize their guards to fit their needs; for example, let us assume a situation in which a developer would like to ensure FIFO ordering [9] between two connected EPAs, A and B, where B consumes events which are derived by A. Although this is in fact a private case of the `fairness guard` (Section 4), in which the two dependency paths π and ψ are the same, applying it to achieve FIFO ordering is somewhat awkward. Instead, one can write a simple guard using Tal, to achieve FIFO ordering. Another example may come in the form of the "biased route", in which instead of striving to fairness between two derivation paths, the developer actually wants to ensure that events derives from π, will have a fixed latency with respect to events derived from path ψ. The third example that may require Tal refers to V shape network topology, shown by Figure 7.1.

Figure 7.1 has two derivation paths in it; the first is the path that consists the EPAs A_1 and A_3, and the second that consists the EPAs A_2 and A_3.

Figure 7.1 EPN with V-shape topology, to which two paths have different derivation policies



TAL enables users to enforce a certain derivation scheme to one of the paths, (say FIFO ordering along the path $\langle A_1, A_3 \rangle$), and another derivation scheme on the second derivation path $\langle A_2, A_3 \rangle$. This flexibility achieved when using TAL, makes it worth while.

TAL is a language that provides the means to define and enforce the correct processing of events in applications, written in some arbitrary event processing language. TAL is an assertion-based declarative language, that controls the timing in which events are derived.

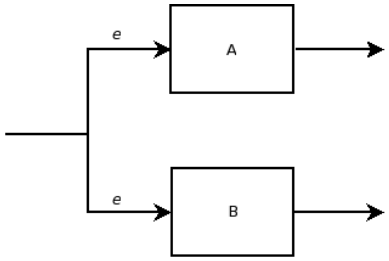
Expressions in TAL are based on propositional logic. Much like other logic-based programming languages such as Prolog [8], algorithms are not explicitly implemented by programmers. Rather, algorithms are described by programmers in logic, and it is up to the compiler of the language find a solution to the problem described. Thus in TAL, rather than expressing "how" to keep the correct derivation of an EPA, the programmer states "what" is the correct derivation on EPA. The constructs of the language are propositional logical expressions. The generated code "guards" the correctness of the proposition it represents, by making sure it applies for each event before it is derived by a given EPA. Hence, the name of a such correctness propositions in TAL is *guards*. Guards are restricted to one (implicit) variable, which is the candidate event for derivation. A guard is composed of one or more *assertions*. Assertions are in fact modus ponens [32] propositions, and thus contains two parts; the *require* part, and the *ensure* part. Both of which contain one or more boolean statements. After composing a guard, a developer can associate it to one or more EPAs in its EPN. At runtime, the code generated from the guard is deployed into its target EPA,

and operates like a valve: it allows its associated EPA to derive events only when the guards it represents "allows" it to.

7.2 The expressive power of Tal

Although useful in some scenarios, some correctness requirements cannot be achieved by guards as we defined them, and thus cannot be solved by TAL programs. Such a requirement is the *mutual exclusion* problem. Terminology borrowed from [29], the mutual exclusion is a case in which the an event stream serves an input for two distinct EPAs, A and B (Figure 7.2). The

Figure 7.2 The mutual exclusion problem, a correctness issue that cannot be solved by TAL



mutual exclusion derivation states that if an event e is derived by EPA A, it should not be derived by EPA B, and vice versa. Applying the mutual exclusion derivation requirement on A and B is beyond the expressive power of TAL, since there is no derivation path that connects the two EPAs. Thus, a derivation guard installed on one of the EPAs will not be able to know whether the other one has derived e or not.

A solution to problem such as the one shown above, can be achieved by reaching consensus between EPA's A and B, on who should derive event e. Consensus may be achieved by algorithm such as [6, 21]. It should be noted, that both algorithms require that at most f EPA instances will fail at a given time, where the total number of EPAs $2f + 1$, which is a reasonable requirement for event processing system. Moreover, the complexity of these algorithm is bounded by the number of EPA instances, unlike the `is_safe` algorithm presented in Section 6, in which the complexity is bounded by the longest derivation path in the EPN.

7.3 Grammar

The first section in TAL programs is the declarations section. This section may variable declarations of type `path`. `path` is a set of distinct labels, each stands for an EPA in the EPN.

1. $\mathcal{P} \rightarrow \ell$ `:= path` $\langle \ell_1, \ldots, \ell_n \rangle$, where ℓ and ℓ_i are distinct labels from an unbounded set of labels $\mathscr{L}$. We assume that each label $\ell_i \in \langle \ell_1, \ldots, \ell_n \rangle$ has one-to-one correspondence to an EPA. Variable of type `event` may also be used by the second section, but those doesn't need to be declared. Their type is automatically synthesized by the compiler.

   The guard section is a list of *temporal assertions*. An assertion (α) consists of two parts; the *require* part (ρ), and the `ensure` part (ϵ).

2. $\alpha \rightarrow \rho\ \epsilon$

   Both parts consists a list of boolean statements.

3. A statement has four forms; the first is a *temporal statement*, (i.e., a comparison between the timestamps of two *temporal objects*, which can be either an event, an EPA or a PATH). The second is a *derivation* statement. A derivation statement receives an event as its first parameter, and an EPA or a path as its second parameter. It returns `true` if a the event is derived by its second parameter. In case its second parameter is an EPA, the function returns `true` if the given event was directly derived by that EPA. If the second argument is a path, the function returns `true` if the event was directly derived by either one of the EPAs in the path. The third form indicates whether a given event is a *context terminator* event. Finally, a statement may be the negation of another statement.

   $$\begin{array}{ll} t \rightarrow & \texttt{event} \mid \ell \\ \sigma \rightarrow & \tau(t_1) \otimes \tau(t_2) \mid \\ & \texttt{event} \rhd \ell_4 \mid \\ & c_t(\texttt{event}) \mid \\ & \texttt{!}\,\sigma' \end{array}$$

   The '$\otimes$' sign is a place holder for one of the following relations: $<$, $>$, $=$, $\leq$ or $\geq$. The function τ operates on events, paths and contexts. When operating on an event, it simply returns the timestamp

associated to that event. When operating on an EPA, it returns a timestamp assigned and maintained by the TAL runtime, that holds the minimal timestamp of all events in the processing buffer. If the processing buffer of the EPA is empty, it returns ∞.

When operating on a path $\langle \alpha_1, \ldots, \alpha_n \rangle$, it returns the minimal timestamp of it EPAs: $\min_{i=1}^{n} \tau(\alpha_i)$.

The complete grammar of TAL is shown by Figure 7.3.

Figure 7.3 The abstract context free grammar of TAL, a declarative temporal assertion language.

start	$\rightarrow$ *declaration* *guard*
declaration	$\rightarrow$ **begin declare** $d_1, \ldots, d_n$ **end declare**
d	$\rightarrow \ell$ **:= path** $\langle \ell_1, \ldots, \ell_n \rangle$ |
guard	$\rightarrow a_1, \ldots, a_n$
a	$\rightarrow$ **begin assertion** $r\ e$ **end assertion**
r	$\rightarrow$ **begin require** $s_1, \ldots, s_n$ **end require**
e	$\rightarrow$ **begin ensure** $s'_1, \ldots, s'_m$ **end ensure**
t	$\rightarrow \ell$ | **event**
s	$\rightarrow$ **event** $\triangleright\ \ell'$ |
	$\rightarrow \tau(t) \otimes \tau(t)$ |
	$\rightarrow c_t($**event**$)$ |
	$\rightarrow$ **!** s'
$\otimes$	$\rightarrow < \mid > \mid \leq \mid \geq \mid =$

7.4 Examples

Listing 3 shows a piece of TAL code, that handles the correctness issues previously shown in Section 3.2. More concretely, it ensures that by installing a valve on A_3 (Figure 3), fairness between the "cash path" and the "credit path" is maintained. The valve makes sure that before A_3 derives an event, it queries its derivation path. If that event is derived by the `cache_path` (line 8), it ensures there is no earlier event still being processed by `credit_bid` (line 11). The same symmetric check is conducted each time A_3 derives an event originated from the `credit_path` (lines 16 and 19).

Listing 3 Fairness guard, implemented in TAL

```
1 begin declare
    credit_path := path (a1, a2, a3);
3   cash_path := path (a3);
  end declare
5 begin guard
    begin assertion(a3)
7     begin require
        (e |> cash_path);
9     end require
      begin ensure
11      ts(e) >= ts(credit_path);
      end ensure
13  end assertion
    begin assertion(a3)
15    begin require
        e |> credit_path;
17    end require
      begin ensure
19      ts(e) >= ts(cash_path);
      end ensure
21  end assertion
  end guard
```

The second code example (Listing 4) shows how a program in TAL can ensure the nested derivation semantics, presented at Section 4.3.

The statement at line 11 ensures that the event e, (which is required to be a context terminator at line 8), will not be derived by the EPA to which the guard is associated with, as long there are event earlier (i.e., with a smaller timestamp), than e still being processed in path `p`.

7.5 Tal code generation

TAL functionality is not restricted to a specific event processing system. It is applicable to every system that conforms with the principles drawn by [14], in which every EPA has an event buffer, containing events designated to it, from which it will derive its output events.

We have implemented one backend for TAL, which generates code in Java. The generated code is then linked into our basic implementation,

Listing 4 Nested derivation guard, implemented in TAL

```
begin declare
  p := path (a1,a2);
end declare

begin guard
  begin assertion(a2)
    begin require
      isterminator(e);
    end require
    begin ensure
      ts(e) < ts(p);
    end ensure
  end assertion
end guard
```

previously described in Sections 6. Although the beckend we have implemented generates code that relies on our runtime package, we believe that it feasible to implement other backends, that will generate code that could be incorporated to other implementations as well. Before we can show the code generated by the example shown by Listing 4, we first need to go over the essentials of our runtime package. The package assumes the existence of two basic external classes (defined in the system to which the correctness code is designated to):

`EPA` —represents an event processing agent.

`Event` —represents an event instance.

On top of those classes, the correctness runtime system contains the following classes:

`Path` —an interface that defines a single method defined as `int timeStamp();`. This interface is realized by all the classes above, (namely `EPA, Event` and `Path`). The implementation of the method `timeStamp` is by the definition described in Section 7.3.

`Condition` —is an interface that represents all the boolean condition in TAL. The interface exposes one method: `boolean applies();`, which

when implemented in implementing classes, indicates whether the underlying condition applies or not. The interface `Condition` is realized by the following classes:

ContextTerminatorCondition —

forged by the constructor `ContextTerminatorCondition(Event e)`, it indicates whether the given event is a context terminator.

`DeriveCondition` —indicates whether two events passed to its constructor (with the prototype `DeriveCondition(Event l, Event r)`), derived by one another.

`Relation` —represent a binary relation between two given temporal objects. The method `applies` of this class returns `true` if the relation it represents ($<, >, \leq, \geq$ or $=$), applies to the two temporal objects passed to its constructor.

`Assertion` —is class that holds two conditions in it; one is the *required* condition, and the other one it the *ensure* condition. Assertion represents a correctness condition deployed by TAL programs. An assertion is guarding the program's correctness, by asserting if the *required* condition applies, so does the *ensure* condition.

The FIFO ordering guard implemented in TAL (Listing 4), results the Java code shown in Listing 5 to be generated.

Listing 5 Java code generated by the TAL BE, out of the program in Listing 4

```
Assertion createAssert(Path p, EPA a2){
2   final Event currentEvent = TAL.nextEvent(a2);
    Condition req = new ContextTerminatorCondition(currentEvent);
4   Condition ens = new Relation(currentEvent, p, RelationTy.LT);
    return new Assertion(req, ens);
6 }
```

The method `createAssert` is a factory method that creates the assertion to be installed on EPA A_2. The method receives a two arguments; the first is `p` of type `Path`, that is defined in the declared section of Listing4, and the second is `a2`, which is the EPA on which the assertion is deployed. The

Listing 6 Assertion generated by TAL, incorporated in the runtime, to enforce correctness.

```
while (true){
  Assertion a = createAssertion(...);
  synchronized(a){
    while (!a.apply())
      a.wait();
  }
  //now it is safe to process the next event
}
```

Listing 7 Implementation of the `apply` method in class `Assertion`

```
public boolean apply(){
  boolean ret = !require.apply() || ensure.apply();
  if (ret) ret.notifyAll();
  return ret;
}
```

statement on Line 2 returns the next event to be processed on `a2`. Next, two conditions are initiated: one of type `ContextTerminatorCondition`, and the second of type `Relation`. The conditions are initiated with respect to Lines 8 and 11 of Listing 4. Listing 6 shows how the code generated by the TAL BE is incorporated in the runtime of the event processing system.

The implementation of the method `apply` (Line 4) is shown by Listing 7.

7.6 Summary

TAL is a domain-specific language that enables its users to define correctness guards in a logic formula. The formula is then processed by the TAL compiler, which generates code that could be integrated in the underlying event-processing system. (Our implementation generated Java code, but new code generators could be implemented to generate code in other languages as well). At runtime, the generated code is installed on an EPA, to ensure that the correctness guards applies for each event derived by the EPA. The expressive power of TAL is limited to the power of the correctness guards as we defined them. We have encountered with correctness problems

that cannot be solved by our algorithm, and thus cannot be expressed (and for that reason solved) by TAL.

Chapter 8

Summary

The event processing area is still young, and as such, has not defined its correctness criteria. This situation is similar to the state of the database field in the early days of multi-user database access, in which correctness anomalies led to the introduction of the concept of transaction [4]. In this thesis, we dealt with the correctness of derived events, which is one aspect of the correctness issues.

We obtained the three guards that we described in this paper by analyzing various event processing applications. We introduced the notion of correctness guards and added templates that enable system designers to define these correctness guards. We also automatically compile them to a specific event processing execution language. This study serves as a proof of concept for the feasibility of this approach. This approach has the potential to contribute to the usability of event processing systems by raising the level of abstractions and by reducing the amount of system-related workarounds. While these correctness guards are model-independent and are applicable in every type of event processing implementation, we have shown their applicability within a rather generic model of an event processing network.

Finally, let us put our solution by three correctness criteria defined by Claypool et al. [22]. The three criteria are:

1. **Ordered** applications. Applications that their output depends input generation order, rather then on their input receiving order.

2. **In time** derivation. This property hold if an event is derived in the early possible moment.

3. **Permanently valid** vs. **Eventually valid**. Permanently valid implies there will be know spurious event generation, whereas eventually valid means that finally, the correct event will be generated.

Scrutinizing our solution under the criteria above, reveals that the first correctness criteria (trivially) holds for our solution, since we only take derived events onto consideration. The second property does not apply for our solution, since before each event derivation, EPAs which have a guard installed on them, runs the `is-safe` algorithm. Finally, our solution respects the stronger condition of the two validity condition (i.e., permanently condition), since `is-safe` ensures that the derived events are not spurious under the assertion stated in the correctness guard.

The extensions to the language, along with the execution algorithm, are part of a "second generation event processing platform" that is under construction to implement proactive computing [11].

The contribution of this thesis was to offer a platform-independent mechanism, to deal with potential correctness issues in event processing applications. Examples of such problems are studied in this thesis, their problems are formalized and generalized into an artifact named correctness guard. Since we know there are more correctness issues out there that could be solved by the guards mechanism, we have composed a domain specific propositional-logic language named Tal, that enables its users to define new correctness guards that may solve some of these problems. We have also implemented a compiler for Tal, that generates code in Java, that can then be deployed into an event processing system, so that events will be derived in the system only after they have satisfied the correctness condition stated in the guard. Alas, guards as we defined them, are not the silver bullet that will slay all the correctness problems in event processing applications. Some problems (e.g,. The mutual exclusion problem), may need a more elaborate runtime mechanism, as well as another runtime algorithm to enforce it. One that will bot be limited to a derivation path, but will also have the ability synchronize derivation in the scope of the entire EPN.

Bibliography

[1] Arvind Arasu, Shivnath Babu, and Jennifer Widom. The cql continuous query language: semantic foundations and query execution. *The VLDB Journal*, 15(2):121–142, June 2006.

[2] Arvind Arasu, Mitch Cherniack, Eduardo F. Galvez, David Maier, Anurag Maskey, Esther Ryvkina, Michael Stonebraker, and Richard Tibbetts. Linear road: A stream data management benchmark. In *VLDB*, pages 480–491, 2004.

[3] Roger S. Barga, Jonathan Goldstein, Mohamed H. Ali, and Mingsheng Hong. Consistent streaming through time: A vision for event stream processing. In *CIDR*, pages 363–374, 2007.

[4] Philip A. Bernstein, Vassos Hadzilacos, and Nathan Goodman. *Concurrency Control and Recovery in Database Systems.* Addison-Wesley, 1987.

[5] Gérard Berry and Georges Gonthier. The esterel synchronous programming language: Design, semantics, implementation. *Sci. Comput. Program.*, 19(2):87–152, 1992.

[6] Tushar Deepak Chandra and Sam Toueg. Unreliable failure detectors for reliable distributed systems. *J. ACM*, 43(2):225–267, 1996.

[7] Mani K. Chandy, Opher Etzion, and Rainer von Ammon. 10201 Executive Summary and Manifesto – Event Processing. In K. Mani Chandy, Opher Etzion, and Rainer von Ammon, editors, *Event Processing*, number 10201 in Dagstuhl Seminar Proceedings, Dagstuhl, Germany, 2011. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, Germany.

[8] W.W.F. Clocksin and C.C.S. Mellish. *Programming in Prolog: Using the Iso Standard*. Springer-Verlag, 2003.

[9] Thomas H. Cormen, Clifford Stein, Ronald L. Rivest, and Charles E. Leiserson. *Introduction to Algorithms*. McGraw-Hill Higher Education, 2nd edition, 2001.

[10] Edsger W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Commun. ACM*, 18(8):453–457, 1975.

[11] Yagil Engel and Opher Etzion. Towards proactive event-driven computing. In *DEBS*, pages 125–136, 2011.

[12] Opher Etzion. Temporal perspectives in event processing. In *Principles and Applications of Distributed Event-Based Systems*, pages 75–89. IGI Global, 2010.

[13] Opher Etzion, Yonit Magid, Ella Rabinovich, Inna Skarbovsky, and Nir Zolotorevsky. Context aware computing and its utilization in event-based systems. In *DEBS*, pages 270–281, 2010.

[14] Opher Etzion and Peter Niblett. *Event Processing in Action*. Manning Publications Company, 2010.

[15] Bugra Gedik, Henrique Andrade, Kun-Lung Wu, Philip S. Yu, and Myungcheol Doo. Spade: the system s declarative stream processing engine. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, SIGMOD '08, pages 1123–1134, New York, NY, USA, 2008. ACM.

[16] Bugra Gedik, Kun-Lung Wu, Philip S. Yu, and Ling Liu. A load shedding framework and optimizations for m-way windowed stream joins. In *ICDE*, pages 536–545, 2007.

[17] J.-H. Hwang, M. Balazinska, A. Rasin, U. Cetintemel, M. Stonebraker, and S. Zdonik. High-availability algorithms for distributed stream processing. In *Data Engineering, 2005. ICDE 2005. Proceedings. 21st International Conference on*, pages 779 – 790, april 2005.

[18] Shawn R. Jeffery, Gustavo Alonso, Michael J. Franklin, Wei Hong, and Jennifer Widom. A pipelined framework for online cleaning of sensor

data streams. Technical Report UCB/CSD-05-1413, EECS Department, University of California, Berkeley, Sep 2005.

[19] Christian Kästner, Sven Apel, Salvador Trujillo, Martin Kuhlemann, and Don S. Batory. Guaranteeing syntactic correctness for all product line variants: A language-independent approach. In *TOOLS (47)*, pages 175–194, 2009.

[20] Matthias Keller, Lothar Thiele, and Jan Beutel. Reconstruction of the correct temporal order of sensor network data. In *IPSN*, pages 282–293, 2011.

[21] Leslie Lamport. The part-time parliament. *ACM Trans. Comput. Syst.*, 16(2):133–169, 1998.

[22] Jin Li, Kristin Tufte, Vladislav Shkapenyuk, Vassilis Papadimos, Theodore Johnson, and David Maier. Out-of-order processing: a new architecture for high-performance stream systems. *PVLDB*, 1(1):274–288, 2008.

[23] David C. Luckham. *The Power of Events: An Introduction to Complex Event Processing in Distributed Enterprise Systems*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001.

[24] Marcelo R. N. Mendes, Pedro Bizarro, and Paulo Marques. A performance study of event processing systems. In *TPCTC*, pages 221–236, 2009.

[25] C.M. Ong. *Dynamic simulation of electric machinery: using MATLAB/SIMULINK*. Prentice Hall PTR, 1998.

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

[27] Sathya Peri and Krishnamurthy Vidyasankar. Correctness of concurrent executions of closed nested transactions in transactional memory systems. In *ICDCN*, pages 95–106, 2011.

[28] Ashish Ray. Oracle data guard: Ensuring disaster recovery for the enterprise. an oracle white paper, March 2002.

[29] Abraham Silberschatz, Peter Baer Galvin, and Greg Gagne. *Operating System Concepts*. Wiley Publishing, 8th edition, 2008.

[30] Peter A. Tucker and David Maier. Exploiting punctuation semantics in data streams. In *ICDE*, page 279, 2002.

[31] Wil M. P. van der Aalst, Niels Lohmann, Marcello La Rosa, and Jingxin Xu. Correctness ensuring process configuration: An approach based on partner synthesis. In *BPM*, pages 95–111, 2010.

[32] Robert L. Vaught. *Set theory - an introduction (2. ed.)*. Birkhäuser, 1995.

[33] ventana research. Innovating with operational intelligence and complex event processing. Technical report, ventana research, 2011.

[34] Zhanyong Wan and Paul Hudak. Functional reactive programming from first principles. In *PLDI*, pages 242–252, 2000.

[35] Mingzhu Wei, Mo Liu, Ming Li, Denis Golovnya, Elke A. Rundensteiner, and Kajal T. Claypool. Supporting a spectrum of out-of-order event processing technologies: from aggressive to conservative methodologies. In *SIGMOD Conference*, pages 1031–1034, 2009.

[36] Ouri Wolfson. A comparative analysis of two-phase-commit protocols. In *ICDT*, pages 291–304, 1990.

לקראת נכונות במערכות לעיבוד מאורעות

אליאור מלול

לקראת נכונות במערכות לעיבוד מאורעות

חיבור על מחקר

לשם מילוי חלקי של הדרישות לקבלת התואר
מגיסטר למדעים במדעי המחשב

אליאור מלול

הוגש לסנט הטכניון – מכון טכנולוגי לישראל
טבת ה'תשע"ג חיפה דצמבר 2012

המחקר נעשה בהנחיית פרופ' יוסי גיל וד"ר עופר עציון בפקולטה למדעי המחשב.

אני מודה לטכניון על התמיכה הכספית הנדיבה בהשתלמותי.

תקציר

למרות שהטכנולוגיה שמאחורי מערכות לעיבוד מאורעות עדיין בהתהוות, תוצאות מחקרים אמפריים שנערכו בנושא הטעמת מערכות אלו [21] מראות כי ארגונים מעידים על עצמם כי היכולת שלהם להטמיע מערכות המבוססות על טכנולוגיות אלו מוטלת בספק. אחד המכשולים העומדים בפני הטעמה מוצלחת של מערכות לעיבוד מאורעות, היא היכולת להתאים את דרישות הנכונות של הישומים לאינטואציה של המשתמשים שלה. זאת מכיוון שדרישות הנכונות של ישומים רבים תלויים במימד הזמן [3], עובדה המקשה על אכיפת דרישות הנכונות, כאשר מממשים ישום מסויים.

עיבוד המאורעות נעשה ביחס לציר הזמן של המערכת, בעוד מאורעות עלולים להגיע אל המערכת עם חותמת זמן שיוצרה יחסית לציר הזמן של מייצר המאורע, עובדה שעלולה להקשות על השריית יחס־סדר טמפורלי בין מאורעות. הקושי בקביעת הסדר מוביל לעיתים לחוסר התאמה בין המאורעות שהתכוון ליצור המפתח, לבין אלו שנוצרים בפועל על ידי הישום במקרה הרע, או ליצירת "מנגנוני־מעקף", שנועדו לגשר על אותן אי־התאמות, במקרה הטוב. מנגנונים עוקפים אלו הינם בגדר "רע הכרחי", שכן שפות לעיבוד מאורעות אינן כוללות מנגנוני נכונות מובנים אשר מאפשרים למפתח להביא לידי ביטוי את הדרישות הטמפורליות של הישום המדובר. לדוגמא, בפרק 3, מובאת דוגמאת הפלט של אפליקציה נאיבית, שמומשה באמצעות מערכת מסחרית מוכרת לעיבוד מאורעות. בתרחיש מתוארת מערכת למכירות פומביות, שבה הודעת המערכת באשר להצעה המנצחת איננה עומדת בקנה אחד עם ההגיון הבריא. העובדה שבדור הנוכחי של מערכות לעיבוד־המארועות עולה הצורך ביצירת "מנגוננ מעקף" שיתמודדו עם מקרים כאלה מעלה עוד יותר את המיומנות הנדרשת ממפתחי האפליקציות, עובדה שמקשה את הטמעתן. כאשר מסתכלים בקפידה על בעיות נכונות רבות בישומים לעיבוד־מאורעות, ניתן להבחין בעובדה, שרבות מהן מתרחשות עקב אופיים הטמפורלי של הישומים. את בעיות הנכונות הנ"ל ניתן לחלק לשתי קטגוריות עקריות:

1. מקרים בהם מאורע הפלט נגזר כתוצאה מדפוס של מאורעות קלט אחרים, והסדר

שבו מאורעות הקלט מסודרים משפיע על זהוי הדפוס (לדוגמא – זהוי של מגמה, או של רצף מאורעות מסויים).

2. מקרים בהם מאורעות נגזרים בהקשר של חלון זמן (הקשר טמפורלי). ההכללה או אי־ההכללה של מאורעות מסויימים בחלונות זמן עשויה להשפיע על אופיו של המאורע הנגזר. בעיות סנכרון בין זמן סגירת החלון לזמן המאירוע, עלול לגרום לתוצאות שאינן מתאימות לדרישות האפלקציה.

לא די בזה שבדור הנוכחי של מערכות עיבוד המאורעות בעיות הסינכרון הטמפורליות עלולות להשפיע על נכונות התוצאות המופקות, אלא גם שאיבחונן של הבעיות הללו הוא באחריות המתכנת. גם לאחר איבחונן, על המתכנת לפתור אותן בצורה ידנית, שלפעמים נראית מגושמת, ואף עלולה לפגוע בביצועי המערכת.

התיזה הזאת עוסקת **בנכונות טמפורלית של עיבוד מאורעות**. גורם האחראי לרבות מבעיות הנכונות בישומים לעיבוד מאורעות. מספר תרחישים העוסקים בנכונות טמפורלית הוצגו בעבר [6], וגם נחקרו על־ידי אחרים [8, 3, 7]. בעבודות אלו נעשה שימוש במנגנוני פיצוי, כמו גם פיתוחים לשיטת־החייץ של מאורעות [1], תפיסות השונות מהותית מזו המוצגת בתיזה זאת. מטרתנו היא לתת למפתחי היישומים את הכלים להגדיר "מנגנוני־נכונות טמפורליים" המתהדרים בצורה אוטומטית לקטעי קוד אשר יוטמעו בשפת עיבוד המוראות. הקוד הנ"ל יכפה על המערכת את ההתנהגות (הנכונה), לה מצפה מפתח היישום. יתרתה של התיזה עוסקת בפרטים השונים הקשורים בבעיה זאת.

פרק 2 עוסק בעבודות מחקריות העוסקות בנושאים הקשורים לנושא זה. הפרק מסביר בקווים כלליים את המודל של עיבוד מאורעות, כמו גם את המושגים שבהם נעשה שימוש בהמשך התיזה. בפרט, מפורט המושג הקשר. דגש מושם על הקשרים מסוג "חלונות נעים", שכן נעשה בהם שימוש בדוגמאות המבואות בהמשך התיזה. חלונות נפתחים ונסגרים ע"י מאורעות מיוחדים הנקראים "נקדים". הנקדים יכולים להיווצר הן ע"י ייצרני מאורעות פנימיים למערכת, או ע"י מקורות חיצוניים למערכת. אפילו לאור ההנחה שמאורעות המגיעים ממקורות חיצוניים מגיעים למערכת ע"פ סדר היווצרותם, לא מובטח כי מערכת זמן הריצה תגזור את המאורעות לפי הסדר, כפי שמראה הדוגמא בפרק 3. הפתרון המוצע בתיזה זאת, הוא השראה מרעיון ה"זקיף" של דיקסטרה [4]. הזקיפים הינם מנגנוני־נכונות המבוטאים בשפה עילית. לאחר הידורם, מיוצרים תוצרים אשר משולבים בתוך המערכת לעיבוד המאורעות. התוצרים הללו אוכפים על המערכת את דרישות הנכונות כפי שבוטאו ע"י הזקיף בזמן ריצה. ברעיון על פיו דרישות נכונות מבטואות בשפה עילית, ונאכפות בצורה אוטמטית נעשה שימוש בתחומים נוספים, כגון ניהול ומידול של תהליכים עסקיים [11]. גם בעיקרון שבו נעשית הפשטה לאילוצי הזמן של המערכת נעשה שימוש בשפות כמו אסטרל [2]. שימוש בזקיפי מידע נעשה גם בתחום

מסדי־הנתונים היחסיים [01], כמו גם בתחום עיבוד המאורעות עצמו [5]. ההבדל בין העבודות הללו לתיזה הנ"ל, היא שבעוד שהזקיפים באיזכורים שהובאו לעי"ל מטפלים במקרים בהם יש כשלים של צמתים במערכת, אי הנכונות הטמפורלית בה אנחנו מטפלים מתרחשות גם כאשר כל הצמתים במערכת מתפקדים כראוי.

למרות שתיזה זאת מתרכזת בנכוונות טמפורלית של מערכות לעיבוד מאורעות, קיימים קשיים נוספים, כמו הבטחת התנהגות דטרמיניסטית וניתוח של מורכבות המערכת וההשפעה על ביצועיה. קשיים אלו הם מחוץ לתחום בו עוסקת התיזה. פרק 3 מתחיל בהצגת המונחים בהם יעשה שימוש בהמשך התיזה. מילון המונחים שאול מעבודתו של לוקהאם [9], אשר תבע לראשונה את המונח "רשת לעיבוד מאורעות", ובזה של "סוכן לעיבוד מאורעות". רשת לעיבוד מאורעות הינה גרף מכוון חסר מעגלים, אשר צמתיו הם סוכנים לעיבוד מאורעות. מאורעות נוצרים הן ע"י מקורות חיצוניים לרשת עיבוד המאורעות, והן ע"י צמתיה. המאורעות מועברים בין הצמתים ע"י "שטפי מאורעות". בהגיעו של מאורע אל מופע של סוכן לעיבוד מאורעות, האחרון מייצר מאורע אחד או יותר בתגובה, ושולח אותו ע"ג שטפי היציאה המחוברים אליו. מאורעות כאלו מכונים על־ידינו "מאורעות נגזרים". בהמשך פרק זה, אנו מדגימים את הבעיה שנדונה בתיזה דרך הצגת שני תרחישים. אנו מראים כי מימוש הבעיות בצורה נאיבית מניבות תוצאות שאינן נכונות. במהשך, אנו מציגים את גישתנו לפתרון הבעיות בפרק 4.

פרק 4 מגדיר שלושה סוגים של זקיפי־נכונות. שלושת סוגי הזקיפים סווגו על־ידינו ככאלו אשר היו יכולים לפתור את בעיות הנכונות הטמפורליות במגוון רחב של אפליקציות אותן סקרנו. זקיפי המידע הינם "זקיף ההוגנות", "זקיף הכללת הגזירה", ו"זקיף הגזירה הרציפה". המשותף לכל הזקיפים הללו, הוא שהם נועדו על־מנת לפתור בעיות נכונות טמפורליות, אשר נוצרו עקב תכונות של מאורעות נגזרים. לטענתנו, הוספת זקיפים אלו לארגז הכלים העומד לרשותם של מפתחי מערכות מהווה קפיצת מדרגה בפשטות הפיתוח של אפליקציות לעיבוד מאורעות.

פרק 5 דן בהרחבות השפה הנדרשות משפות לעיבוד־המאורעות הקיימות, על־מנת שיוכלו לתמוך במערכת הנכונות המוצעת על־ידינו. ההרחבות כוללות שני מנגנונים עקריים: הראשון הינו מנגנון השמת חותמת־הזמן למאורעות נגזרים. אנו מציעים לאפשר למתכנת לקבוע בכל סוכן לעיבוד־מאורעות את אחד משלושת מהמדיניות הבאות: השמת זמן גילוי, השמת זמן התרחשות או השמת זמן סוף־חלון. המנגנון השני עוסק בסמנטיקת הכללת המאורעות של חלונות זמן. לבסוף, אנו מרחיבים על הדרך בה מהודר שומר־נכונות לקוד בשפה. בפרק 6, אנו מציגים אלגוריתם זמן־ריצה אשר משמש את שומרי הנכונות בפעולתם. פרק 7 מתאר את שפת TAL, שפה דקלרטיבית שביטוייה מבוססים על תחשיב הפסוקים, שבעזרתה ניתן להגדיר שומרי נכונות, מהם מהדר השפה מייצר קוד אשר משולב ביישום. לבסוף, בפרק 8 אנחנו מסכמים את מסקנותינו, ומציגים בעיות

נכונות נוספות שלא ניתן לפתור אותן בעזרת הגישה שלנו. בנוסף, אנו מציגים מהן ההרחבות הנוספות הנדרשות על־מנת לתמוך בשומרי נכונות לאותן בעיות.

התרומה העיקרית של התיזה היא לשמירת הנכונות ביישומים לעיבוד מאורעות. ראשית אנו מזהים את הטקסונומיה של בעיות נכונות, ושנית אנו מציעים פתרון לבעיות אלו. לאחר מכן הפתרון המוצע מוכלל לידי מנגנון בשם "שומר־נכונות". מכיוון שאנו מודעים לעובדה שלא כל בעיות הנכונות הטמפורליות נפתרות בעזרת מנגנוני הנכונות שהגדרנו, ישמנו את שפת TAL, בעזרתה ניתן להגדיר שומרי נכונות נוספים.

Bibliography

[1] Arvind Arasu, Shivnath Babu, and Jennifer Widom. The cql continuous query language: semantic foundations and query execution. *The VLDB Journal*, 15(2):121–142, June 2006.

[2] Gérard Berry and Georges Gonthier. The esterel synchronous programming language: Design, semantics, implementation. *Sci. Comput. Program.*, 19(2):87–152, 1992.

[3] Mani K. Chandy, Opher Etzion, and Rainer von Ammon. 10201 Executive Summary and Manifesto – Event Processing. In K. Mani Chandy, Opher Etzion, and Rainer von Ammon, editors, *Event Processing*, number 10201 in Dagstuhl Seminar Proceedings, Dagstuhl, Germany, 2011. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, Germany.

[4] Edsger W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Commun. ACM*, 18(8):453–457, 1975.

[5] J.-H. Hwang, M. Balazinska, A. Rasin, U. Cetintemel, M. Stonebraker, and S. Zdonik. High-availability algorithms for distributed stream processing. In *Data Engineering, 2005. ICDE 2005. Proceedings. 21st International Conference on*, pages 779 – 790, april 2005.

[6] Namit Jain, Shailendra Mishra, Anand Srinivasan, Johannes Gehrke, Jennifer Widom, Hari Balakrishnan, Ugur Çetintemel, Mitch Cherniack, Richard Tibbetts, and Stanley B. Zdonik. Towards a streaming sql standard. *PVLDB*, 1(2):1379–1390, 2008.

[7] Shawn R. Jeffery, Gustavo Alonso, Michael J. Franklin, Wei Hong, and Jennifer Widom. A pipelined framework for online cleaning of sensor data streams. Technical Report UCB/CSD-05-1413, EECS Department, University of California, Berkeley, Sep 2005.

[8] Jin Li, Kristin Tufte, Vladislav Shkapenyuk, Vassilis Papadimos, Theodore Johnson, and David Maier. Out-of-order processing: a new architecture for high-performance stream systems. *PVLDB*, 1(1):274–288, 2008.

[9] David C. Luckham. *The Power of Events: An Introduction to Complex Event Processing in Distributed Enterprise Systems*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001.

[10] Ashish Ray. Oracle data guard: Ensuring disaster recovery for the enterprise. an oracle white paper, March 2002.

[11] Wil M. P. van der Aalst, Niels Lohmann, Marcello La Rosa, and Jingxin Xu. Correctness ensuring process configuration: An approach based on partner synthesis. In *BPM*, pages 95–111, 2010.

[12] ventana research. Innovating with operational intelligence and complex event processing. Technical report, ventana research, 2011.