JOHANNES KEPLER
UNIVERSITY LINZ | JKU

Implementation of a Bitemporal Complex Event Processor (BiCEP) with H2 and a Compiler for its Language (BiCEPL)

Master's Thesis

to confer the academic degree of

Master of Science

in the Master's Program

Business Informatics

Author:
Felix Burgstaller BSc

Submission:
Data & Knowledge Engineering Institute

Thesis Supervisor:
o. Univ.-Prof. DI Dr. Michael Schrefl

Assistant Thesis Supervisor:
Mag. Michael Huemer

Linz, June 2014

Eidesstattliche Erklärung

Ich, Felix Burgstaller, erkläre an Eides statt, dass ich die vorliegende Masterarbeit selbstständig und ohne fremde Hilfe verfasst, andere als die angegebenen Quellen und Hilfsmittel nicht benutzt bzw. die wörtlich oder sinngemäß entnommenen Stellen als solche kenntlich gemacht habe. Die vorliegende Masterarbeit ist mit dem elektronisch übermittelten Textdokument identisch.

Signed: _____

Date: June 18, 2014

Acknowledgements

I am indebted to many people for their support and encouragement during the last year of my studies. Without them the successful completion of my master thesis would not have been possible. In the following paragraphs I would like to thank some of them. I am aware that there are many more and nothing I write here can express the gratitude I feel for them.

I am grateful to o. Univ.-Prof. DI Dr. Michael Schrefl who gave valuable feedback regarding my thesis and is supporting me in my striving for a scientific career. I would like to express my gratitude to my assistant thesis supervisor Mag. Michael Huemer for providing useful information, comments, remarks, and help whenever needed. Moreover, I would like to thank him for his guidance in writing scientific reports, papers and theses and for being an excellent discussion partner.

Last but by no means least, I would like to thank my family, my girlfriend and friends for their continued support and encouragement. I am particularly indebted to my parents who supported and financed my studies and my year abroad.

Abstract

Events and their announcements are ubiquitous today, especially in the World Wide Web. These are of interest as they allow to gain information and knowledge as well as to derive actions and reactions to take. To check all the event announcements manually is a cumbersome task. Fortunately it can be automated - the dissertation *Bitemporal Complex Event Processing of Web Event Advertisements* by Michael Huemer, 2014, focusses on Web events and introduces an approach for extracting events from the Web, processing them into complex ones and execute actions on their occurrence.

This thesis implements parts of Mr. Huemer's approach: It develops a compiler for the event and condition-action specification language Bitemporal Complex Event Processing Language (BiCEPL). Moreover a Complex Event Processor (CEP) based on in-memory database system H2 is implemented.

Zusammenfassung

Events und ihre Bekanntmachungen sind in der heutigen Zeit nahezu allgegenwärtig, vor allem im Web. Diese sind vorwiegend deswegen von großem Interesse da sich Informationen und Wissen sowie Aktionen und Reaktionen aus ihnen ableiten lassen. Allerdings ist die manuelle Überprüfung dieser Event-Bekanntmachungen beschwerlich. Glücklicherweise kann diese Aufgabe automatisiert werden - die Dissertation *Bitemporal Complex Event Processing of Web Event Advertisments* von Michael Huemer, 2014, konzentriert sich auf Web Events und führt eine Herangehensweise zum extrahieren von Events aus dem Web, deren Verarbeitung zu komplexen Events und die Ausführung von Aktionen bei deren Eintreten, ein.

Diese Arbeit implementiert Teile von Huemers Herangehensweise: Ein Compiler für die Sprache zur Event und Condition-Action Spezifikation, Bitemporal Complex Event Processing Language (BiCEPL), wird entwickelt. Weiters wird ein Complex Event Processor (CEP) basierend auf dem in-memory Datenbanksystem H2 implementiert.

Contents

Chapter 1

Introduction

Contents

This section gives an introduction to the topic of complex event processing, the thesis, its contributions, the running example used and finally describes the organisation of the thesis.

1.1 Preface

Events are ubiquitous today - you find them on the Internet, in businesses, at home, etc. So what is an event? Michelson (2006) describes an event as "a notable thing that happens [...]", Etzion (2010) as transitions between states of entities. Furthermore Etzion (2010) states that the term event also refers to the representation of events in the computer domain. Moreover, "Event" can mean event occurrence as well as event specification (Michelson, 2006). Each event has at least a timestamp; Michelson (2006) describes events as only having one timestamp, Etzion (2010) delineates occurrence time, when an event occurred in the domain; detection time, the time an event is detected by an event processing system; and valid time, the time within which an event is relevant for processing. Besides these timestamps each event conveys information about what happened (Michelson, 2006).

Such events provide lots of data which have to be processed in order to gain information and knowledge from them and/or to choose appropriate actions and reactions respectively. Systems providing the means to do so are so called event processing systems (or information flow processing applications (Cugola & Margara, 2012)). Such systems consist of several modules (Etzion, 2010; Michelson, 2006):

Event Generators/Detectors Event generators or detectors emit or detect events, respectively.

Event Channel Events emitted or detected in the previous module are forwarded into an event channel, e.g., a message queue, which routes the events to the respective event processing engine(s). This channel is also used to transport the results of the processing to the event consumers.

Event Processing Engine The event processing engine processes the events and forwards the results into the event channel(s) and thus to the event consumer(s).

For such a system Michelson (2006) delineates three different kinds of event processing styles:

Simple Event Processing The most primitive form of event processing; a notable event occurs and triggers some action(s) (Michelson, 2006).

Stream Event Processing In stream event processing multiple input streams, possibly from different sources, are processed. These input streams are screened for notable events and event content and sub-

sequently are routed to the corresponding output stream. Assumed is that events of the input stream are processed in order of their arrival, thus simplifying the processing algorithm as only little memory is needed and no event occurrences have to be remembered (Luckham, 2006). (Cugola & Margara, 2012; Michelson, 2006)

Complex Event Processing In complex event processing the assumption that events arrive in order is not made; they take all events that were detected so far into account. Furthermore these systems have a different scope than stream event processing. They find patterns of events which form higher-level/complex events. These patterns have to be specified beforehand. Additionally to detecting such complex events interested parties are informed about them. Moreover complex event processing systems can also be used to simplify and abstract information in event patterns. (Cugola & Margara, 2012; Luckham, 2006; Michelson, 2006)

The event processing systems described in Etzion (2010) and Huemer (2014) are complex event processing systems which make use of the modules described above. Both utilize occurrence, detection as well as valid time albeit in different ways. Etzion (2010) specifies two processing methods: detection time semantics, events are ordered by the time detected, or occurrence time semantics, the order of events is given by their occurrence time. The order of events is important, for instance, for trend patterns in data. Only one of the semantics can be chosen. Huemer (2014) on the other hand concentrates on events from the Web. Such events are often changed and due to the nature of the Web updates or publications of events can be delayed. These situations can trigger actions in Huemer (2014), for instance, informing the event consumer. To address the various situations a novel bitemporal event model (occurrence and detection time) for the complex event processing framework Processing Event Ads into Complex Events (PeaCE) is introduced. The possibility to define conditions and actions upon events is similar to active databases and event condition action (ECA) rules (Huemer, 2014).

This thesis is based on the PeaCE framework introduced in Huemer (2014). In particular a compiler for the event specification language Bitemporal Complex Event Processing Language (BiCEPL) is developed as well as an event processing engine based on H2. Section 1.2 describes the problems addressed in this thesis in more detail. The subsequent Section 1.3 introduces the running example used throughout the thesis for illustration. Finally, Section 1.4 delineates the further organization of the thesis.

1.2 Statement of Problems

Today countless events are posted on the Internet at any given point in time - processing these masses is rather challenging. A field coping with processing and utilizing these masses is (complex) event processing. Huemer (2014) proposes a complex event processing framework which consists of event detectors, extracting events from the Web; at least one Complex Event Processor (CEP), processing these events and deducing their corresponding actions; and action executors conducting these actions.

The objective of this thesis is to extend the Java prototype of the PeaCE framework described in Huemer (2014). In particular this means

- implementation of a compiler for BiCEPL, the language used to define event and action classes and conditions for these actions,
- as well as providing an H2 implementation of the Complex Event Processor (CEP) of the PeaCE framework.
- Additionally, these implementations are explained by introducing a running example which gives the reader a better understanding of the framework and the implementations.

The main contributions of this thesis do not consider event detection nor action execution. They solely regard the CEP. Furthermore, any (visual) assistance regarding BiCEPL, except for the console error output, is out of scope. Finally, it has to be noted that the Java implementation of the PeaCE framework is a prototype, thus is by no means complete nor fully tested.

1.3 Buying Groceries as Running Example

This chapter describes the running example used throughout this thesis to explain different parts of the PeaCE framework. The scenario plays in the future when products for daily use are able to communicate with each other. This communication can take part over the Web (Internet of Things) or any other form of network (eg. Bluetooth, Radio frequency Identification (RFID),...) but is not part of this thesis.

In this future one can abolish to some extent the, by some considered tedious, task of shopping groceries by employing the PeaCE framework. First, the

traditional process needs to be analysed, and subsequently the process, as implemented using the PeaCE framework, is described. This example focuses on products which need to be refrigerated.

Today's process Today we have to check the refrigerator for products which need to be replenished, either because they are empty, almost empty, or expire soon. Once we have done this we decide on a day to buy groceries based on our free time and the urgency of shopping. Subsequently we go to one or several shops and buy what we need, pay and bring the groceries home and place them in the fridge. This process is then repeated in more or less regular time intervals.

Suggested Process In this process the fridge is informed by the products it contains about their status. Once a resource has to be replenished the fridge orders the appropriate product, or carries out some other operation. The fridge is then notified about the deliveries and can react to any changes to them.

The example focuses on a certain area of the scenario described above which is delineated here. The resources inform the refrigerator if they are getting low, are already empty, or expire soon. The refrigerator then orders products from which it got status messages. Depending on whether these messages are received late, i.e., detected later than they were issued, for instance, due to network latencies, and whether a resource is already empty an instant replenishing need or a normal replenishing need arises. The fridge will order products for normal replenishing needs, depending on the detection delay, with normal or one-day delivery. If an instant replenishing need occurs the refrigerator informs the owner. Once it has ordered a product the fridge is informed about the delivery and any changes, e.g., a delay. When a delivery is late the fridge informs its owner, when it is early a twitter message is posted.

1.4 Outline

With the introduction given above the remaining thesis is organized as described here. First an introduction to the basics is given in Section 2. Afterwards the contributions, the BiCEPL compiler and the H2 version of the CEP, to the PeaCE framework are delineated in Section 3 and Section 4, respectively. The thesis ends with the conclusion in Section 5. The sections are described in the following paragraphs in more detail.

The first section presents the basis for this thesis - the PeaCE framework (Section 2). In this chapter an introduction to the PeaCE framework by Huemer (2014) is given (Section 2.1). Subsequently, the underlying novel bitemporal event model is explained in Section 2.2. Thereafter, the language for specifying event classes, conditions-action statements, and action classes, BiCEPL, is delineated in Section 2.4. Section 2.5 explains the semantics of the PeaCE framework. Finally, the Java prototype of the framework is introduced in Section 2.6.

Section 3 delineates the first contribution to the framework, namely the implementation of a compiler for BiCEPL. To do so an introduction to compilers (Section 3.1) and compiler compilers (Section 3.2) is given. The following section (Section 3.3) describes the syntax implemented and changes made to the syntax introduced in Huemer (2014). Subsequently the Java implementation is sketched in Section 3.4.

The second contribution to the framework is described in detail in Section 4. This section copes with extending the framework by an H2 based CEP. First the H2 database and its qualities are described in Section 4.2 before explaining the implementation approach for the CEP using an H2 database (Section 4.1). Section 4.4 depicts the differences to the SQLite implementation. Sections 4.5 and 4.6 describe the performance improvements achieved and the test method used, respectively.

Finally, Section 5 summarizes the thesis and gives an outlook on future work.

Chapter 2

The PeaCE Framework by Huemer (2014)

Contents

The PeaCE framework introduced in Huemer (2014) is summarized in this section. Therefore a short introduction is given here as well as some fundamentals are investigated in more detail as they are needed to understand the implementations. Finally, the Java prototype, a concrete implementation of the PeaCE framework is introduced so the reader can understand the practical part of this thesis which extends this prototype.

2.1 A Sketch of PeaCE

The Web is an excellent source for events; innumerable events are posted each moment. It is desired to utilize these events to determine how to react/act upon them, i.e., which actions to execute on their occurrence. Utilizing these masses of events is insurmountable for humans, especially as soon as tasks require combinations of events increasing complexity. Issues for humans arise for instance when checking all sources whether new events have occurred (e.g. time consuming and error-prone). Furthermore, deducing actions by hand from all events found is prone to mistakes. Another problem when working with events is that not just an event can be late but also its detection, thus further complicating the situation. Such a process is desired to be automated; capable of doing so is the event processing framework PeaCE which, with its novel bitemporal event model, allows to react, for instance, to late detections. (Huemer, 2014)

The previous paragraph implies that such an event processing framework consists of several parts:

Event Detectors Event detectors are components which retrieve events from various (web) sources and pre-process them into a form which can be used by CEP components. PeaCE uses OXPath (Sellers, 2011) to extract events from the Web.

CEP CEPs process events and publish actions deduced. Which events are input and which actions when to take has to be defined beforehand. In PeaCE this is defined with a language called BiCEPL.

Action Executor Action executors conduct actions derived by CEPs. Such actions can be, for instance, writing an email, ordering a product, or posting on Facebook.

The components described form an application instance of the PeaCE framework. Probably the most important part is the definition of events and actions. Each type of event is defined in a separate class with its attributes, key attributes, lifespan, and optional condition-action statements to trigger actions based upon that event class. Such a definition is also called the schema of an event. Three basic types of event classes are available, namely, subscribed event, complex event, and action classes. The instances of the first are created by event detectors, the second are computed as Structured Query Language (SQL) queries over other subscribed and complex event classes. An condition-action statement defines the condition under which an

instance of the corresponding action class is created. Conditions can refer to the previous and current version of an event as well as various timing primitives (Section 2.2) and attribute/value comparisons (Section 2.4). The schema of an action class is determined by the action part of the statement, that is a list of attributes to be used from the event class it is defined upon.

The following chapters describe the PeaCE framework as defined in Huemer (2014). Section 2.2 describes the bitemporal event model on which PeaCE is based. Thereafter, the process of creating a concrete instance of PeaCE is depicted in Section 2.3. The language, BiCEPL, used to define events and actions is delineated in Section 2.4. Subsequently the event processing semantics of the PeaCE framework are investigated (Section 2.5) and finally the Java implementation of the framework is described in Section 2.6.

2.2 Bitemporal Event Model

PeaCE's novel feature is its bitemporal event model. This bitemporal event model is necessary as events can not only be late, their announcements might be late too and moreover events can change over time. Therefore Huemer (2014) defines an *occurrence time*, the time at which an event actually occurs, and a *detection time*, the time at which an event is detected, which not necessarily equals the occurrence time, for each event (refer to Example 2.1 for an example). These two kinds of times are used to define various timing primitives which can then be used as predicates in BiCEPL condition-action statements to trigger actions (Section 2.4).

Example 2.1: The Bitemporal Model

Consider the resources in the running example. The instant these resources notice that they are running low is the occurrence time of the event (ResourceLow). The detection time is when the refrigerator reads the message from the resource. If the resource is, for instance, because of a network outage unable to post the event right away, then the occurrence and detection time will be different. The detection time is then the moment the fridge reads the event after the network outage has been rectified.

The timing primitives presented in Figure 2.1 can be grouped into `ANNOUNCEMENT`, `CANCELLATION`, and `CHANGE`. The former refers to the initial occurrence of an event whereas the latter two groups consider events which

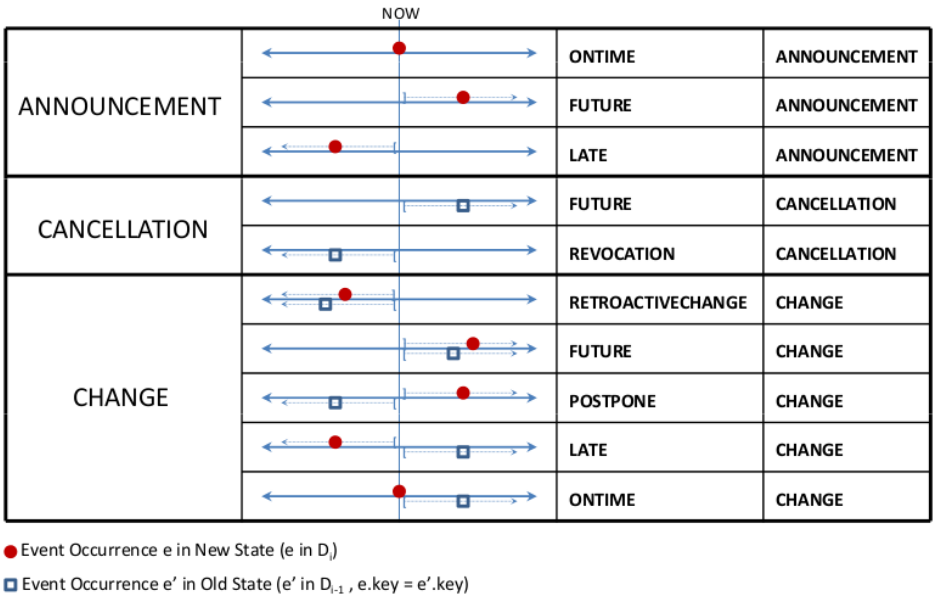


Figure 2.1: The timing primitives for event classes (Huemer, 2014)

already exist. The timing primitive `ONTIME` is for events happening on time, `FUTURE` describes event which will occur and `LATE` refers to events being detected too late, i.e., they have already happened before the event was detected. The remaining timing primitives are for events which allegedly occurred in the past and are modified now. `RETROACTIVECHANGE` describes the changing of an event with its new occurrence time still in the past whereas `POSTPONE` describes events being postponed to the future. Finally, `REVOCATION` is for events which are cancelled, thus do not have an occurrence time after the update. Examples are given in Example 2.2.

Example 2.2: Timing Primitives

Assume a resource is issuing an event (for the first time) that it is running low at 5pm. The refrigerator detects the event instantly, thus also at 5pm. Consequently the timing primitives `ANNOUNCEMENT` and `ONTIME` are matched. The fridge then orders the resource. The delivery company posts the event for the delivery at 5:02pm but due to a network outage the fridge detects the event at 5:30pm. Since the occurrence time of the delivery event describes the time the delivery will occur the timing primitives `ANNOUNCEMENT` and `FUTURE` are matched in this case. Unfortunately issues in the supply chain delay the delivery. The delivery company publishes the delivery event with the

updated delayed delivery time, thus the timing primitives `CHANGE` and `FUTURE` match. The delivery time is changed once more after the delivery should have occurred due to more severe issues in the supply chain than expected, ergo `POSTPONE` matches.

2.3 PeaCE Setup Routine

Once the scenario is known, i.e., the actions to be executed, their conditions and basic events, four tasks need to be done in order to create a PeaCE application:

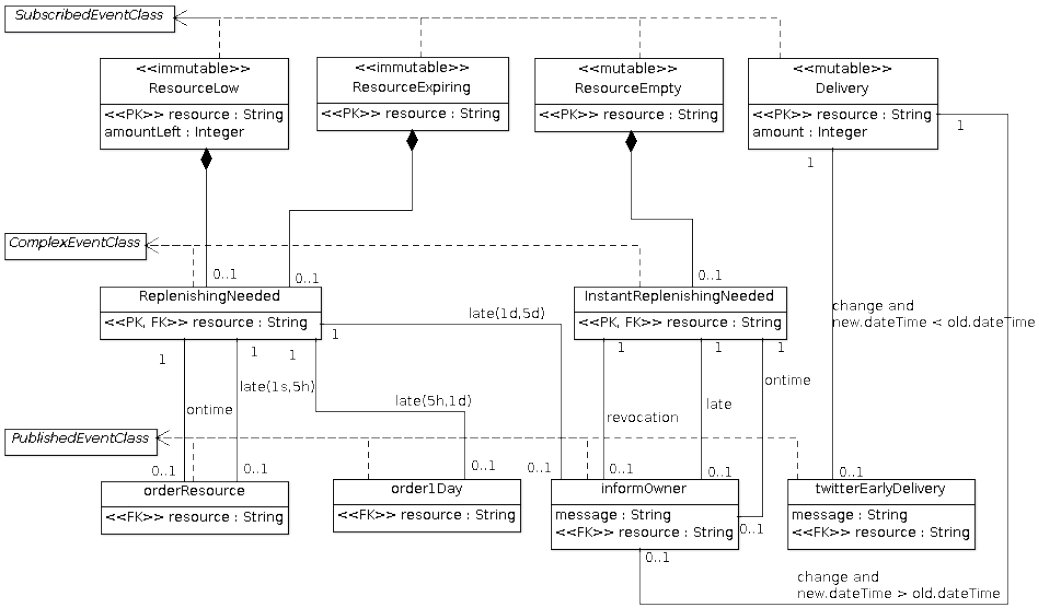
1. *Event wrappers* for extracting events from the Web (using OXPath) or any other source need to be defined.
2. *Action wrappers* for conducting actions based on event conditions need to be described. These actions can be web actions, for instance, posting on Facebook, or any other action which is implemented as an action wrapper.
3. Once the starting point, desired outcome, and actions are known the *subscribed event classes* (events of these classes are outcomes of event wrappers) need to be defined in BiCEPL. For each subscribed event class optional *condition-action-statements* can be declared which represent the connection to action wrappers. They define conditions on attributes of an event class which, if satisfied, trigger the action specified.
4. For complex scenarios subscribed event classes are not sufficient, therefore complex event classes are introduced to enable combinations of subscribed and other complex event classes. Analogously to subscribed event classes these are defined in BiCEPL and can have *condition-action statements*.

Example 2.3 shows the result of the process for the running example in UML notation.

Example 2.3: The Event Classes for the Scenario

Figure 2.2 shows the different event classes for the scenario defined in Section 1.3. The events are specified in UML here, extracts of the BiCEPL

definition are shown in Section 2.4.



≪PK≫ ...	Primary Key
≪FK≫ ...	Foreign Key
≪immutable≫ ...	Events of this class cannot change.
≪mutable≫ ...	Events of this class can change.

Figure 2.2: The running example event classes represented in UML.

2.4 Bitemporal Complex Event Processing Language (BiCEPL)

BiCEPL is the language used by the PeaCE framework to define how events are to be processed and which actions to perform under which conditions. A BiCEPL program contains all information necessary for a runnable CEP, i.e., the schema of all subscribed and complex event classes as well as their corresponding condition-action statements.

Section 2.4.1 describes the syntax for defining event classes whereas Section 2.4.2 delineates the notation of condition-action statements and their corresponding action.

2.4.1 Event Definition

In this section event definitions using BICEPL are delineated. Therefore BICEPL's syntax for defining an event class is depicted in Extended Backus-Naur Form (EBNF) in Grammar 2.1. A BICEPL program ⟨program⟩ is a sequence of event classes. These can either be subscribed ⟨sclass⟩ or complex event classes ⟨cclass⟩. Each such event class defines a ⟨schema⟩ which describes its ⟨attributes⟩ and ⟨keys⟩ (a subset of its attributes). The keys are important as they are used for updating events; if a second event with the same key is detected it is an update. The lifespan of an event is declared as ⟨time_literal⟩ which is depicted in Grammar 2.2. Finally the condition-action statements ⟨condition_action⟩ can be defined optionally (Section 2.4.2).

```
⟨program⟩    ::= { ⟨sclass⟩ | ⟨cclass⟩ }

⟨sclass⟩     ::= 'CREATE' ('MUTABLE' | 'IMMUTABLE')
                 'SUBSCRIBED EVENT CLASS' ⟨schema⟩ 'LIFESPAN'
                 ⟨time_literal⟩ [ ⟨condition_action⟩ { ⟨condition_action⟩ }
                 ] ';'

⟨cclass⟩     ::= 'CREATE' 'COMPLEX EVENT CLASS' ⟨schema⟩ 'LIFESPAN'
                 ⟨time lit⟩ 'AS' ⟨selection⟩ [ ⟨condition_action⟩ {
                 ⟨condition_action⟩ } ] ';'

⟨schema⟩     ::= ⟨name⟩ '(' ⟨attributes⟩ ')' 'ID' '(' ⟨keys⟩ ')'

⟨attributes⟩ ::= ⟨name⟩ ⟨type⟩ { ',' ⟨name⟩ ⟨type⟩ }

⟨keys⟩       ::= ⟨name⟩ { ',' ⟨name⟩ }

⟨selection⟩  ::= 'SELECT' ⟨select_clauses⟩ 'OCCURRING AT' ⟨time⟩

⟨time⟩       ::= ⟨table_ref⟩ '.' ⟨name⟩ | ⟨time⟩ [ ( '+' | '-') ⟨time_literal⟩] |
                 ('MAX' | 'MIN' '(' ⟨time⟩ { ',' ⟨time⟩ } ')'
```

Grammar 2.1: The EBNF of the event class definition in BICEPL as sketched in Huemer (2014). ⟨select_clauses⟩ and ⟨table_ref⟩ refer to non-terminals of an SQL grammar.

Subscribed event classes can be MUTABLE, i.e., they can be changed once detected, or IMMUTABLE, i.e., their first occurrence is final and thus no further changes are possible. Complex event classes on the other hand do not specify this due to their dependence on their constituent event classes (the event classes on which they are based). Complex event classes are defined as SQL queries ⟨selection⟩ over other event classes where the ⟨select_clauses⟩ are borrowed from an SQL grammar. Since these event classes are derived they

do not have a logical occurrence time, it needs to be specified explicitly using `OCCURRING AT ⟨time⟩`. The `⟨time⟩` itself can be an attribute of the selection, an arithmetic combination with a time literal, or a min or max function of several `⟨time⟩` instances. A sample BiCEPL class definition for the running example is given in Example 2.4.

Example 2.4: Subscribed Event Class ResourceLow

Listing 2.1 depicts the BiCEPL statement declaring the immutable subscribed event class ResourceLow.

```
1 CREATE IMMUTABLE SUBSCRIBED EVENT CLASS ResourceLow
    (resource VARCHAR(30), amountLeft NUMERIC) ID
    (resource) LIFESPAN(5d);
```

Listing 2.1: The BiCEPL statement defining the subscribed event class ResourceLow.

ResourceLow has two attributes: the id of the resource, *resource*, and the amount left of it, *amountLeft*, where *resource* is the key attribute. The lifespan of event instances is defined with five days.

The `OCCURRING AT` clause is not supported by the standard SQL. Therefore it needs to be rewritten into an SQL statement. Huemer (2014) calls this "SQL-Query Rewriting". The query rewriting extends table references by ".occ", i.e., references to a table name (thus event class) are references to the occurrence time of this class. The `OCCURRING AT` clause is rewritten as an attribute into the SQL select statement of the complex event class. The selection of attributes is then extended by the "det" attribute, the detection time (with the value `NOW`). An example query rewriting is given in Example 2.5.

Example 2.5: SQL-Query Rewriting

The SQL-query rewriting is shown in Listing 2.2 and Listing 2.3. Listing 2.2 depicts the original BiCEPL SQL-query, Listing 2.3 shows the rewritten pure SQL-query.

```
1 ... SELECT resource FROM ResourceEmpty WHERE ...
  OCCURRING AT ResourceEmpty.occTime + 10m ...
```

Listing 2.2: The original BiCEPL SQL-query.

```
1
  ... SELECT ResourceEmpty.occTime + 600 AS occTime, NOW AS
      det, resource FROM ResourceEmpty...
```

Listing 2.3: The rewritten BiCEPL SQL-query.

2.4.2 Event-Condition-Action Model

Section 2.4.1 describes the part of the BiCEPL grammar for defining event classes. For each event class condition-action statements can be declared which will be explained in this section. Grammar 2.2 depicts the syntax for these statements in EBNF; the embedding into the event class definitions is shown in Grammar 2.1.

⟨condition_action⟩ ::= 'ON' ⟨cond⟩ 'DO' ⟨action⟩

⟨cond⟩ ::= ⟨atom⟩ | 'NOT' ⟨cond⟩ | ⟨cond⟩ 'AND' ⟨cond⟩ | ⟨condition⟩ 'OR' ⟨condition⟩

⟨atom⟩ ::= ⟨value⟩ ⟨predicate⟩ ⟨value⟩
| 'ANNOUNCEMENT' | 'CANCELLATION' | 'FUTURE' | 'CHANGE' | 'ONTIME' | 'LATE' | ⟨late⟩ | 'RETROACTIVECHANGE' | 'REVOCATION' | 'POSTPONE' | 'FIRED'

⟨value⟩ ::= ('OLD.' | 'NEW.') ⟨name⟩ | ⟨literal⟩ | 'NOW'

⟨action⟩ ::= ⟨name⟩ '(' ⟨value⟩ { ',' ⟨value⟩ }] ')'

⟨time_literal⟩ ::= ⟨integer⟩ ('d' | 'h' | 'm' | 's')

Grammar 2.2: The EBNF definition of condition-action statements in BiCEPL as defined in Huemer (2014).

The condition-action model distinguishes between subscribed/complex event classes, and action classes. The former are the classes upon which the conditions are defined, the latter describe the action classes for event instances which satisfy the conditions. Each condition-action statement starts with ON followed by the condition ⟨cond⟩, the terminal DO, and the action class ⟨action⟩. A condition can contain timing primitives ONTIME, LATE, etc. ; FIRED; value comparisons ⟨value⟩ ⟨predicate⟩ ⟨value⟩; and logical combinations of these ⟨cond⟩. FIRED is true if an action has been triggered for this event instance before. Furthermore, previous, OLD., and current, NEW., attributes of events can be referred to, as well as the current time NOW and

literals ⟨literal⟩. The action definition ⟨action⟩ is analogous to a method call. It has a name and an optional sequence of parameters which are restricted by ⟨value⟩. Finally the time literal ⟨time_literal⟩ is introduced for usability. It allows to specify time durations like days, hours, etc. so the user does not have to calculate the time span in seconds. An example is given in Example 2.6.

Example 2.6: Condition-Action Statements

This example shows such a condition-action statement to give an idea what it looks like. Listing 2.4 depicts the event class definition of *Delivery* for which two condition-action statements are declared. Since the event class is mutable the events can change. If the delivery date changes we want to inform the customer or tweet a message. Listing 2.4 triggers action *informOwner* if an event instance is changed and the new delivery date is later than the old one. Moreover the action *twitterEarlyDelivery* is set off when a change occurs and the delivery is earlier than first announced.

```
1 CREATE MUTABLE SUBSCRIBED EVENT CLASS Delivery (resource
    VARCHAR(30), amount NUMERIC)
  ID (resource) LIFESPAN(14d)
3 ON CHANGE AND new.occTime > old.occTime DO
    informOwner('delayed', new.resource,new.occTime),
  ON CHANGE AND new.occTime <= old.occTime DO
    twitterEarlyDelivery('Delivery company delivers %s
    amazingly fast!',new.resource);
```

Listing 2.4: The BiCEPL statement defining the subscribed event class Delivery.

2.5 Event Processing Semantics

The elements of the PeaCE framework have been described in the previous sections. This section investigates the semantics of the event processing core starting with the time concept used, simplifications applied, updating semantics of events, and evaluation of conditions as well as triggering actions. Finally, the semantics for event expiration is depicted. For the semantics of timing primitives refer to Section 2.1.

The PeaCE framework is embedded into the real world and is able to compare times of event instances. Therefore the occurrence time of events is usually given using the conventional time concept and not some arbitrary one. As a result the *wall clock time* is introduced in Huemer (2014), representing the time in the real world. The occurrence and detection times use the wall clock time metric and thus can be compared on this wall clock time scale and also with the current wall clock time (e.g. by using `NOW`).

Processing all events in real time is costly and most of the time not necessary (apart from the fact that one cannot process events in "real" time; no event would ever be on time if one measures exactly enough). Consequently, Huemer (2014) utilizes the *chronon* concept. Each chronon has a certain time span; time is represented by an uninterrupted sequence of chronons. All events that occur within a chronon are treated as if they had the same occurrence time. The instance one chronon ends and a new one begins is called *clock tick*. For an example refer to Example 2.7.

A BiCEPL program defines the subscribed and complex events to be processed in the PeaCE framework instance. Each distinct event (having different key values than all the other event instances of this event class) has an occurrence and a detection time. Based on the condition-action statements, thus the current wall clock time, detection and occurrence time, and possible changes to an event, actions are triggered. These changes require one event (one unique key) to occur in two different chronons. Once a condition-action statement for a specific event instance is satisfied this event has *fired*. This is depicted in Example 2.7.

Example 2.7: Basic Semantics of PeaCE

Consider the event history shown in Table 2.1. The chronon length for this example is given with 15 minutes with each clock tick occurring exactly on the quarter hour. This means that at the clock ticks the following events are detected:

16:00 No event is detected yet.

16:15 The first event is detected and the expected delivery is on 7.4.14 at 9:00.

16:30 A second event is detected which has the same key (resource = milk) as the previous one. Furthermore the occurrence time attribute changed, thus this is a change of the previous event updating the delivery time. Since the condition *change and new.occTime* > *old.occTime*

holds for this update the action *informOwner* is triggered and thus fired.

Occurrence Time	Detection Time	Resource	Amount
7.4.14 09:00	3.4.14 16:01	Milk	2
7.4.14 17:00	3.4.14 16:27	Milk	2

Table 2.1: Event history for the subscribed event class Delivery.

The most natural semantics for such a system is the unlimited *buffering semantics*. This allows to compare all events of all time points with each other but can hardly be realized for systems with a high event throughput. Hence Huemer (2014) introduces the *sliding windowing semantics* to PeaCE. Each event class has a lifespan. This lifespan is used to calculate the expiration times of its instances; once an event expires it is purged from the system. For subscribed events which do not have any dependent event classes the expiration time is yielded by $exp = occTime + lifespan$. Complex events and subscribed events with dependent event classes on the other hand are more complicated resulting in the following (for an example refer to Example 2.8):

- The expiration times of complex event classes must be processed in a fixed order such that complex event classes which depend on other event classes are processed after these (analogous to stratification in mathematical logics).
- The preliminary expiration times of each complex event is then calculated as $prelmExp = MAX(occTime + lifespan, S)$ with S being the set of the occurrence time plus lifespan of all constituent events. These times are preliminary as they can be changed in the next step.
- Thereafter the expiration times of each constituent event is calculated as $exp = MAX(occTime + lifespan, S)$ with S being the set of the expiration times of all dependent events.
- All preliminary expirations times not changed in the last step are set as expiration times ($exp = prelmExp$).

Once every event has an expiration time associated each event expired, i.e., its expiration time is smaller than the current chronon start time (clock tick), is purged from the system and can be archived optionally. One sanity condition for the PeaCE framework is that events are not updated beyond

their lifespan. Furthermore, a complex event may only be purged if all its constituent events are purged, and vice versa.

Example 2.8: Purging Semantics of PeaCE

The expiration time calculation is shown here using the event histories depicted in Table 2.2.

ResourceEmpty (lifespan 3 days)

Occurrence Time	Detection Time	Resource
9.4.14 9:00	9.4.14 9:30	Milk

InstantReplenishingNeeded (lifespan 5 days)

Occurrence Time	Detection Time	Resource
9.4.14 9:00	9.4.14 9:30	Milk

Table 2.2: Event histories for ResourceEmpty and InstantReplenishingNeeded.

For complex event *InstantReplenishingNeeded* its constituent event *ResourceEmpty* needs to be taken into account. The expiration time then calculates as the maximum of its occurrence time plus lifespan and occurrence time plus lifespan of the *ResourceEmpty* event. Consequently this leads to *MAX(9.4.14 9:00 + 5 days, 9.4.14 9:00 + 3 days)* thus resulting in 14.4.14 9:00.

For subscribed event *ResourceEmpty* the expiration time is now given by *MAX(9.4.14 9:00 + 3 days, 14.4.14 9:00)*, ergo the expiration time is 14.4.14 9:00. Note that the lifespan of the event is thus prolonged by two days.

2.6 Implementation of PeaCE

The modular design of the PeaCE framework described in Section 2.1 allows for high adaptability. For simple cases the framework might have one event detector, one CEP and one action executor. For more complex scenarios more instances of these components can be used. This architecture also allows to

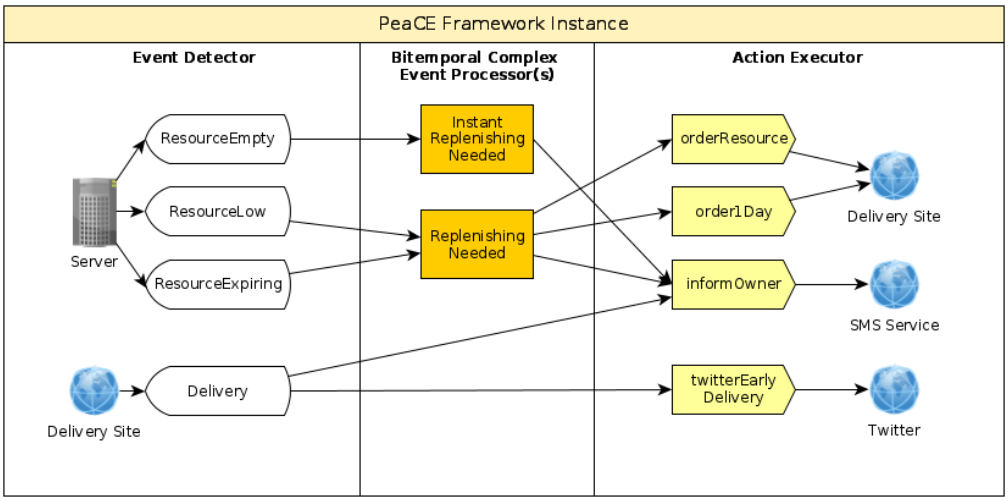


Figure 2.3: The PeaCE implementation for the running example.

distribute the components over different nodes, e.g., the CEP might run on a different computer than the detectors. Moreover these three components run in parallel thus utilizing multi-core processors when run on one computer.

Each event detector can supply multiple CEPs, each CEP can receive events from multiple detectors. Finally, action executors wait for actions from one or multiple sources and conduct the actions in the order they arrive. For the implementation of the running example with PeaCE refer to Example 2.9.

Example 2.9: Implementation of the Running Example

Figure 2.3 depicts the implementation and deployment of the running example. In this scenario four event detectors are employed, three of them, namely *ResourceEmpty*, *ResourceLow*, and *ResourceExpiring* access a server (via remote procedure calls) and one, *Delivery*, accesses the Web using OXPath. The complex events *InstantReplenishingNeeded* and *ReplenishingNeeded* are derived in the CEP. These two complex event classes could also be derived on two different CEPs. Finally, four action executors, viz. *orderResource*, *order1Day*, *informOwner*, and *twitterEarlyDelivery* are used to carry out the actions. All of them use OXPath to access web sites and to conduct their corresponding actions.

The Java implementation of PeaCE implements these components and fulfils

most of the qualities (e.g. not all combinations of conditions are supported) described in the previous sections. Furthermore the Java prototype can be deployed on mobile devices due to its lightweight and portable implementation style. Currently the framework supports SQLite and H2 databases, both can be used server-less which is particularly helpful for mobile devices. Additionally PeaCE allows detectors and action executors to be outsourced and accessed via web services easing the computation load on devices with small computing power. To increase usability the PeaCE framework provides a visual editor as well as a simulation and visualization environment.

A PeaCE Framework Instance at Runtime Once the process described in the next section is completed a PeaCE framework instance has been created. In this framework instance each event detector retrieves its events from a specified data source. Usually each event detector is responsible for collecting events of one subscribed event class. These events are forwarded into a buffer which is connected to the CEP. Through this connection the CEP is able, at each clock tick, to consume the buffered subscribed events. These events are then processed, this means, possible complex events are derived and condition-action statements evaluated. If such a condition-action statement holds true the corresponding action class instance is written to a buffer connected to the action executors. These action executors read the actions from the buffer and execute them.

Building a PeaCE Application Using the PeaCE Framework As mentioned before, the Java prototype of PeaCE implements the system as described in Huemer (2014) with some constraints, for instance, not all possible conditions are supported. The following short description of the instantiation process gives an overview of how the framework works. Note that the components embedded into a handler are those which need to provide methods to stop, start and pause their threads. These methods are used for instance by the simulation environment to control the PeaCE framework.

1. First, the *knowledge base* is created from a BiCEPL program. To create it the chronon length, whether to purge or archive expired events as well as whether global occurrence time optimization (Section 4.1.4) is to be used need to be announced. The knowledge base then represents the BiCEPL program in abstract syntax in form of Java objects which are subsequently used to create the CEP. A CEP is a set of tables and triggers for a specific database which implements a BiCEPL program. Therefore this knowledge base is compiled for the chosen underlying

database, for instance, H2, using a *mapper* class. After this step the database is initialized.

2. Once the CEP is set up it needs to be controlled, i.e., events need to be forwarded, complex events derived, conditions evaluated and actions published at each clock tick. Thus the next step is to create the so called *Bitemporal Complex Event Processor (BiCEP) handler*. This handler has a database dependent *task manager* assigned which has the three tasks **(a)** *read*, **(b)** *execute* and **(c)** *publish* which are executed in this order at each clock tick. **(a)** is responsible for reading events from the detectors described in step 3 into the database created in step 1. Furthermore it triggers the deduction of complex events, the expiration time calculation and the purging. **(b)** executes tasks for an optimization technique, namely global occurrence time optimization, described in Section 4.1.4. Finally, **(c)** forwards the derived actions to action executors as defined. The BiCEP handler is responsible for controlling the task manager thread, for instance, starting, pausing and stopping it. This step leads to a fully functional CEP component which has yet to be assigned an input buffer for **(a)** and an output buffer for **(c)**.

3. As mentioned in the previous step the buffer assignments for the CEP are not performed yet. Hence, in this step the *event detectors* and their *buffers* are created. For each detector an event buffer has to be created which can operate in two modes: forward delta (only new/changed events are forwarded) and forward all events. Currently event detection buffers utilize an SQLite database. Each buffer is then embedded into a *buffer handler*. Once the buffer is created the event detectors are instantiated. An event detector is represented by an event detection task which is executed every x seconds (the detection time interval) returning a list of detected events. This detector is then again embedded into a *detector handler*.

4. Next the action executors and the corresponding buffers are instantiated. First a *buffer manager* is created, a thread checking whether there are events in the buffer and processing them if some exist. In the next step the *buffer handler* and the *buffer* are created; also the buffer manager is then embedded into the buffer handler. Subsequently, the manager is provided with tuples of action classes and *action executors*. Each action executor is responsible for the execution of a specific action.

5. Now that all components have been created, what is missing is connect-

ing them. In this step *event detector buffer handlers* are assigned to the *read task* (created in step 2) via the *BiCEP handler*. Afterwards the assignment of the *action executor buffer handlers* to the corresponding *action classes* is done via the *BiCEP handler*, in particular the publish task (see step 2). Thus the event detectors and action executors are now connected to the CEP.

6. The last step is to assign all created components - event detection handler, BiCEP handler, and action execution buffer handler - to the *BiCEP model*. The model allows to control the PeaCE framework by providing methods to start, stop, and pause it. This is especially important for the simulation and visualization of the system.

Chapter 3

BiCEPL Compiler

Contents

This section copes with the implementation of a compiler for BiCEPL programs. Before diving into implementation details an introduction to compilers and compiler compilers is given in Section 3.1 and Section 3.2, respectively. The latter also describes the motivation for choosing ANother Tool for Language Recognition version 4 (ANTLRv4) as compiler compiler for the implementation. The subsequent section, Section 3.3, lists and explains the changes made to the grammar defined in Huemer (2014). Finally, Section 3.4, and its subsections give a detailed description of the Java implementation of the BiCEPL compiler.

3.1 Compilers

compiler.net (2010) defines a compiler as *"a computer program that translates a computer program written in one computer language (called the source language) into an equivalent program written in another computer language (called the output, object, or target language)"*. Most compilers translate a high level language into machine language. A parser (also syntax analyser) is part of a compiler; it checks the syntax of a program against a given grammar (the language of that program). Moreover, parser usually create a data structure, mostly parse trees, of the given program which can then be processed further. A grammar is a set of rules explaining the structure of that language. (Parr, 2012, p. 9f.)

Typical compilers comprise the following parts which are executed in the order given:

Lexer The lexer is responsible for grouping single characters into tokens. Furthermore the lexer can group tokens into token types, for instance integer. This step is also called lexical analysis or tokenizing. (Parr, 2012, p. 10)

Parser The parser considers tokens and token types found by the lexer and recognizes the sentence structure. Most parsers create a so called parse or syntax tree to record how it recognized a certain input sequence. The leaves of that tree represent the tokens, the nodes represent parsing rules, and the root is the most abstract rule of the input sequence. If there are any syntax errors in the program the parse tree cannot be constructed correctly, for instance, the root of the parse tree would not correspond to the root element of the grammar. Such syntax trees are easy to process in any following steps. (Mössenböck, 2007; Parr, 2012)

Semantic Analysis The semantic analysis can include, for instance, type checking, checks whether variables referred to in the program exist, etc. on the syntax tree generated by the parser. This step is optional, in the implementation of the BiCEPL parser this step is omitted. (Mössenböck, 2007)

Optimization This optional step optimizes the given input. This is often done when compiling high level languages, for instance Java, where given code is optimized to increase the performance of executables. (Mössenböck, 2007)

Code Generation The final task is the generation of code in the target

language. (Mössenböck, 2007)

The BiCEPL compiler will not comprise the two optional steps semantic analysis and optimization.

To write a compiler which comprises at least the three required steps is a time consuming, complicated task. Fortunately compiler compilers exist which allow to generate compilers and/or parsers from a formal description. Since the generation of a complete compiler is challenging most current compiler compilers create parsers and allow the user to add the code generation part.

3.2 Compiler Compilers

A compiler compiler is a tool generating a parser, interpreter or a compiler from a formal description like an EBNF. This is particularly useful in this case as the grammar of BiCEPL is given as EBNF. This section will state the qualities required from such a compiler compiler for the given task and why ANTLRv4 was chosen over other comparable tools.

First a few knock-out criteria are defined to constrain the possible alternatives for implementing a BiCEPL compiler:

- The compiler compiler must generate Java code in order to work with the existing PeaCE framework prototype.
- It must be free of charge.
- Extensions to the generated code must be possible.

Keeping the knock-out criteria in mind, four different compiler compilers, namely Coco/R (Mössenböck, Löberbauer, & Wöß, 2011), JavaCC (JavaCC, 2013b), SableCC (Gagnon, 1998) and ANTLRv4 (Parr, 2012, 2013), have been found. These are discussed in more detail below. In Table 3.1 a comparison of the compiler compilers named is depicted to support the decision for ANTLRv4.

Coco/R uses an attributed grammar (semantic actions), this means the code generation part is written directly into the formal description to add application specific code (Mössenböck, 2007). This will get chaotic and confusing with increasing amount of code. Furthermore it gets quite difficult to get a look at the underlying grammar. Additionally, no implementation of the SQL (2003) standard has been available at the point of writing thus Coco/R was ruled out as an option.

Property	ANTLRv4	CocoR
Code	Grammar actions, Java visitor, Java listeners	Semantic actions
Ignore Case	✗	✓
Left Recursion	✓	✗
Parser Type	LL(*)	LL(k)
SQL available	✓	✗
License	Berkeley Software Distribution (BSD)	Extended GNU General Public License
Latest Update	June 2013	April 2013

LL Left to right and Leftmost deviation top-down parser for a subset of the context-free grammars. Usually it is given as LL(x) where x indicates the number of tokens it can look ahead; LL(*) describes an unlimited look ahead. (Lewis, 2014)

Table 3.1: Comparison of the compiler compilers ANTLRv4 and CocoR. (Mössenböck et al., 2011; Parr, 2012, 2013)

Property	JavaCC	SableCC
Code	BNF productions	Java visitor
Ignore Case	✓	✗
Left Recursion	✗	✓
Parser Type	LL(k)	LALR
SQL available	✓	✗
License	BSD	GNU Lesser General Public License (LPGL)
Latest Update	May 2013	November 2012

LALR Look Ahead Left to right and Rightmost deviation parser for specific grammars. A LALR grammar is one for which its parser does not contain conflicts. LALR is a simplification of a canonical LR parser needing less resources. (Johnson, 2008)

Table 3.1: Comparison of the compiler compilers continued with JavaCC and SableCC. (Gagnon, 1998; JavaCC, 2013a, 2013b)

JavaCC uses so called BNF productions, similar to Coco/R's semantic actions, for application code notation. This results once again in a difficult to read compiler. Although an SQL (2003) grammar is available JavaCC was not used as the documentations found were incomplete and the code gets confusing.

The remaining compiler compilers are SableCC and ANTLRv4. Both are in terms of grammar and action definition very similar although they are based on different parsing techniques. Both allow to separate parsing and code generation and create tree walkers and/or listeners which allow to write actions for certain parts of the grammar. Furthermore, SableCC and ANTLRv4 support left recursion. The final decision is ANTLRv4 as an SQL (2003) grammar is available, although only ANTLR 3. Additionally, many widely known projects, for instance, NetBeans' C++ parser or HQL, use ANTLRv4. Albeit only the select statement of the SQL (2003) grammar is needed for BiCEPL the grammar is so complex and vast that it would have cost more time to implement just the select statement in all its variants than to adapt the existing grammar. (Gagnon, 1998; Parr, 2013)

A typical ANTLRv4 grammar comprises two parts, namely lexer rules specifying the lexer and parser rules defining the parser. These two parts can

either be combined into one single file or split into two or more files. The ANTLRv4 compiler compiler then reads this file or these files. These represent the formal description of the language to build a parser/compiler for. The compiler compiler then outputs the Java classes necessary to parse a program of that language. Optionally a visitor or a listener class can be created, thus allowing the user to insert application code into the parser. These classes are utilized in this thesis to compile a BiCEPL program into the knowledge base format required by the PeaCE framework.

3.3 BiCEPL Syntax Implemented - Differences to Huemer (2014)

The basic EBNF defining BiCEPL is taken from Huemer (2014) and has been developed further by the author since. This section describes the additional non-terminal and terminal symbols which have been introduced. On the one hand these are the ones necessary to represent the further developments in the grammar; on the other hand these are extensions which are essential so the grammar and the SQL (2003) integration could be implemented with ANTLRv4. The EBNF used for the BiCEPL parser is depicted in Grammar 3.1 and Grammar 3.2. As can be seen BiCEPL's structure is reminiscent of SQL.

Grammar 3.1, Grammar 3.2, and the original EBNF as defined in Huemer (2014) differ slightly. The specific differences are listed below.

- The non-terminal ⟨name⟩ is replaced by ⟨Regular Identifier⟩ from the SQL (2003) grammar.
- The symbol ⟨type⟩, which allowed long, real and text, is replaced by ⟨predefined type⟩ from the SQL (2003) EBNF. This is necessary to provide consistency within the BiCEPL statements regarding the type definition as not all of these types are available in SQL (2003) under these names. During parsing these predefined types, ⟨predefined type⟩, are converted to the three basic types long, real and text supported by the PeaCE framework.
- ⟨table_ref⟩ '.' ⟨name⟩ is replaced by ⟨identifier chain⟩ from SQL (2003).
- The non-terminal ⟨condition⟩ is extended by two options to support conjunctive and disjunctive clauses in parenthesis, thus allowing more

```
⟨program⟩   ::= { ⟨sclass⟩ | ⟨cclass⟩ }

⟨sclass⟩    ::= 'CREATE' ('MUTABLE' | 'IMMUTABLE')
                'SUBSCRIBED EVENT CLASS' ⟨schema⟩ ⟨lifespan⟩ [ ⟨ca stmt⟩ {
                ⟨ca stmt⟩ } ] ';'

⟨cclass⟩    ::= 'CREATE' 'COMPLEX EVENT CLASS' ⟨schema⟩ ⟨lifespan⟩ 'AS'
                ⟨bicepl select⟩ [ ⟨ca stmt⟩ { ⟨ca stmt⟩ } ] ';'

⟨schema⟩    ::= ⟨Regular Identifier⟩ '(' ⟨attrs⟩ ')' 'ID' '(' ⟨keys⟩ ')'
                ⟨attrs⟩ ::= ⟨Regular Identifier⟩ ⟨predefined type⟩ ⟨keys⟩
                ::= ⟨Regular Identifier⟩ { ',' ⟨Regular Identifier⟩ }

⟨lifespan⟩  ::= 'LIFESPAN' '(' ⟨time lit⟩ ')'

⟨bicepl select⟩ ::= ⟨query specification⟩ ⟨occ at clause⟩

⟨occ at clause⟩ ::= 'OCCURRING AT' ⟨time⟩

⟨time⟩      ::= ⟨identifier chain⟩
              | ⟨time⟩ ( '+' | '-' ) ⟨time lit⟩
              | ( 'MAX' | 'MIN' ) '(' ⟨time⟩ { ',' ⟨time⟩ } ')'
```

Grammar 3.1: The BNF definition of BiCEPL. Non-terminal symbols, which are all lower case except for SQL (2003) symbols, are represented by parser rules in the ANTLR4 grammar; terminal symbols, which are all capital letters, are lexer rules. The ⟨query specification⟩, ⟨identifier chain⟩, ⟨Regular Identifier⟩, and ⟨predefined type⟩ are taken from SQL (2003).

```
⟨ca stmt⟩   ::= 'ON' ⟨condition⟩ 'DO' ⟨action⟩

⟨condition⟩ ::= ⟨atom⟩
              | 'NOT' ⟨condition⟩
              | ⟨condition⟩ 'AND' ⟨condition⟩
              | '(' ⟨condition⟩ 'AND' ⟨condition⟩ ')'
              | ⟨condition⟩ 'OR' ⟨condition⟩
              | '(' ⟨condition⟩ 'OR' ⟨condition⟩ ')'

⟨atom⟩      ::= ⟨timing⟩ | ⟨lit⟩ ⟨comp op⟩ ⟨lit⟩ | 'FIRED'

⟨timing⟩    ::= 'ANNOUNCEMENT' | 'CANCELLATION' | 'CHANGE' | 'ONTIME' |
                'LATE' | ⟨late⟩ | 'RETROACTIVECHANGE' | 'REVOCATION' |
                'POSTPONE'

⟨late⟩      ::= 'LATE' '(' ⟨min delay⟩ ',' ⟨max delay⟩ ')'

⟨lit⟩       ::= ⟨val expr⟩ | ⟨general literal⟩

⟨val expr⟩  ::= ⟨val term⟩ | ⟨val term⟩ '-' ⟨val term⟩ | ⟨val term⟩ '+'
                ⟨val term⟩

⟨val term⟩  ::= 'NOW' | ⟨NEWOLDCOL⟩ | ⟨val numeric⟩

⟨val numeric⟩ ::= ⟨Unsigned Float⟩ | ⟨Unsigned Integer⟩ | ⟨Signed Float⟩|
                  ⟨Signed Integer⟩ | ⟨time lit⟩

⟨action⟩    ::= ⟨Regular Identifier⟩ ⟨paramlist⟩

⟨paramlist⟩ ::= '(' [ ⟨param⟩ { ',' ⟨param⟩ } ] ')'

⟨param⟩     ::= ⟨value expression primary⟩ | ⟨NEWOLDCOL⟩

⟨time lit⟩  ::= ⟨Unsigned Integer⟩ ⟨TIME UNIT⟩

⟨NEWOLDCOL⟩ ::= ('NEW' | 'OLD') '.' ⟨Regular Identifier ⟩

⟨TIME UNIT⟩ ::= 'D' | 'H' | 'M' | 'S'
```

Grammar 3.2: The BNF definition of BICEPL condition-action statments. Non-terminal symbols, which are all lower case except for SQL (2003) symbols, are represented by parser rules in the ANTLR4 grammar; terminal symbols, which are all capital letters, are lexer rules. The ⟨comp op⟩, ⟨literal⟩ extended by ⟨unsigned numeric literal⟩ as an additional option, ⟨Unsigned Integer⟩, ⟨Unsigned Float⟩, ⟨Signed Integer⟩, ⟨Signed Float⟩, ⟨general literal⟩, and ⟨value expression primary⟩ are taken from SQL (2003).

complex conditions.

- The symbol ⟨predicate⟩ is replaced with ⟨comp op⟩ from the SQL (2003) EBNF.
- ⟨value⟩ is extended and renamed to ⟨lit⟩ to support addition and subtraction of numeric terms. These are implemented anew since the SQL (2003) non-terminal ⟨numeric value expression⟩ implements multiplication and division as well. Nevertheless, the non-terminal ⟨literal⟩ is replaced with ⟨general literal⟩ from the SQL (2003) since they are semantically identical.
- The timing primitive FUTURE is not implemented as it has not been defined yet at the time the compiler was written.

3.4 BiCEPL Compiler Implementation

With the theoretical concepts and the grammar to be implemented at hand, the actual implementation is discussed in this section. Before delineating the specifics a short introduction to the PeaCE components used, in particular their realization in the Java prototype, is given in Section 3.4.1. Thereafter the ANTLRv4 formal description of BiCEPL (Section 3.4.2 - 3.4.4), the parser generated, and the application code used (refer to Section 3.4.5 for the latter two) to create the necessary PeaCE framework objects (the knowledge base) are discussed. Note that the following sections will give just an overview of the Java implementation, for the detailed description of classes, methods, attributes, grammar rules, etc. refer to the files and the Javadoc respectively.

The BiCEPL compiler implemented for the PeaCE framework, and described below, is rather slow for compiling/parsing programs. This is mainly due to the vast SQL (2003) grammar. Fortunately, since the compilation of a BiCEPL program takes place before the framework is run, the performance of the compiler is not essential. Therefore no performance tweaks were attempted.

3.4.1 PeaCE Components Used

The BiCEPL compiler uses only one part of the PeaCE framework, namely the so called knowledge base which represents a BiCEPL program in abstract syntax in form of Java objects. The compiler therefore uses the classes

of biceps.bicep.knowledge_base and its subpackages. The structure of these packages, their classes and their tasks are explained below.

Approaching the elements top-down the KnowledgeBase is the root element. It contains all events, their names, attributes, keys, condition-actions, etc. Thus the KnowledgeBase corresponds to the whole BiCEPL program. Each event definition in BiCEPL is represented with either an object of SubscribedEventClass or ComplexEventClass. The Java classes currently support three data types for attributes, namely long, real, and text. Therefore types specified in BiCEPL have to be mapped to these three.

The previous paragraph described the BiCEPL event definition in Java; here the event-condition-action declaration for event classes is delineated. For each condition-action statement of an event an EventPublicationStatement is created. This EventPublicationStatement is referenced in its corresponding EventClass. It contains an object hierarchy representing the condition (Section 3.4.5 and Figure A.6) and a PublishedEventClass object representing the action. A PublishedEventClass is a construct used to implement action classes. For each condition-action statement one published event class is created unless the same action class is used several times, then one published event class is created for all of them. Such a class contains the same attributes as the event class its condition-action statement is defined upon. The name of the published event class is given by the condition-action statement; the attributes specified for an action in BiCEPL are currently ignored. At each clock tick the instances of the published event classes are sent into the buffer forwarding them to the action executors. A published event class is thus nothing else but the implementation of an action class.

3.4.2 Structured Query Language Grammar

As described previously, BiCEPL is not just very similar to SQL it also uses the SQL (2003) ⟨query specification⟩ inter alia to define complex event classes. Hence the SQL (2003) EBNF has to be implemented, at least partly, or an existing grammar has to be employed. Since the SQL (2003) grammar is extensive and the interdependence of the various symbols is high it is difficult to implement just one specific part of it. Therefore an existing SQL (2003) grammar provided by Godfrey (2011) is modified so it can be used with ANTLRv4. Furthermore this SQL grammar is extended by BiCEPL specific symbols. Since Godfrey (2011) is not guaranteed to be tested JUnit tests are utilized to test the ⟨query specification⟩ part of the grammar (details

```
lexer grammar BiCEPLLexerDef ;
//Keywords for subscribed events
MUTABLE : 'MUTABLE' | 'mutable' ;
IMMUTABLE : 'IMMUTABLE' | 'immutable' ;
SEVENT : 'SUBSCRIBED EVENT CLASS' | 'subscribed event
    class' ;
LIFESPAN : 'LIFESPAN' | 'lifespan' ;
ID : 'ID' | 'id' ;
```

Listing 3.1: The ANTLRv4 lexer definition of the keywords for subscribed events.

follow in Section 3.4.6). For the concrete modifications see the grammar files; modifications are accompanied by comments starting with "fb:".

3.4.3 BiCEPL Lexer

A lexer is responsible for tokenizing the input stream, i.e., it groups the characters into tokens (Parr, 2013). The parser then only uses these tokens for further processing. The lexer for BiCEPL described in this section has been derived from the EBNF shown in Grammar 3.1 and Grammar 3.2.

The lexer developed for the PeaCE framework comprises two lexers which are maintained in separate files. One is the SQL (2003) lexer, the other is the BiCEPL lexer. The former is not described in detail as it has not been developed as part of this thesis. Nevertheless, it has to be mentioned that the EBNF productions have been reordered to work with ANTLRv4 and BiCEPL. Additionally, the white spaces which are skipped in Godfrey (2011) are now sent to a hidden channel. This is necessary as the parsed query_specification needs to contain the white spaces so it can be passed without further processing to an SQL data base. Skipped tokens do not appear in the parse tree and therefore white spaces would have to be added to the query_specification again.

The lexer specific to BiCEPL contains the BiCEPL tokens. These are divided into several parts:

- the keywords for subscribed events (Listing 3.1, line one shows the statement of the grammar type and its name),
- the keywords for complex events (Listing 3.2),
- the keywords for the optional condition-action statements (Listing 3.3),

```
//Keywords for complex events
CEVENT : 'COMPLEX EVENT CLASS' | 'complex event class' ;
OCC_AT : 'OCCURRING AT' | 'occurring at' ;
CHK_AT : 'CHECKING AT' | 'checking at' ;
```

Listing 3.2: The ANTLRv4 lexer definition of the complex event keywords.

```
//Keywords for the optional condition action statment
DO : 'DO' | 'do' ;
T_ANN : 'announcement' | 'ANNOUNCEMENT' ;
T_CAN : 'CANCELLATION' | 'announcement' ;
T_CHG : 'CHANGE' | 'change' ;
T_ON : 'ONTIME' | 'ontime' ;
T_LATE : 'LATE' | 'late' ;
T_RET : 'RETROACTIVECHANGE' | 'retroactivechange' ;
T_REV : 'REVOCATION' | 'revocation' ;
T_POS : 'POSTPONE' | 'postphone' ;
NOW : 'NOW' | 'now' ;
FIRED : 'FIRED' | 'fired' ;
```

Listing 3.3: The ANTLRv4 lexer definition of condition-action statements.

- a type definition, namely, the definition of the time unit enumeration (Listing 3.4),
- and the identifiers for new and old column specifications (Listing 3.4).

The `time_lit` rule, which semantically belongs to the lexer, is defined in the parser (Listing 3.9). This is necessary as the visitors and listeners generated by ANTLRv4 only contain methods for parser rules. These methods are needed to convert the `time_lit` to seconds.

```
TIME_UNIT : [dhms] | [DHMS] ;
NEWOLDCOL : ( NEW | OLD ) Period Regular_Identifier ;
```

Listing 3.4: The ANTLRv4 lexer definition of the column identifier referring to the new or old value of an event as well as the definition of the time units.

3.4.4 BiCEPL Parser

A parser recognizes a specific language, i.e., the syntax of the defined language (Parr, 2012, p. 10). In this section the parser implemented for the PeaCE framework, the BiCEPL parser, is explained in detail. This parser is, as is the lexer, derived from the EBNF depicted in Grammar 3.1 and Grammar 3.2.

Analogous to the lexer the parser comprises two separate specifications: one concerning the SQL (2003) grammar and one regarding BiCEPL. The SQL (2003) parser has been modified so it can be used with ANTLRv4 and BiCEPL. These changes are listed below:

- The import statement has been removed, all imports are declared in the BiCEPL parser specification.
- The options declaration at the beginning of the file has been deleted as it is no longer needed in ANTLRv4.
- All embedded actions have been removed from the ANTLRv4 grammar since they were of no use to the BiCEPL parser.
- The rule `literal` has been extended to support the BiCEPL specific time literals.
- The token `Space` has been removed from all rules as there is no need to explicitly define it. White spaces are automatically taken care of by the lexer.
- The `general_value_specification` has been changed in order to support the predicate `like`.
- Several sections of the grammar have been removed as they caused errors during the parser generation and are not viable for the `query_specification` rule.

For the specific modifications refer to the file; all modifications are annotated with "fb:" and an description of the alteration.

With the SQL (2003) parser part described above the specific BiCEPL parser are investigated below. This parser contains all parser rules necessary to recognize BiCEPL programs. The specification for the parser is split into several parts:

- The declaration part states the name, imported grammars, and the `@header` option. This option allows the specification of Java code which

```
grammar BiCEPL ;

import BiCEPLLexerDef , sql2003Parser ;//, CommonLexer ;

@header {
  package biCEPLCompiler.parser ;
}
```

Listing 3.5: The declaration part of the ANTLRv4 parser for BiCEPL.

is inserted before each class definition (Listing 3.5).

- The syntax of a BiCEPL file or command which consists of zero or more simple event and/or complex event classes. Notice that the syntax differs only in one point, the complex event class contains a modified SQL statement which declares how it is derived, i.e, how it depends on its constituent event classes (Listing 3.6).
- The modified SQL statement which defines the derivation of a complex event (Listing 3.7).
- The definition of the optional condition-action statement can be seen in Listing 3.8. The condition-action statement is extended to support conditions in parenthesis and therefore more complex conditions. This is represented in Listing 3.8 by lines 7 and 9.

  As depicted in Listing 3.8 the different possible alternatives for condition, atom, and val_expr are labelled (# followed by the name). Labelled alternatives allow more precise events in the ANTLRv4 generated Java code. Normally ANTLRv4 generates one listener/visitor method per rule. Using labelled alternatives one method for each alternative is created. This is very useful as no additional rules have to be introduced in order to get the method granularity needed.

  The action part of the condition-action statement is defined in lines 31 to 33 in Listing 3.8. An action looks like a method call in Java or a similar programming language; the difference is that the allowed parameters are SQL expressions and references to old or new values of an event instance.

- The time literal, which semantically would belong to the lexer, is declared in the parser (Listing 3.9). This is necessary to get ANTLRv4 to generate listener and visitor methods for the time literal.

```
//The program contains subscribed and complex event
    definitions.
program : ( sclass | cclass ) * ;

//Event definitions
sclass : CREATE ( IMMUTABLE | MUTABLE ) SEVENT schema
    lifespan ( ca_stmt ( Comma ca_stmt ) * ) ? Semicolon ;

cclass : CREATE CEVENT schema ( lifespan ) ? AS
    bicepl_select  ( ca_stmt ( Comma ca_stmt ) * ) ?
    Semicolon ;

lifespan : LIFESPAN Left_Paren ( time_lit | '-1' )
    Right_Paren ;
//The schema for the events, both subscribed and complex.
schema : Regular_Identifier Left_Paren attrs Right_Paren
    ID Left_Paren keys Right_Paren ;
attrs : attr ( Comma attr ) * ;
attr : Regular_Identifier predefined_type ;
keys : Regular_Identifier ( Comma Regular_Identifier ) * ;
```

Listing 3.6: The definition of a BiCEPL program which can contain simple as well as complex events.

```
//The modified select statement for BiCEPL
bicepl_select : query_specification occ_at_clause ;
occ_at_clause : OCC_AT time ;

time : identifier_chain
  | time ( Plus_Sign | Minus_Sign ) time_lit
  | ( MAX | MIN ) Left_Paren time ( Comma time )*
     Right_Paren ;
```

Listing 3.7: The modified SQL statement used to define the derivation of a complex event class.

```
//The condition action statement extended to support
    parenthesis
ca_stmt : ON condition DO action ;
condition :
  atom                                                #
     atomCond
 | NOT condition                                      # notCond
 | condition AND condition                            # andCond
 | Left_Paren condition AND condition Right_Paren     # andCond
 | condition OR condition                             # orCond
 | Left_Paren condition OR condition Right_Paren      # orCond
  ;
atom :
  timing             #timingAtom
 | lit comp_op lit    #compAtom
 | FIRED              #firedAtom
  ;
timing : T_ANN | T_CAN | T_CHG | T_ON | T_LATE | late |
    T_RET | T_REV | T_POS ;
late : T_LATE Left_Paren time_lit Comma time_lit
    Right_Paren ;

//lit updated to support sum and substraction
lit : val_expr | general_literal ;

val_expr :
  val_term                       #valExprTerm
 | val_term Minus_Sign val_expr  #valExprMinus
 | val_term Plus_Sign val_expr   #valExprPlus
 ;
val_term : NOW | NEWOLDCOL | val_numeric ;
val_numeric : Unsigned_Float | Unsigned_Integer |
    Signed_Float | Signed_Integer | time_lit ;

//The action is written like a method call
action : Regular_Identifier paramlist ;
paramlist : Left_Paren ( param ( Comma param ) * ) ?
    Right_Paren ;
param : value_expression_primary | NEWOLDCOL ;
```

Listing 3.8: The grammar for the optional condition-action statement.

```
//Time Literal ; Semantically it belongs to the Lexer but
    listener methods are only generated for rules!
time_lit : Unsigned_Integer TIME_UNIT ;
```

Listing 3.9: The grammar for the time literal.

3.4.5 BiCEPL Application Code / Compiler

The previous sections, Section 3.4.3 and Section 3.4.4, discuss the definition of the BiCEPL parser. Using the ANTLRv4 library the corresponding Java parser is generated. This chapter describes the application code added to the generated parser to create the needed PeaCE object, namely the knowledge base. This knowledge base is then used by the PeaCE framework to create the CEP.

The BiCEPL compiler comprises four packages, three containing application code and one package for JUnit tests (Figure A.1). The `biCEPLCompiler` package comprises the interface `BiCEPLRuntime` and the implementing `BiCEPL` class which are used to run the compiler and the parser. Furthermore, it contains the `biCEPLParser` package, consisting of the ANTLRv4 generated classes, and the `biCEPLCompiler` package, comprising the application specific code. A more detailed description of the different packages and the changes made to the PeaCE framework is given in the sections below.

Runtime Interface This package provides an interface and a class implementing it. These can be used to run the BiCEPL compiler or the BiCEPL parser. The parsing function is provided to the visual editor included in the PeaCE framework. The interface `BiCEPLRuntime` is employed to introduce an abstraction layer to the compiler. Hence it is possible to exchange the ANTLRv4 parser for any other parser as long as the interface is implemented.

The class `BiCEPL` implements the interface `BiCEPLRuntime` and provides the concrete implementation for the ANTLRv4 grammar. The classes and their relationship are depicted in Figure A.2.

Parser The `parser` package of the BiCEPL compiler contains the classes generated by the ANTLRv4 library. These allow to parse a given BiCEPL program and provide classes to extend the functionality of the parser. The built classes are depicted in Figure A.3. The class `BiCEPLBaseVisitor` provides the means to process the BiCEPL program via a visitor pattern allowing more control over traversing the parsing tree. The `BiCEPLBaseListener` on the other hand provides no means to steer the traversing. It ”only” provides listener methods for entering and exiting parser rules; thus every node of the parsing tree is visited when using the listener.

Compiler This package contains the extensions to the ANTLRv4 generated parser. As described in Section 3.2 two options for extending the basic parser exist which are both used in the BiCEPL parser. This is necessary as a BiCEPL program can be split into two differently organized parts. These are:

- the event class definition, and
- the condition-action statement.

This difference in the underlying structure demands distinct processing. The event class definition requires the existence of entering and exiting methods, e.g. for the time literal, and therefore has to be processed with a listener. The condition-action statement on the other hand is based on a tree structure, at least the condition part, and thus a visitor is preferred. Furthermore, the objects generated by the two parts are different; the definition part generates `EventClass`es whereas the condition action part creates `EventPublicationStatement`s. Finally, the methods of the listener do not return objects and thus are not as suitable for building hierarchical structures as is the visitor. Although both parts could be implemented using just a listener or only a visitor this has not been done for the sake of code readability and reduced complexity. Furthermore the performance of the parser is not critical resulting in the decision for cleaner code. The class diagrams for the compiler can be seen in Figure A.4 and Figure A.5. Below the different classes and their functions are explained in more detail.

`TimeLiteral` This class is used to convert time literals into seconds. The conversion is provided via the static method `getSeconds`.

`BiCEPLPeACEListener` This class extends the ANTLRv4 generated listener and processes the event class definition part of a BiCEPL statement. The disadvantage of using the listener is that it runs through the whole program and not just the class definitions.

The `BiCEPLPeACEListener` is responsible for creating the `KnowledgeBase` containing the `EventClass` objects for all event classes defined in the BiCEPL program. In order to achieve this it has to perform the SQL-query rewriting. The time literal is rewritten in the entering method of `time_lit` using the `TokenStreamRewriter`, provided by the ANTLRv4 framework, and the `TimeLiteral` class. The rewriting of the occurring-at clauses is handled in the exit method of `occ_at_clause`. The `TokenStreamRewriter` is not used here as the rewritten SQL clause is directly stored in the `EventClass` object. The class diagram can be found in Figure A.3; for more information regard the Javadoc.

`BiCEPLPeACEVisitor` This class extends `BiCEPLBaseVisitor<Symbol>` to convert the condition-action statements into `PublishedEventClass`es. To do so it uses the elements defined in PeACE framework. The result is a hierarchy of elements with a `Symbol` object as root for the `Condition`. In combination with a `PublishedEventClass` object for the action these two form the corresponding `EventPublicationStatement` for a given condition-action statement. The `PublishedEventClass` simply contains the name of the action. For detailed information regarding the hierarchy created refer to the next section.

The ANTLRv4 library allows to add custom error listeners to a parser. The following classes, depicted in Figure A.3, build such a listener and are used to log errors while parsing.

`LexerError` This class is a subclass of `Exception` and contains information about a lexical error including the line number, character position, offending symbol, etc. A `toString` method is provided to print the information in a convenient way.

`BiCEPLErrorListener` The `BiCEPLErrorListener` extends the `BaseErrorListener` provided by the ANTLRv4 library. It therefore can be used instead of or additionally to the standard error listener while parsing a file or a string. Each time an error occurs it creates a new `LexerError` object and adds it to the internal error list. These error objects can then be accessed via a getter. Furthermore, a method is provided which returns whether errors have occurred during parsing.

Necessary Changes to the Knowledge Base Packages The condition part of the condition-action statement is represented as a hierarchy of classes in the PeACE framework. The old hierarchy structure was not suitable for ANTLRv4 and its visitor extension. The visitor class of an ANTLRv4 generated parser returns the same class for all methods, thus the whole hierarchy needs to have one common class/interface. Consequently the hierarchy has been remodelled and extended by the features added to BiCEPL. The resulting remodelled hierarchy is depicted in Figure A.6. As shown all objects directly or indirectly (through inheritance) implement the interface `Symbol`. Thus it is possible to construct the object representation of a BiCEPL condition subsetting the visitor of the ANTLRv4 generated parser.

3.4.6 Testing

Before implementing the parser the SQL grammar retrieved from Godfrey (2011) has to be tested. Therefore the first JUnit test, namely `TestSQL`, is written to do so; in particular the select statement which is used in the BiCEPL grammar is examined. Select statements containing joins, predicates, group by, subselects, having, complex columns and order by are tested. In order to avoid writing one test method for each select statement *Parameterized JUnit Tests* are used. These allow to define a `Collection` of objects (in this case Strings containing select statements) and subsequently run the specified test method for every object in it.

Once it has been ascertained that the SQL grammar provided by Godfrey (2011) works as expected the BiCEPL compiler is developed. Three JUnit 4 tests check the correctness of the compiler. These confirm the functionality of the original implementation and test the continued functionality after changes of the compiler/parser. The JUnit test `TestBiCEPL` examines the parsing capabilities of the parser created. Various BiCEPL statements to be run and whether to expect parsing errors or not are specified. Finally, tests are created for the `BiCEPLPeACEListener` part of the BiCEPL compiler as well as the `BiCEPLPeACEVisitor` part. These run several BiCEPL statements and test the created Java objects for their correctness.

These JUnit tests could be grouped into a test suite to run all of them by simply starting the test suite. For the BiCEPL compiler no suite is created as eclipse provides the functionality to run all JUnit tests within a project, thus making a test suite superfluous.

Chapter 4

Implementation of PeaCE's Complex Event Processor in H2

Contents

The Java prototype delineated in Section 2.6 uses an SQLite database for its CEP. The second contribution of this thesis is the implementation of the PeaCE framework's CEP using H2. The first part of this chapter describes

the implementation approach for the CEP with extended condition-action statement support (Section 4.1). This includes tables, views, and triggers which are used to create the basic functionality of the CEP. Thereafter, before diving into implementation details, an introduction to the H2 database and its qualities is given in Section 4.2. Subsequently, the implementation of the approach within the Java prototype using H2 is explained. Section 4.4 then explicitly states the differences between the H2 and the SQLite implementation of the CEP. In Section 4.5 the performance improvements achieved are discussed. Finally, the test methods applied are described in Section 4.6.

4.1 Implementation Approach

This section depicts the implementation approach developed for the CEP supporting higher complexity in condition-action statements. Section 4.1.1 describes the tables and views needed whereas Section 4.1.2 delineates the triggers defined upon them. The subsequent sections Section 4.1.3 and Section 4.1.4 describe the features purging and global occurrence time optimization, respectively. Note that <SEC> is a placeholder for Subscribed Event Class (SEC), <CEC> for Complex Event Class (CEC), <AC> for Action Class (AC), and <EC> for Event Class (EC) name.

Database tables store on the one hand information about the PeaCE instance, e.g., chronon length, event classes, etc. On the other hand each event class is implemented with tables and views. Each of these tables contains the instances of the corresponding event class. All subscribed events are implemented with two tables, one containing an as-of-now view and one containing the history of events. Complex event classes have additionally to these two a view which is used to derive the complex event class from its constituent event class(es). These tables and views are maintained by triggers. These triggers move entries from the as-of-now to the history tables and vice versa, maintain materialized views, purge events, derive complex events, evaluate condition-action statements and perform optimization strategies. An example of the cooperation between tables and triggers is shown in Example 4.5.

4.1.1 Tables & Views

This section depicts tables and views needed for a CEP. This includes system tables, and event class specific tables and views. The corresponding triggers

for maintaining the tables are described in Section 4.1.2. Finally, Example 4.1 exhibits some tables created for event classes of the running example.

System Tables The system tables contain data about the PeaCE framework instance; some of them are static tables (Table 4.1) others are modified during runtime (Table 4.2).

The static tables, depicted in Table 4.1, are described in more detail here. Table *S_SituationClass* stores *name*, *lifespan* and *metatype* of all event classes defined for a certain PeaCE framework instance. The *metatype* field is a reference to an *S_SituationClassType* entry. *sitId* is an auto-increment field which is the primary key.

Table *S_SituationClassType* defines three different metatypes of event classes which can exist in a PeaCE framework. These are

- subscribed event classes,
- complex event classes, and
- published event classes (desbribed in Section 3.4.1).

S_SituationClass, depicted in Table 4.1, contains all event classes and their corresponding information; *S_SituationClassAttribute* stores all attributes defined for each event class in *S_SituationClass*. The event class is referenced via the attribute *sitId*. For each attribute its name and whether it is part of the key is stored.

Table *S_System* stores hyperparameters of the specific PeaCE instance, for instance, the length of a chronon. Each parameter is stored with a text/name describing it (*parameter*) and the corresponding value (*value*).

S_Trigger contains the *name* and the *priority* of every trigger employed in the PeaCE framework instance. The priorities of triggers define the order in which they are executed; this is of mayor importance as the result of the CEP depends on this order.

The last system table depicted in Table 4.1 is *S_ProcessorLog*. This table is used to log events of the PeaCE framework instance. For each clock tick the begin and end time of the three tasks read, execute, and publish are inserted.

Each dynamic system table (Table 4.2) contains just one row. This row is updated at each clock tick but at different times, for instance, *S_UpdateHistory* is updated in the read task whereas *ClockTick* is in the execute task. Consequently the triggers depending on them are started at different times when

Table Name	**Attributes**	**Key**
S_SituationClass	sitID INT name VARCHAR lifespan INT metatype INT	sitId
S_SituationClassAttribute	sitID INT att_name VARCHAR isKey INT	sitId att_name
S_SituationClassType	typeID INT name VARCHAR	typeId
S_System	parameter VARCHAR value VARCHAR	parameter
S_Trigger	name VARCHAR priority INT	name
S_ProcessorLog	occTime INT timeStamp BIGINT state VARCHAR event VARCHAR	occTime state event

Table 4.1: The static system tables used in PeaCE framework.

processing events of the previous chronon (the events arriving in the current chronon are processed when the next clock tick occurs). *ActClockTick*, unlike *ClockTick* and *S_UpdateHistory*, does not set off any triggers. *ClockTick* causes the triggers for action classes which contain `ONTIME` or `LATE` in their conditions to fire whereas *S_UpdateHistory* starts the triggers for updating the tables of complex event classes.

Table Name	**Attributes**	**Key**
ActClockTick	occTime INT	occTime
ClockTick	occTime INT	occTime
S_UpdateHistory	occTime INT detTime INT	-

Table 4.2: The dynamic system tables used in the PeaCE framework.

Subscribed Event Classes For each subscribed event class two tables are created, namely <*SEC*> and *H_*<*SEC*> (depicted in Table 4.3). The former contains the as-of-now view of the subscribed event class <SEC>. Consequently, there is always at most one event with a certain key in the table; no second event with the same key can exist. Changes to this event lead to an update of the corresponding entry. The latter, *H_*<*SEC*>, is the history table of the subscribed event class. It contains all changes to any event of this class, i.e., the initial insert, updates, and deletion (represented by a *occTime* of -1). Entries of the history table are only deleted in case the purging option of the PeaCE framework is activated.

Both tables store occurrence time (*occTime*) and detection time (*detTime*) of event instances. Additionally, user defined attributes can be added by specifying them in the BiCEPL program. These user defined attributes are represented as <*user defined*> in Table 4.3.

At each clock tick new events are inserted into *H_*<*SEC*>; The trigger transporting these new events to <*SEC*> is depicted in Section 4.1.2.

Complex Event Classes A complex event class is defined upon other event classes by an SQL select statement. This statement is used to derive the complex event class from its constituent one(s). In the implementation approach this is done by creating a view <*CEC*> which employs this SQL statement (Table 4.4).

Table Name	Attributes	Key
<SEC>	occTime INT detTime INT <user defined>	<user defined>
H_<SEC>	occTime INT detTime INT <user defined>	detTime <user defined>

Table 4.3: The tables created for each subscribed event class.

View Name	Attributes	Key
<CEC>	user defined	-

Table 4.4: The view necessary for deriving the instances of a complex event class. This view is defined upon the SQL select in the event class' BICEPL definition.

Analogous to subscribed each complex event class has a corresponding history (*H_<CEC>*) and as-of-now table (*R_<CEC>*) containing concrete event instances (Table 4.5). The view *<CEC>* and the table *R_<CEC>* contain the same entries; the difference is that the latter is a materialized view and thus allows faster queries. How these tables are updated and synced with the view is described in Section 4.1.2.

Table Name	Attributes	Key
H_<CEC>	detTime user defined	detTime user defined
R_<CEC>	user defined	user defined

Table 4.5: The as-of-now and the history table created for a complex event class analogous to subscribed event classes.

Action Classes (Published Event Classes) For each event class condition-action statements can be defined. Every condition-action statement is depicted with one table unless the same action is used multiple times. This is only permitted when the same attributes are provided. In this case one table is created for an action not for every condition-action statement. Every published event class table contains the event instances

for which the condition(s) defined is/are satisfied. For the table description refer to Table 4.6.

Table Name	Attributes	Key
H_<AC>	based on <CEC>	based on <CEC>

Table 4.6: One table, as depicted above, is created for each distinct action class. For an action used multiple times in the BiCEPL program one table is created.

Example 4.1: H2 Tables & Views for the Scenario

This example shows what some of the event classes of the running example look like in an H2 database using the implementation approach described above. Listing 4.1 depicts the SQL statements to create the tables for ResourceLow, a subscribed, and ReplenishingNeeded, a complex event class. In this case purging with archiving is activated. The tables are created as delineated in the previous section.

4.1.2 Triggers

The previous section, Section 4.1.1, describes the tables of the implementation approach. These are managed and maintained by triggers and can themselves cause triggers to execute. These triggers are described in this section. Each trigger is assigned a priority as they need to be executed in a certain order for the CEP to work. Higher values equal a higher priority. Finally Example 4.2 gives an example of an action class trigger.

Subscribed Event Classes Section 4.1.1 describes the tables created for each subscribed event class. Each materialized view *<SEC>* is maintained by two triggers (Table 4.7). *t_MView<SEC>_insert* transfers new events and event updates to the materialized view whereas *t_MView<SEC>_delete* deletes entries from the materialized view *<SEC>* which have been revoked.

Complex Event Classes Identical to subscribed, complex event classes have a materialized as-of-now view and a history table. Subscribed event

```
1  --ResourceLow
   CREATE memory TABLE H_ResourceLow(occTime INT,detTime
       INT,resource VARCHAR,amountLeft INT, PRIMARY
       KEY(detTime, resource ));
3  CREATE memory TABLE ResourceLow(occTime INT,detTime
       INT,resource VARCHAR,amountLeft INT, PRIMARY
       KEY(resource ));
   --Purging ResourceLow
5  CREATE TABLE P_ResourceLow(resource VARCHAR, detTime
       INTEGER, exp INTEGER, PRIMARY KEY(resource ));
   CREATE memory TABLE A_ResourceLow(occTime INT,detTime
       INT,resource VARCHAR,amountLeft INT, PRIMARY
       KEY(detTime, resource ));
7
   --ReplenishingNeeded
9  CREATE view ReplenishingNeeded
     as SELECT tab.occTime as occTime, unixTime(NOW()) as
        dettime, resource
11   FROM (SELECT resource,occtime FROM ResourceLow UNION
        SELECT resource,occtime FROM ResourceExpiring
     where (occTime - unixTime(NOW())) < 432000) tab;
13 CREATE memory TABLE H_ReplenishingNeeded(occTime
       INT,detTime INT,resource VARCHAR, PRIMARY KEY(detTime,
       resource ));
   CREATE memory TABLE R_ReplenishingNeeded(occTime
       INT,detTime INT,resource VARCHAR, PRIMARY KEY(resource
       ));
15 --Purging ReplenishingNeeded
   CREATE TABLE P_ReplenishingNeeded(resource VARCHAR,
       detTime INTEGER, exp INTEGER, PRIMARY KEY(resource ));
17 CREATE memory TABLE A_ReplenishingNeeded(occTime
       INT,detTime INT,resource VARCHAR, PRIMARY KEY(detTime,
       resource ));
```

Listing 4.1: The H2 SQL statements for the event classes ResourceLow and ReplenishingNeeded of the scenario.

Trigger Name	Condition	Priority
t_MView<SEC>_insert	AFTER INSERT ON H_<SEC>	-10
t_MView<SEC>_delete	AFTER UPDATE ON clockTick	-100

Table 4.7: The triggers for a materialized view of a subscribed event class.

classes use two triggers to keep them up to date whereas complex event classes use only one trigger depicted in Table 4.8.

Trigger Name	Condition	Priority
t_H<CEC>_history	AFTER UPDATE ON S_UpdateHistory	-80

Table 4.8: The trigger for updating the history and as-of-now table of a complex event class.

Action Classes (Published Event Classes) For each action class at least one condition-action statement in the BiCEPL program exists. These conditions are transformed into SQL queries which are then used in the triggers depicted in Table 4.9 (for the transformation refer to Section 4.3.2). Depending on the main timing primitive (`ONTIME`, `LATE`, ...) the triggers have different priorities and different triggering conditions (note that only one timing primitive per condition-action statement is permitted). When an event occurs and one of the conditions is satisfied the event instance is inserted into the corresponding table (Section 4.1.1). In case that several condition-action statements refer to the same action only one published action class table is created. Nevertheless, for every condition-action statement one trigger is created.

Example 4.2: H2 Trigger Example

Listing 4.2 displays the H2 Java class trigger for the condition-action statement for an one-day delivery. This trigger is dynamically created at runtime. Lines 14–19 are part of the global occurrence time optimization whereas lines 20–21 represent the actual query for determining whether the condition is met. The trigger is then registered in the H2 database using Listing 4.3.

```
1  package biceps.bicep.knowledge_mapper.h2;

3  import java.sql.Connection;
   import java.sql.PreparedStatement;
5  import java.sql.ResultSet;
   import java.sql.SQLException;
7
   import org.h2.api.Trigger;
9
   public class T_order1day implements Trigger {
11
     public void fire(java.sql.Connection conn, Object[]
        oldRow, Object[] newRow) throws SQLException {
13     PreparedStatement prep = null;
       PreparedStatement potStmt =
          conn.prepareStatement("select * from
          s_potential_ClockTick where eventclass like
          'ReplenishingNeeded' and occTime <= ("+newRow[0]+"
          - 10)");
15     ResultSet rsPot = potStmt.executeQuery();
       long count=-1;
17     if(rsPot.next()) count = rsPot.getLong(1);
       rsPot.close();
19     if (count > 0){
         prep = conn.prepareStatement("merge into H_order1day
            (occTime ,detTime ,resource ) select a.occTime
            ,a.detTime ,a.resource  from R_ReplenishingNeeded
            a where ((a.dettime-(a.occTime +
            ((a.dettime-a.occTime)% 10 ))) >= 18000) and
            ((a.dettime-(a.occTime + ((a.dettime-a.occTime)%
            10 ))) <= 86400) and a.occTime <> -1 AND
            a.occTime <= (a.dettime - 10) AND  NOT EXISTS
            (select t1.* from (select * from
            H_ReplenishingNeeded where
            dettime!="+newRow[0]+") t1 left outer join
            (select * from H_ReplenishingNeeded where
            dettime!="+newRow[0]+") t2 ON (t1.resource =
            t2.resource AND t1.dettime < t2.dettime) where
            t2.occtime IS NULL AND t1.resource = a.resource
            AND  t1.occtime < ((select occTime from
            ActClockTick )-10))");
21       prep.execute();
       }
23     if(prep!=null) prep.close();
     }
25 }
```

Listing 4.2: The H2 SQL trigger method for the condition-action statement order one-day delivery.

Trigger	Condition	Prior.	Note
t_<AC>	AFTER UPDATE ON clockTick	-70	OnTime & Late
	AFTER INSERT ON H_<E>	-5	Announcement
	AFTER INSERT ON H_<E>	-40	Cancellation & Change
	BEFORE INSERT ON H_<E>	-30	Retroactive change
	BEFORE INSERT ON H_<E>	-80	Revocation & Postpone
	AFTER INSERT ON H_<E>	-80	(*)

(*) Conditions which do not contain any timing primitives.

Table 4.9: The action class triggers for the different timing primitives in the condition-action statements.

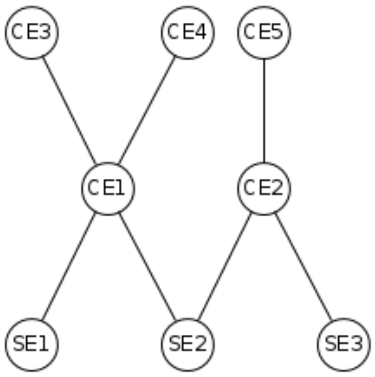
```
1 CREATE TRIGGER t_order1day AFTER UPDATE ON clocktick FOR
    EACH ROW CALL
    "biceps.bicep.knowledge_mapper.h2.T_order1day";
```

Listing 4.3: The H2 SQL statement to register the "order1day" trigger depicted in Listing 4.2.

4.1.3 Purging

The PeaCE framework implements its sliding windowing semantics through purging. Purging, when activated, deletes event instances after their defined lifetime has expired. This would be simple if it were not for complex event classes which depend on their constituent event classes. These dependencies form a graph with subscribed event classes at the bottom and complex event classes as internal nodes and at the top. The lifespan of a complex event class then depends on this graph. An example graph is shown in Figure 4.1; an example is given in Example 4.3.

Consequently the lifespans of complex events have to be calculated; Table 4.10 shows the tables used. The table *S_CalcExpTimes* contains one row which is updated at each clock tick and sets off the triggers for computing the expiration times of events. For each node (event class) a table



CE ... Complex Event Class
SE ... Subscribed Event Class

Figure 4.1: An example graph for purging.

P_<EC> (Table 4.10) is created. Each current event instance of the corresponding class is stored in this table with its key and its expiration time. The expiration time is calculated by traversing the graph from top to bottom and back. This computation of the expiration time is done by the triggers *t_P_Calc_Leaves* and *t_P_Calc_Roots* shown in Table 4.12. The calculation is triggered by an update of the entry in table *S_CalcExpTimes*. Once the triggers have finished the updated expiration times of the event instances are stored in the corresponding *P_<EC>* tables (Table 4.10).

Table Name	**Attributes**	**Key**
S_CalcExpTimes	occTime INT detTime INT	-
P_<EC>	detTime INT exp INT key from <EC>	key from <EC>

Table 4.10: The tables used to trigger the expiration time calculation and the table created for each event class containing the expiration information.

Now that the expiration times have been calculated the purging of expired events needs to be started. Table 4.11 depicts the table *S_PurgeHistory*, containing one entry, which is used to start the purging of events from their corresponding tables (history, published events, etc.). The entry of this table is updated in the read task.

Purging is performed with three triggers; one, *t_Purge_<SEC>*, purges subscribed event instances from *<SEC>* and subsequently *H_<SEC>*. If any published events exists these are cleansed too. Finally the expiration time entry is removed from *P_<SEC>*. The purging of complex events works in a similar fashion; first the trigger *t_Purge_H_<CEC>* removes expired events from the history table. This triggers *t_Purge_R_<CEC>* which then cleanses *R_<CEC>* and thereafter removes the expiration time entry from *P_<CEC>*. As with subscribed events any published events are deleted afterwards.

Table Name	**Attributes**	**Key**
S_PurgeHistory	occTime INT detTime INT	-

Table 4.11: The table for triggering the purging of events.

Trigger Name	**Condition**	**Priority**
t_P_Calc_Leaves	BEFORE UPDATE ON S_CalcExpTimes	-100
t_P_Calc_Roots	AFTER UPDATE ON S_CalcExpTimes	-100
t_Purge_<SEC>	BEFORE UDPATE ON S_PurgeHistory	-80
t_Purge_R_<CEC>	AFTER UPDATE ON S_PurgeHistory	-80
t_Purge_H_<CEC>	AFTER DELETE ON H_<CEC>	-80

Table 4.12: The triggers used for purging.

Optionally, purging can be performed with archiving activated; if the archiving option is used purged events are stored in archive tables depicted in Table 4.13. These archive tables are created for complex and subscribed event classes; archives for action classes are not created as they can be deduced from the former. Archiving, if active, is done in the triggers *t_Purge_<SEC>* and *t_Purge_H_<CEC>*. Before purging the history events concerned are archived in their corresponding *A_<EC>* tables.

Example 4.3: H2 Purging Example

Consider Example 2.8 with the archiving option is activated. The subscribed event *ResourceEmpty* is to be purged at 14.4.14 9:00 as is the complex event

Table Name	Attributes	Key
A_<EC>	based on <EC>	based on <EC>

Table 4.13: The archive table for every event class if the archiving option is activated.

InstantReplenishingNeeded. The trigger *t_P_Calc_Leaves* is set off first, *t_P_Calc_Roots* follows. These calculate and write the expiration time to the tables *P_ResourceEmpty* and *P_InstantReplenishingNeeded*. Thus at clock tick 14.4.14 9:15 first the subscribed event is purged from *ResourceEmpty*, then archived and subsequently cleansed from the history. Finally the expiration entry from *P_ResourceEmpty* is deleted.

Afterwards the purging of InstantReplenishingNeeded begins. First the entry is archived in *A_InstantReplenishingNeeded*. Subsequently *H_InstantReplenishingNeeded* is cleansed followed by *R_InstantReplenishingNeeded* and *P_InstantReplenishingNeeded*. Subsequently the event is purged from all published event classes.

4.1.4 Optimization Strategy: Global Occurrence Time Optimization

This section gives an overview of the global occurrence time optimization strategy. The main idea is that triggers only need to be executed if they will potentially discover something thus reducing the execution time. Currently this is only implemented for *ontime* and *late* triggers by performing the SQL inserts only on a potential clock tick. When such a potential clock tick for an event class occurs is stored in table *S_Potential_ClockTick* (Table 4.14). The potential clock ticks are calculated and stored in this table by the trigger *t_h_<ec>_PotClockTick* depicted in Table 4.15. This trigger is created for each event class having event publication statements associated with it. It merges the occurrence time and the event class name of an event into *S_Potential_ClockTick*. Subsequently all entries having occurrence times before or equal to the current $clocktick - chronon$ are deleted. For an example refer to Example 4.4.

Table Name	Attributes	Key
S_Potential_ClockTick	occTime INT eventclass VARCHAR	eventclass occTime

Table 4.14: The table storing the potential clock ticks for the global occurrence time optimization.

Trigger Name	Condition	Priority
t_H_<EC>_PotClockTick	AFTER INSERT ON H_<EC>	-10

Table 4.15: The trigger for the global occurrence time optimization.

Example 4.4: The Global Occurrence Time Optimization

A Delivery event arrives with an occurrence time of 14.5.14 9:00; the current time is 12.5.14 15:00. Thus *t_h_<ec>_PotClockTick* writes the occurrence time to the table *S_Potential_ClockTick* with "Delivery" as *eventclass*. Between 12.5.14 15:00 and 14.5.14 9:00 no new Delivery is detected thus the triggers *ontime* and *late* are never entered. On the clock tick 14.5.14 9:00 the *ontime* trigger is executed as an potential clock tick entry is found in *S_Potential_ClockTick*. The next clock tick this entry is deleted.

Example 4.5: H2 Trigger Sequence Example

Consider the scenario described in Section 1.3. A new event instance of ResourceLow occurs at 11:00 and is detected at 10:30, thus it is in the future. Later on this event is changed to occur at 10:45 and is detected at 10:45, thus it is on time (assuming that ResourceLow is mutable for this example). Table 4.16 depicts the trigger sequence set off by forwarding the events to the CEP, i.e., inserting the instances into the history table, without purging. The *tx* are symbols for the triggers.

Tables 4.17 to 4.19 show the state of the tables after the trigger *tx* defined in Table 4.16 has performed. Table 4.17 depicts the entries in the tables for ResourceLow. The first event is initially inserted into *H_ResourceLow* and consequently it is copied to *ResourceLow*. The next clock tick the updated event is detected and inserted into *H_ResourceLow* and *ResourceLow* is updated. Note that there is only one entry in *ResourceLow* whereas the history contains two. Table 4.17 depicts the state after the second event has been

forwarded. Once the event is in *ResourceLow* the view *ReplenishingNeeded* contains the event. *t2* transfers this view update to *H_ReplenishingNeeded* and *R_ReplenishingNeeded*. In the same step the entry for the optimization strategy is made in *S_Potential_ClockTick*, *t3*. Since the event is on time the trigger *t4* copies the event to the published event class *orderResource*. *t5* does not update any table content as its condition is not met. The entry in *S_Potential_ClockTick* is then deleted when the next clock tick is processed. This completes the sequence triggered by inserting the two ResourceLow events.

	Trigger	**Condition**
t1	t_MViewResourceLow_insert	AFTER INSERT ON H_ResourceLow
	Update S_UpdateHistory	
t2	t_H_ReplenishingNeeded_history	AFTER UPDATE ON S_UpdateHistory
t3	t_H_ReplenishingNeeded_PotClockTick	AFTER INSERT ON H_ReplenishingNeeded
	Update clockTick	
t4	t_orderResource	AFTER UPDATE ON clockTick
t5	t_orderResource0	AFTER UPDATE ON clockTick

tx... A symbol for the specific trigger.

Table 4.16: An example trigger sequence in the CEP after forwarding a ResourceLow event.

H_ResourceLow

occTime	detTime	resource	amountLeft
11:00	10:30	Yoghurt	50g
10:45	10:45	Yoghurt	100g

ResourceLow

	occTime	detTime	resource	amountLeft
t1	10:45	10:45	Yoghurt	100g

Table 4.17: The changes made to the subscribed event class ResourceLow tables when a ResourceLow event is forwarded.

ReplenishingNeeded

	occTime	detTime	resource
t1	10:45	10:45	Yoghurt

H_ReplenishingNeeded

	occTime	detTime	resource
t2	10:45	10:45	Yoghurt

R_ReplenishingNeeded

	occTime	detTime	resource
t2	10:45	10:45	Yoghurt

orderResource

	occTime	detTime	resource
t4	10:45	10:45	Yoghurt

Table 4.18: The changes made to the complex event class ReplenishingNeeded tables when a ResourceLow event occurs.

S_Potential_ClockTick

	occTime	eventclass
t3	10:45	ReplenishingNeeded

Table 4.19: The entry made in *S_Potential_ClockTick* for the global occurrence time optimization when a ReplenishingNeeded occurs.

4.2 H2 Data Base

Before delineating the details of the CEP's implementation using H2 a short overview regarding H2 is given here. In particular the focus lies on the qualities relevant for the implementation and differences to SQLite.

H2 is a fast open source Java database outperforming comparable databases like MySQL, Hyper SQL Database (HSQLDB), Derby, etc. in most benchmarks (refer to H2 (2014b) for the specific results). H2 provides an embedded version as well as a client-server version and clustering support; SQLite on the other hand provides only an embedded version. Furthermore it can either be run completely in-memory (data is not persisted) or as a disk based database. Such H2 databases can be accessed via Java Database Connectivity (JDBC), for easy access from Java applications, or Open Database Connectivity (ODBC). The ODBC Application Programming Interface (API) offered can be accessed using the PostgreSQL ODBC driver. Multiple connections to a database at one time are supported. Additionally to these qualities H2 has a very small footprint of under 1.5 MB making it a good choice for mobile devices and low resource systems. To manage an H2 database a handy web user interface for browsing, querying, and managing data within a database is provided. (H2, 2014a)

With the general features of H2 described above the SQL specific features of H2 are named here. H2 is an SQL based database and thus supports most of its features. The relevant features for this thesis are referential integrity, joins, subqueries, aggregation functions (including `group by` and `having`), auto increments, indexes, and stored procedures/Triggers. Additionally H2 provides transaction support and a query optimizer. (H2, 2014a)

These qualities suggest that an H2 based CEP should run faster than an SQLite based one on a PC/laptop or mobile device. Especially, since H2, unlike SQLite, supports hash indexes speeding up specific value queries. These hash indexes can only be utilized if an in-memory database is used or the table is declared as memory table (i.e. the index data is stored in the main memory). As mentioned previously, H2 supports triggers but currently these triggers, like all user defined functions, have to be written in Java. Each trigger is one Java class extending the `Trigger` class of H2 which is subsequently added to the database via an SQL statement. (H2, 2014a)

Taking all these features and qualities into account H2 has some advantages over SQLite but also causes issues:

- The indexing feature of H2 works as long as enough memory is available.
- The requirement to provide triggers as Java classes results in the necessity to create the triggers dynamically at runtime. This is due to the fact that the BiCEPL program is read when the framework is already running.
- Moreover H2 does not provide a `when` clause for triggers. Therefore a trigger is executed in any case and consequently such conditions need to be evaluated inside the trigger.
- The implementation of triggers as Java classes is problematic regarding performance as any SQL statements issued in a trigger, like selects, inserts, etc., are executed via JDBC. Consequently Java triggers, when accessing or modifying tables, will be slower than a Procedural Language/Structured Query Language (PL/SQL) implementation.
- H2 does not provide functions to determine the max or the min value of a given list of values. This can easily be dealt with by creating two one line Java functions in H2.

These issues being stated, the implementation of the CEP will be described in the next section taking these problems into account.

4.3 Implementation of the Approach with Java and H2

The previous section, Section 4.1, introduces the implementation approach developed for the PeaCE framework's extended CEP. This section delineates how this approach is implemented using Java and H2. Therefore the Java prototype depicted in Section 2.6 has to be extended. Fortunately, PeaCE is designed with extensibility in mind, thus only a few new subclasses have to be created to support H2 and its specific features. Particularly challenging is the fact that H2 requires triggers to be in Java. These triggers need to be created dynamically at runtime as their content is only known then and not beforehand. A tool to modify and create Java code at runtime is Javassist which is described in Section 4.3.1. Thereafter Section 4.3.2 and Section 4.3.3 depict the implementation of the H2 CEP. The new classes are located in the package `biceps.bicep.knowledge_mapper.h2` whereas the SQLite implementation is contained in `biceps.bicep.knowledge_mapper.sqLite`. Section 4.3.2 describes the classes used to create the BiCEPL program specific CEP, i.e.,

the H2 database is completely set up for the given program. Section 4.3.3 then explains the classes needed to embed the H2 CEP into the PeaCE framework instance, i.e., to feed events into it, to control the execution, and to forward the derived actions. Note that the following sections give only a brief overview; for the details refer to the Javadoc.

4.3.1 Javassist

As depicted in Section 4.2 using H2 for implementing the CEP requires dynamic creation of Java classes. One alternative for dynamically creating/altering Java code is the Javassist library which is described in detail in Chiba (2000). Javassist provides structural reflection additionally to behavioural reflection (changing the behavior of operations), i.e., data structures can be altered once they are created and new classes can be defined at runtime. The main advantage of Javassist is that it provides a source code API thus no knowledge about Java bytecode is necessary. (Chiba, 2013)

In this thesis Javassist is used to dynamically create the triggers needed to instantiate the specific CEP for a given BiCEPL program. In particular its ability to create classes on-the-fly by specifying Java source code is utilized (refer to Section 4.3.2).

4.3.2 Knowledge Mapper

The knowledge mapper, contained in the package `biceps.bicep.knowledge_mapper`, compiles a knowledge base (`biceps.bicep.knowledge_base.KnowledgeBase`) created by the BiCEPL parser, or by hand, into the specific database schema and the corresponding triggers. The term mapper is a legacy term used in the Java prototype and is used in this thesis as well (although not correct). This compilation is implemented using an abstract class `KnowledgeBaseMapper` as base class which is extended by concrete mappers. Currently two specific mappers, namely `H2Mapper`, developed in this thesis, and `SQLiteMapper` are provided by the prototype. The sections below each describe one package of the knowledge mapper.

H2Mapper The `H2Mapper` extends the `KnowledgeBaseMapper` and thus implements the abstract methods for creating the database schema and the corresponding triggers, i.e., event classes are compiled into the H2 database as

described in Section 4.1. Furthermore, `H2Mapper` creates the database functions necessary for the CEP to work, for instance the `max(int... args)` function returning the maximum value of all the values provided. The `KnowledgeBaseMapper` and its H2 subclass are depicted in Figure A.8.

In particular `H2Mapper` inherits four abstract methods:

- `mapSubscribedEventClass` which compiles a given subscribed event class into the corresponding tables and triggers.
- `mapComplexEventClass` which compiles complex event classes.
- `createEnvironment`/`createMobileEnvironment` which use the former two to compile the whole knowledge base into an H2 database. Additionally these two are responsible for creating the system tables, compiling the purging feature if activated, compiling the global occurrence time optimization feature, as well as creating the additional database functions needed and ensuring the execution of triggers according to their priorities. Since H2 does not support trigger priorities these are implemented using a workaround; the Java triggers are first stored in a list, subsequently ordered and thereafter registered to the database.

`mapSubscribedEventClass` and `mapComplexEventClass` utilize the classes of the packages described in the sections following, especially these of the `triggerBuilder` package.

Helpers The `H2Mapper` uses the classes `DBHelper` and `H2Handle` contained in this package besides `H2TriggerBuilder` and `H2FireBuilder` from `triggerBuilder` to fulfil its task. `H2Handle` is a singleton class which allows to access and execute SQL statements on an H2 database. The singleton is implemented to ensure only one connection at a time and to be able to access the same connection object from any class. `DBHelper` on the other hand provides methods like getting the column names of a table as string with different separators. This class also provides methods to `H2FireBuilder`. Like `H2Handle` `DBHelper` too is implemented as singleton as only one instance is needed and consequently passing references of the object is thus not necessary. The two classes and their relationship to `H2Mapper` are depicted in Figure A.9. The package `triggerBuilder`, which is used to build the actual Java triggers needed, is explained in the next section.

TriggerBuilder The package `TriggerBuilder` (Figure A.10 and Figure A.11) contains two classes, viz. `H2TriggerBuilder` and `H2FireBuilder`.

The `H2TriggerBuilder` provides the means to dynamically create Java classes using the Javassist library. This class implements a modified builder pattern to allow for building and registering various types of triggers (e.g. after insert, before update, ...). Furthermore it permits to assign priorities to the triggers and to sort them according to their priorities by implementing the `Comparable` interface. The code generation for triggers is performed in `H2FireBuilder`; `H2TriggerBuilder` then creates a subclass of the `org.h2.api.Trigger` class using Javassist, adds the generated code to the method `fire`, loads the new class and finally registers the trigger with the H2 database.

The `H2FireBuilder`, like the `H2TriggerBuilder`, implements a modified builder pattern which allows to assemble triggers without having to write a separate method for each one. The parts common to the various triggers are extracted into methods; the building of the trigger code is then an execution sequence of these methods based on what is needed for a certain trigger. How a trigger looks like depends on its type, "static" triggers look more or less always the same (e.g. materialized view triggers, purging triggers, ...). The complex triggers are the ones for action classes. These triggers insert events into published event classes if their specified condition is met. Consequently these conditions, defined in BiCEPL, have to be transformed into Java Triggers. Each trigger needs to evaluate the condition on the H2 database and to insert the events for which the condition holds into the corresponding tables. This is implemented utilizing the condition object hierarchy created by the BiCEPL parser. Each Class in this hierarchy implements a method called `toH2ConditionSQL` which rewrites the current object into an SQL condition. Such a condition can contain place holders which are then replaced in `H2FireBuilder` with the corresponding SQL conditions. These place holders are necessary as some conditions require information about the event upon which the condition-action statement is defined. These information are not available within the condition object hierarchy. A restriction made here is that each condition may only contain one timing primitive.

4.3.3 Execution Model

The package `execution_model` contains all classes necessary for the execution of a CEP. These are the database handles forming the `db` package, the tasks building the `task` package, and the task managers located in the main package. For the tasks refer to Section 2.6.

The main class of the execution model is `TaskManager` and its subclasses as depicted in Figure A.12. They are responsible for executing the task

queue (read, execute, and publish) at each clock tick. The specific subclasses address different databases and platforms. Each `TaskManager` contains a `DBHandle` from the package `db` for the specific database used for the CEP. These handles provide methods to insert events into the database, for transaction handling, updating the various system tables, establishing database connections, executing SQL statements, etc. The read task, for instance, uses one of these methods to forward events to the database. The handles available right now, SQLite and H2, are depicted in Figure A.7.

4.4 Differences to the SQLite Implementation of the PeaCE Framework

In this section the differences of the described H2 implementation and the SQLite implementation of the CEP are explicitly stated. First the functional differences are delineated. Thereafter the differences caused by the qualities of the H2 database system are explained.

The foremost difference is the implementation of more complex conditions. Additionally to the eight basic timing primitives the H2 implementation allows complex conditions on event attributes (of the current as well as of the previous instance) and literals. Furthermore `FIRED` and `NOW` are supported. `NOW` refers to the current time whereas `FIRED` returns true if the event instance of the corresponding action class has occurred yet. A restriction continued from SQLite is that only one timing primitive per condition is supported.

This extended support of conditions requires changes to the way conditions are represented in the Java prototype. These changes are described in detail in Section 3.4.5.

Besides these changes to the Java implementation the implementation approach has been modified in order to enable extended conditions. Some of these alterations have been made regarding triggers. Most triggers are implemented analogously to the SQLite CEP but with different trigger conditions. Consequently the SQL statements in the triggers have been changed too but the end result is the same. The main differences are regarding the triggers for condition-action statements due to the extended condition support.

Furthermore, alterations regarding the tables have been made, namely the removal of the fired tables. In the SQLite implementation each of these tables contains actions fired and the corresponding timing primitive of the condi-

tion. One table is created for each event class containing condition-action statements. The evaluation of some timing primitives (e.g. `LATE`) then queries this table whether another timing primitive has already fired, for instance for `LATE` no `ONTIME`, `LATE` or `RETROACTIVECHANGE` must have fired. This only works if these timing primitives occur in the condition-action statements of a class, otherwise the fired table would not contain any entries on them. This, the extended condition support in the H2 CEP, the restructuring of the triggers and trigger sequences, and keeping in mind the future support of several timing primitives in one condition results in the decision to discontinue the use of the table. Consequently it is replaced by queries on the tables of the event classes (see Section 4.1.1) thus making queries on timing primitives not contained in any condition of an event class possible. This on the one hand allows for more complex condition evaluation, on the other hand tables are queried which contain more data than the fired table thus the query evaluation takes more time.

The qualities of H2 lead to further distinctions between the H2 and the SQLite version. Especially the requirement of functions and triggers to be in Java demands changes in the Java architecture. The SQLite mapper, compiling the knowledge hierarchy into SQLite, creates the PL/SQL code for triggers as strings and then to adds them to the database. The H2 version on the other hand needs to build Java classes for these triggers, and subsequently load and register them to the database at runtime. Since the H2 `Trigger` class does not support conditional triggering beyond tables, i.e., conditions on attributes, e.g. occtime < 0, cannot be used when registering the trigger. This has to be implemented using if statements within the triggers. SQLite on the other hand implements a where clause on the trigger registration which allows to further restrict the execution of a trigger.

4.5 Performance Improvements

This section describes the techniques used to improve the performance of the H2 CEP. They concentrate on reducing the computation load by optimizing the design rather than tweaking the database.

Before describing the changes the measurement methods shall be explained here. Each of the three main tasks executed at each clock tick encapsulates its database tasks into transactions. Whenever a task starts or ends a transaction an entry of the current system time and the task is made into a log table. This table is then subsequently used to determine the performance

of the system. This technique is also used in Huemer (2014) for comparing the SQLite with the H2 implementation. Furthermore the simple H2 Profiler provided by the H2 database is used for determining which database processes took the most time.

With the measurement methods now explained the improvments are described here. Some improvements have been achieved reducing the select statements issued to the database. Moreover, the queries on views have been replaced due to their slow response times. For the same reason queries on the history tables are only made when unavoidable. Moreover, using the hash indexes available in H2 the performance is increased especially if high event capacities are the case. Interestingly the H2 Profiler shows that using hash indexes for these cases sometimes actually decreases the time spent with maintaining them (time spent in org.h2.index).

4.6 Testing

The previous sections described how the H2 version of the CEP is implemented, this section copes with the test method used.

Testing the system is quite difficult as the result can vary from run to run as system parameters like available memory or CPU time have a high influence. Therefore the tests are, to keep it simple, conducted not automated. To do so the simulation environment of the PeaCE framework is used. This environment uses a static detector which is a Java class containing events and the times when they should be forwarded to the CEP. A BiCEPL program has been developed which is sufficient to cover the test scenarios. Test scenarios are, for instance, specific timing primitives, condition-action statements, global occurrence time optimization, etc. For the classes defined in the program the event instances necessary for the scenarios are set in the static detector. Using this setup the functionalities of the H2 CEP are tested. The checks whether the system works correct are done via the simulation environment user interface and the database console.

The method described above is used to test the semantic correctness of the system. To test the performance two major approaches are used: measuring time spent in tasks, and the H2 Profiler. The former has first been tried with JLog and files but this approach slows down the system too much. Consequently the times are now logged in the database allowing for better performance. The latter is a simple tool for CPU profiling (H2, 2014b) which

is used to find out which H2 database processes consume the most time and thus are potential candidates for optimizing.

Chapter 5

Conclusion

This thesis extends the event processing framework Processing Event Ads into Complex Events (PeaCE) (Huemer, 2014) by two components, a compiler for its Bitemporal Complex Event Processing Language (BiCEPL) which transfers a BiCEPL program into an H2 based Complex Event Processor (CEP). The PeaCE framework is based on a novel bitemporal event model, defining an event's occurrence time and detection time and thus supports mutable events as well as delayed event detections and processings. Moreover the PeaCE framework is formed by three parts, detector(s), complex event processor(s), and action executor(s).

In this thesis a compiler implementation for BiCEPL using ANother Tool for Language Recognition version 4 (ANTLRv4) is introduced. This compiler is integrated into the Java prototype of the PeaCE framework which requires minor changes to it. The parser part of the compiler supports full BiCEPL, the compiler supports the parts available in the PeaCE CEP.

Moreover, an H2 based implementation of the CEP is created for the Java prototype. Since H2 requires triggers to be Java classes and triggers to be created are only known after compiling a BiCEPL program, these Java classes must be created dynamically at runtime. The H2 version of the CEP has extended condition support compared to its SQLite sibling. Nevertheless, the H2 version is still restricted by the use of at most one timing primitive per condition-action statement. The performance of this implementation is then improved by tweaking the design and usage of hash indices.

Concluding, future work concentrates on three areas: First, the compiler can be extended to support BiCEPL fully. Second, further performance improvement is possible regarding the H2 implementation by tweaking the

database parameters and settings. Moreover, H2 will support PL/SQL in future (H2, 2014c) making the trigger access via JDBC obsolete and thus faster. At last future work can cover the porting and optimization of the H2 CEP to mobile devices.

Bibliography

Chiba, S. (2000). Load-time structural reflection in java. In E. Bertino (Ed.), *Ecoop 2000 — object-oriented programming* (Vol. 1850, pp. 313–336). Springer Berlin Heidelberg.

Chiba, S. (2013). *Javassist.* Retrieved from `http://www.csg.ci.i.u-tokyo.ac.jp/~chiba/javassist/`

compiler.net. (2010). *compilers.net > paedia > compiler.* Retrieved from `http://www.compilers.net/paedia/compiler/`

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

Etzion, O. (2010). Temporal perspectives in event processing. In A. Hinze & A. P. Buchmann (Eds.), *Principles and applications of distributed event-based systems* (pp. 75–89). IGI Global.

Gagnon, E. (1998). *Sablecc, an object-oriented compiler framwork* (Unpublished master's thesis). School of Computer Science McGill University, Montreal.

Godfrey, D. (2011). *Iso sql 2003 grammar.* Retrieved from `http://www.antlr3.org/grammar/1304304798093/SQL2003_Grammar.zip`

H2. (2014a). *Features.* Retrieved from `http://www.h2database.com/html/features.html`

H2. (2014b). *Performance.* Retrieved from `http://www.h2database.com/html/performance.html`

H2. (2014c). *Roadmap.* Retrieved from `http://www.h2database.com/html/roadmap.html`

Huemer, M. (2014). *Bitemporal complex event processing of web event advertisements* (Doctoral dissertation). Johannes Kepler University.

JavaCC. (2013a). *Javacc features.* Retrieved from `https://javacc.java.net/doc/features.html`

JavaCC. (2013b). *Javacc home.* Retrieved from `https://javacc.java.net/`

Johnson, M. (2008). *Lalr parsing.* Retrieved from `http://`

dragonbook.stanford.edu/lecture-notes/Stanford-CS143/11-LALR-Parsing.pdf

Lewis, F. D. (2014). *Top-down parsing and ll(1).* Retrieved from http://www.cs.engr.uky.edu/~lewis/essays/compilers/ll-lang.html

Luckham, D. (2006). *What's the difference between esp and cep.* Retrieved from http://www.complexevents.com/2006/08/01/what%E2%80%99s-the-difference-between-esp-and-cep/

Michelson, B. M. (2006). *Event-driven architecture overview.* Retrieved from http://www.omg.org/soa/Uploaded%20Docs/EDA/bda2-2-06cc.pdf

Mössenböck, H. (2007). *Generating compilers with coco/r.* Retrieved from http://www.ssw.uni-linz.ac.at/Research/Projects/Coco/Tutorial/slides.zip

Mössenböck, H., Löberbauer, M., & Wöß, A. (2011). *The compiler generator coco/r.* Retrieved from http://www.ssw.uni-linz.ac.at/Research/Projects/Coco/

Parr, T. (2012). *The definitive antlr 4 reference* (S. Pfalzer, Ed.). The Pragmatic Bookshelf.

Parr, T. (2013). *Antlr.* Retrieved from http://www.antlr.org/

Sellers, A. J. (2011). The oxpath to success in the deep web. In *Proceedings of the 20th international conference companion on world wide web* (pp. 409–414).

SQL. (2003). *Iso/iec 9075:2003 sql language* (No. 2). Retrieved from http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=34133

List of Figures

List of Tables

List of Listings

List of Grammars

List of Examples

Acronyms

PeaCE Processing Event Ads into Complex Events. 3–9, 11, 12, 16–21, 23, 26, 29, 31–34, 38, 41, 43–46, 52, 60, 61, 66, 68

AC Action Class. 43, 48, 52

ANTLRv4 ANother Tool for Language Recognition version 4. 24, 26–29, 31, 33–36, 38, 40–42, 68, A, B, F

API Application Programming Interface. 59, 61

BiCEP Bitemporal Complex Event Processor. 22, 23

BiCEPL Bitemporal Complex Event Processing Language. III, IV, 3–6, 8, 9, 11–17, 21, 24–26, 28–38, 40–42, 46–48, 50, 60, 61, 63, 66, 68, A

BSD Berkeley Software Distribution. 27, 28

CEC Complex Event Class. 43, 47, 48, 50, 54

CEP Complex Event Processor. III, IV, L, 4–6, 8, 12, 19–23, 38, 43–45, 48, 56, 57, 59–66, 68, 69, G

EBNF Extended Backus-Naur Form. 13, 15, 26, 29, 31, 33, 34

EC Event Class. 43, 52, 53, 56

HSQLDB Hyper SQL Database. 59

JDBC Java Database Connectivity. 59, 60

LPGL GNU Lesser General Public License. 28

ODBC Open Database Connectivity. 59

PL/SQL Procedural Language/Structured Query Language. 60, 65, 69

RFID Radio frequency Identification. 4

SEC Subscribed Event Class. 43, 46–48, 50, 54

SQL Structured Query Language. 8, 13–15, 29, 33, 36, 37, 41, 42, 46–52, 55, 59, 60, 63, 64

Appendix A

Class Diagrams

The UML diagrams presented in this chapter are, for the sake of readability and to be able to present them here, not complete.

A.1 BiCEPL

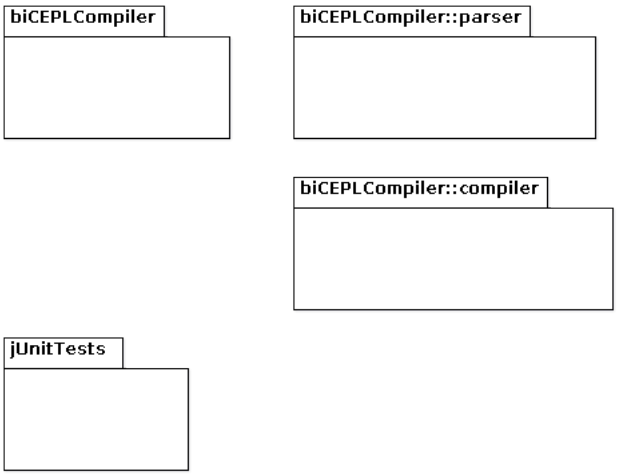


Figure A.1: The UML class diagram of the packages in the BiCEPL compiler. The package compiler contains only classes generated by ANTLRv4.

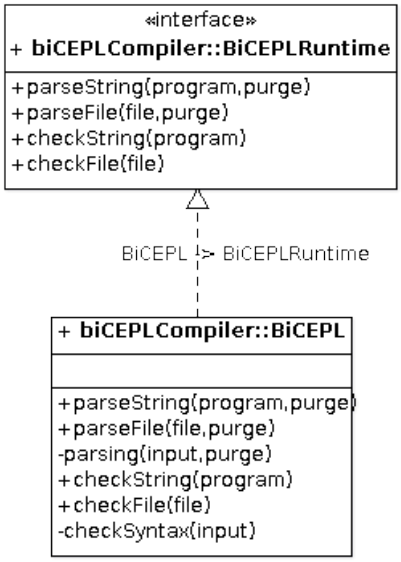


Figure A.2: The UML class diagram of the interface and its concrete implementation.

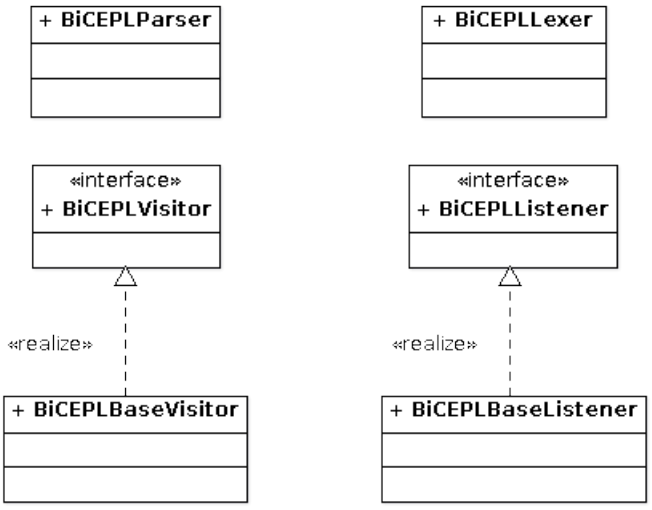


Figure A.3: The UML class diagram for the parser package. These classes are generated by ANTLRv4.

Figure A.4: The UML class diagram for the compiler package. The classes in gray are generated/imported classes.

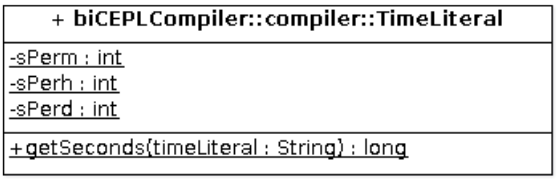


Figure A.4: The UML class diagram for the compiler package (continued).

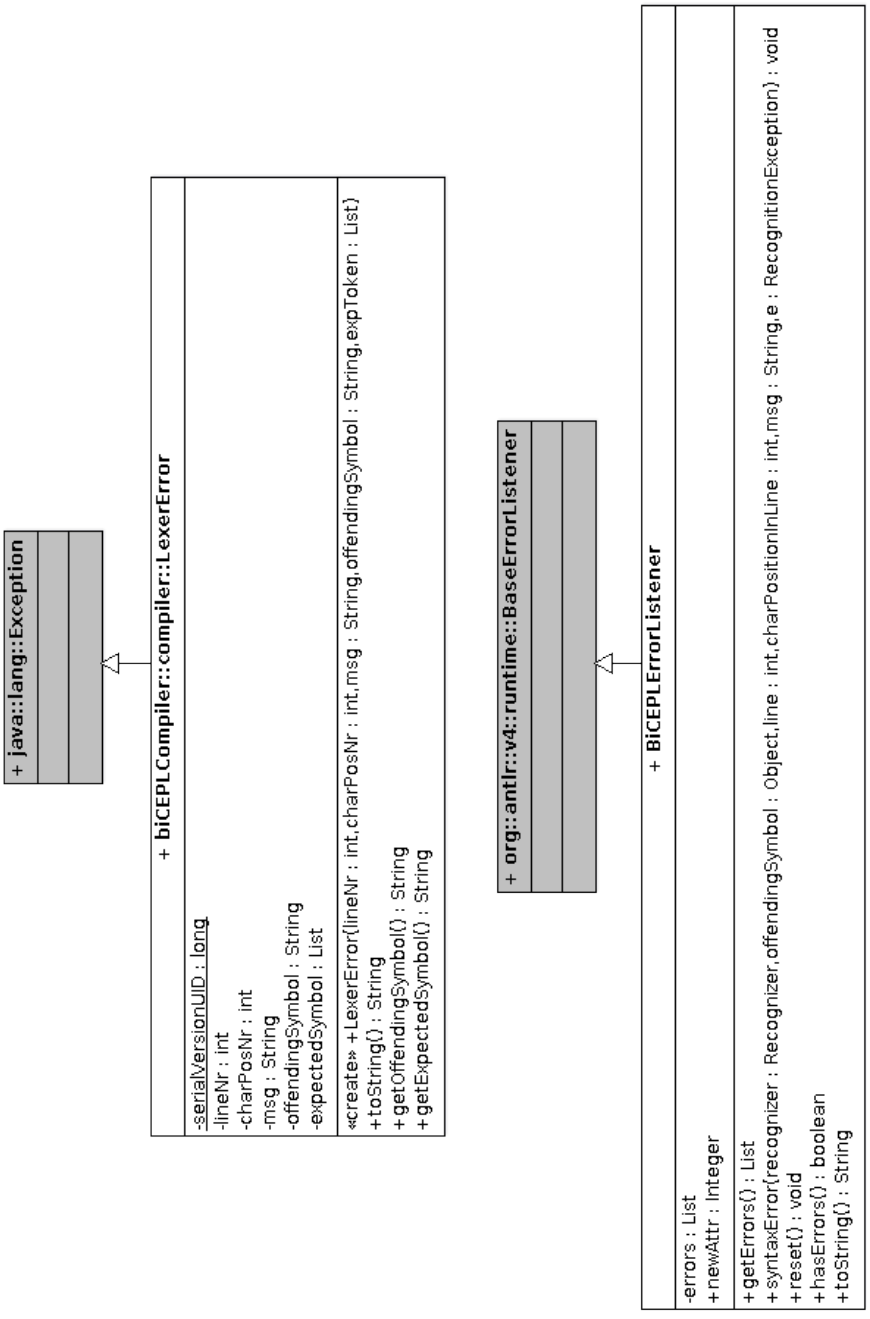


Figure A.5: The UML class diagram for the error classes in the parser package.

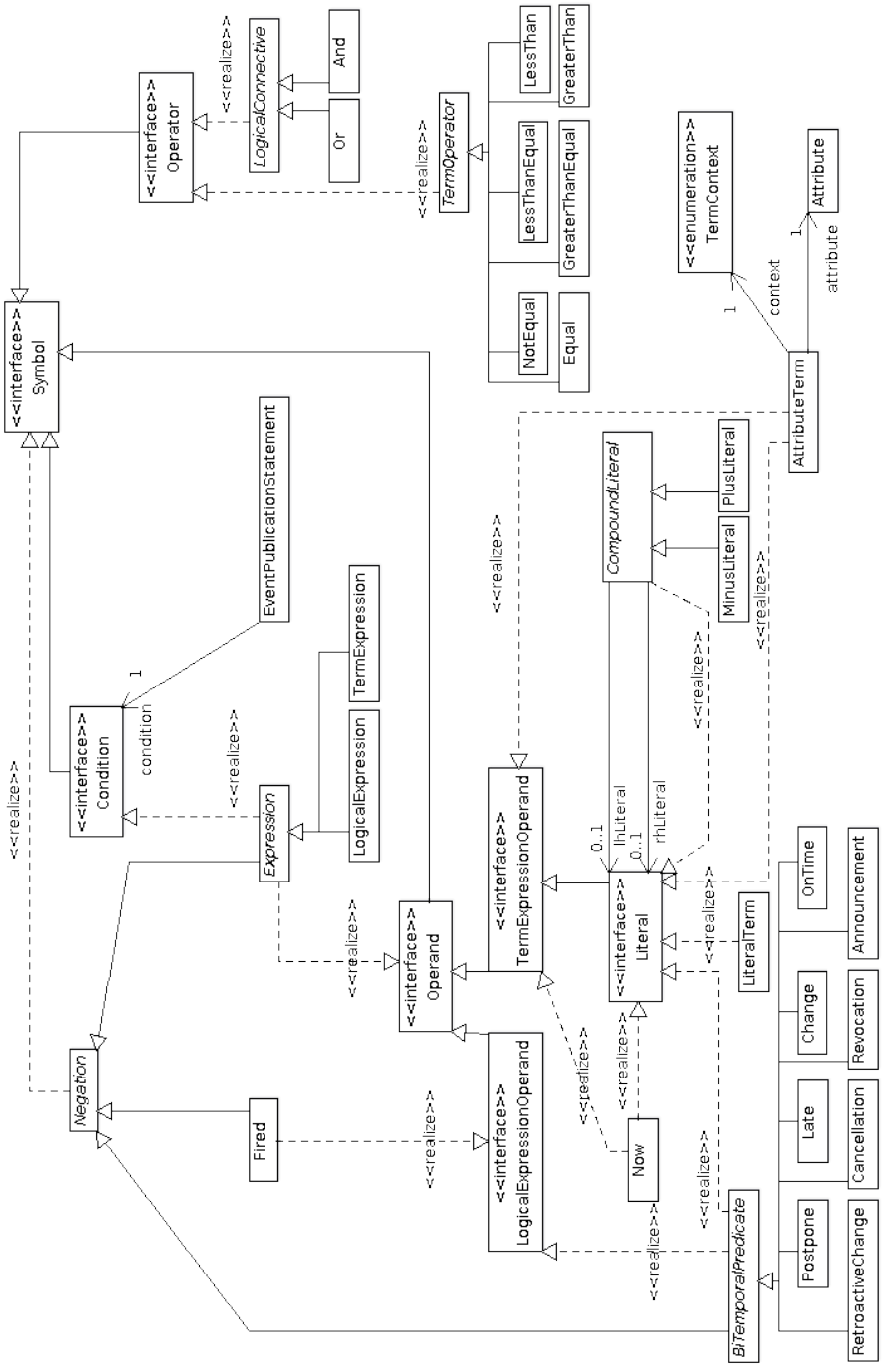


Figure A.6: The UML class diagram for the condition elements after remodelling them to work with ANTLRv4 and `BiCEPLPeACEVisitor`.

A.2 H2

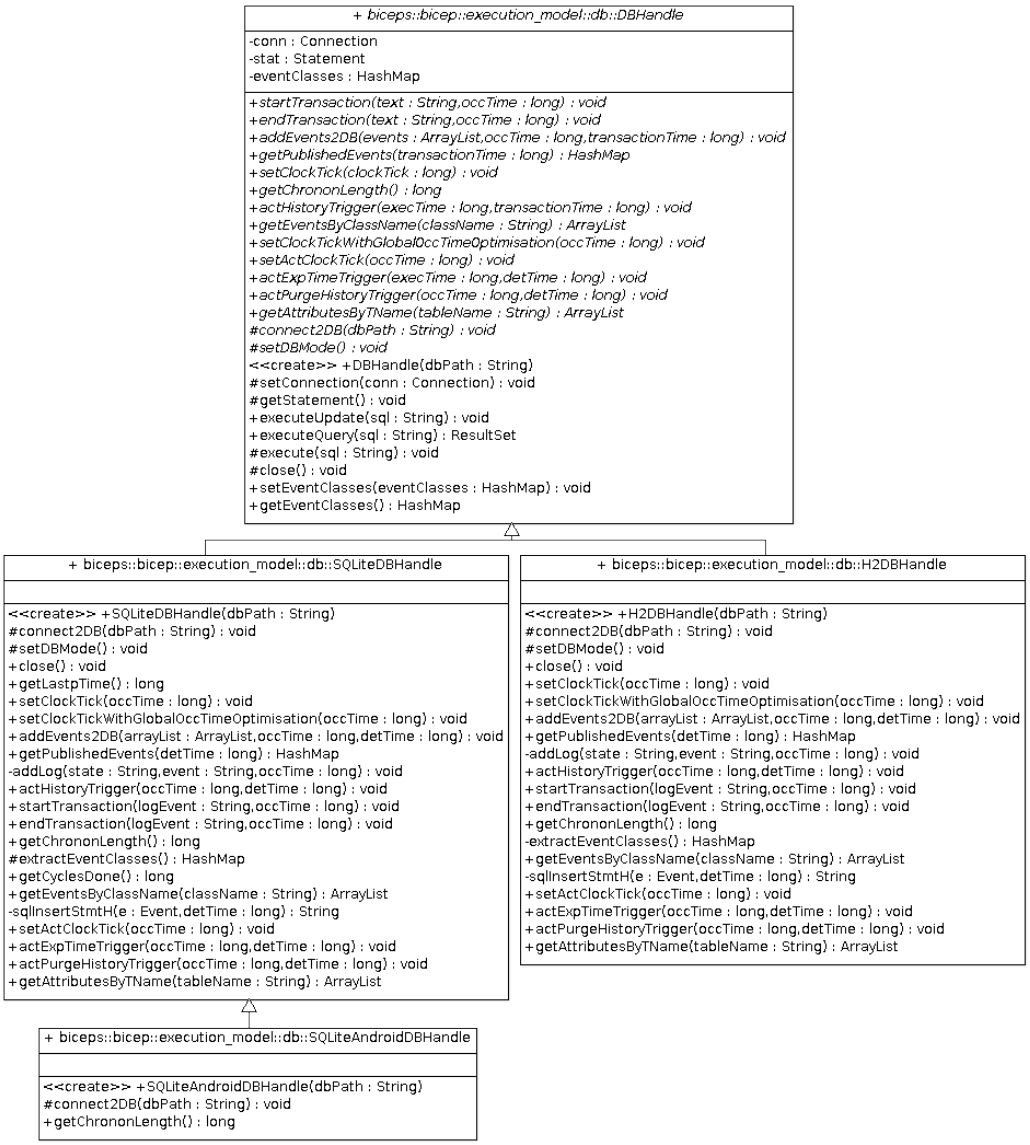


Figure A.7: The Java database handles used to access the CEP database.

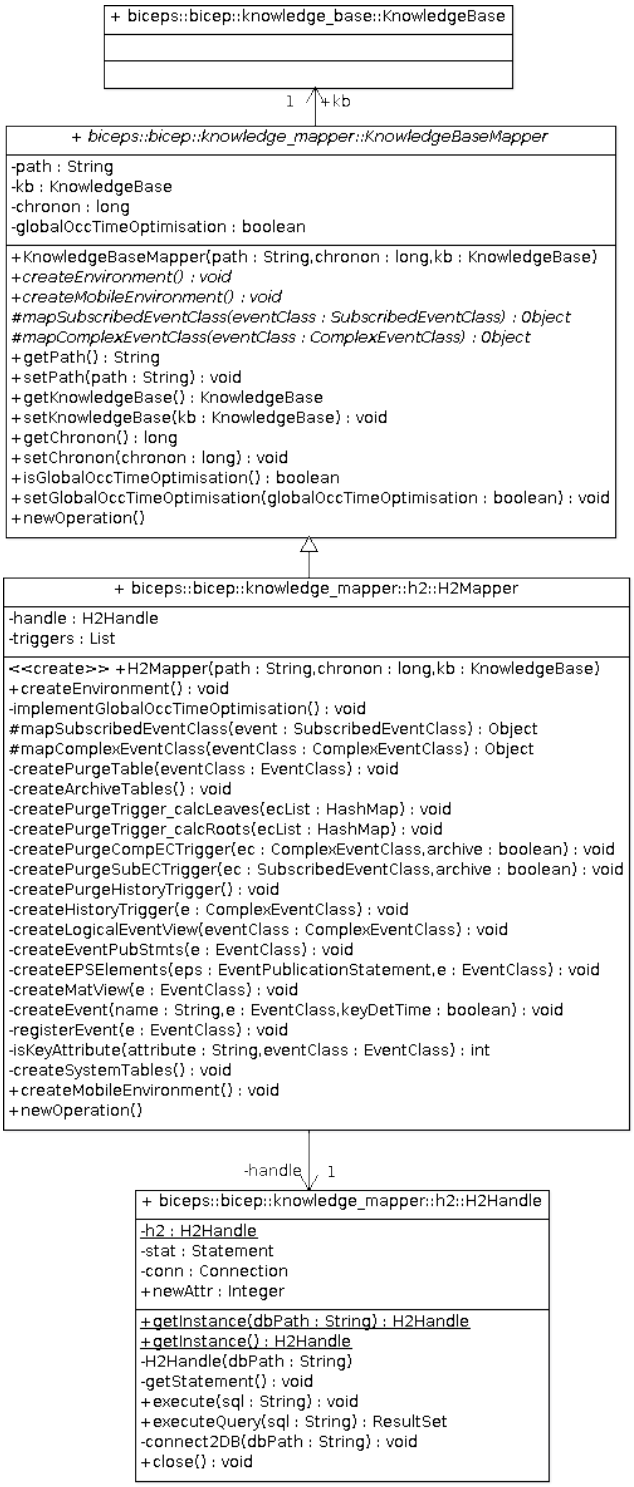


Figure A.8: The UML class diagram of the H2 knowledge mapper and its super class.

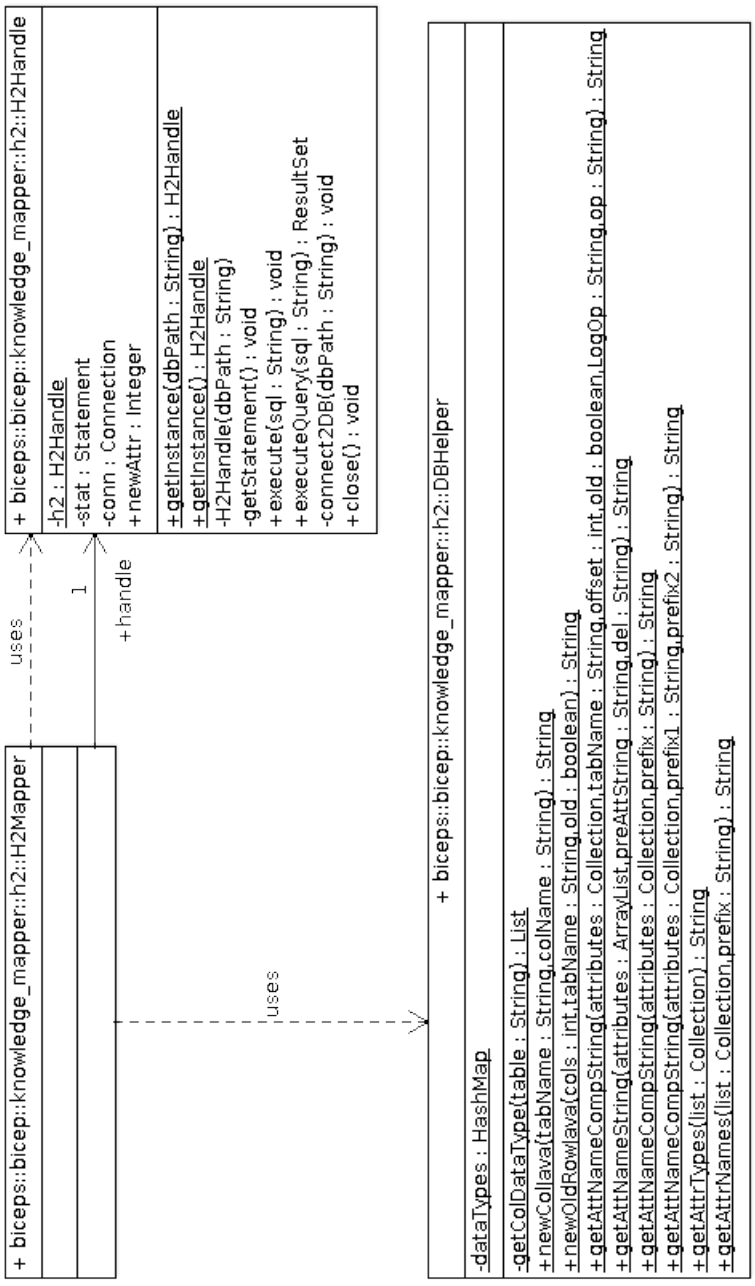


Figure A.9: The `H2Mapper` and the helper classes used by it.

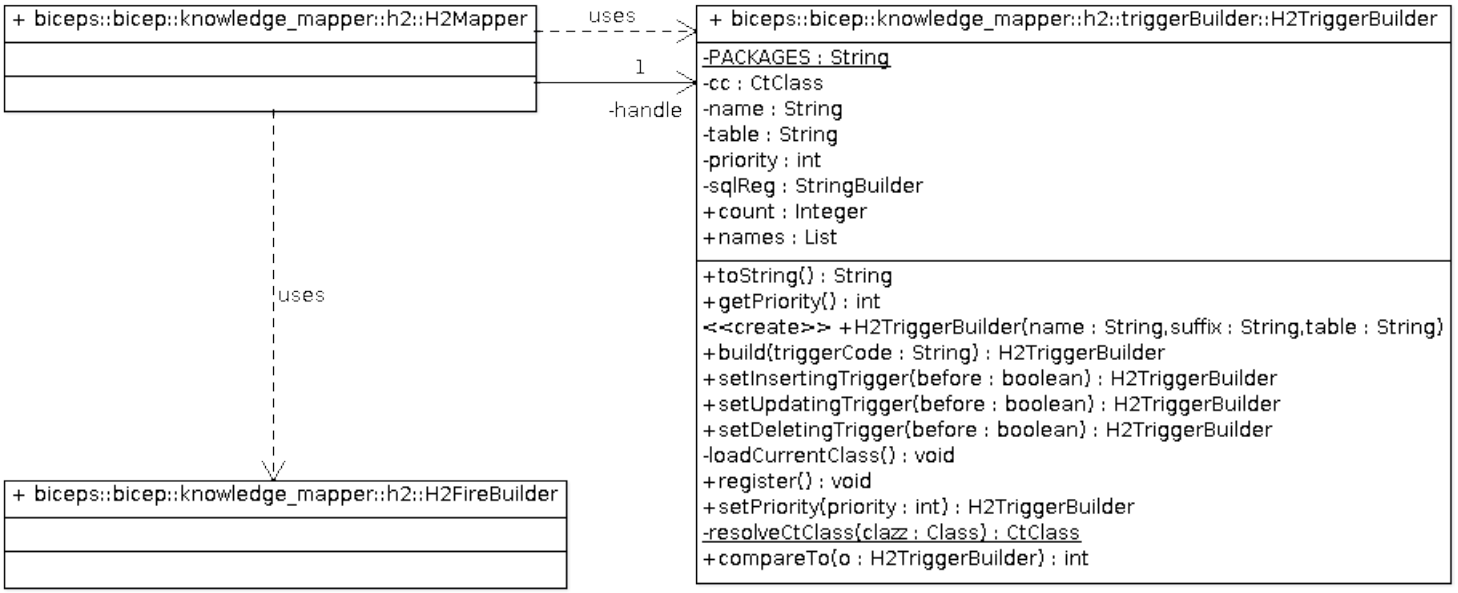


Figure A.10: The UML class diagram of the H2 trigger building classes and their relationships with `H2Mapper`. For the full diagram of `H2Firebuilder` refer to Figure A.11.

Figure A.11: The UML class diagram of the `H2FireBuilder`.

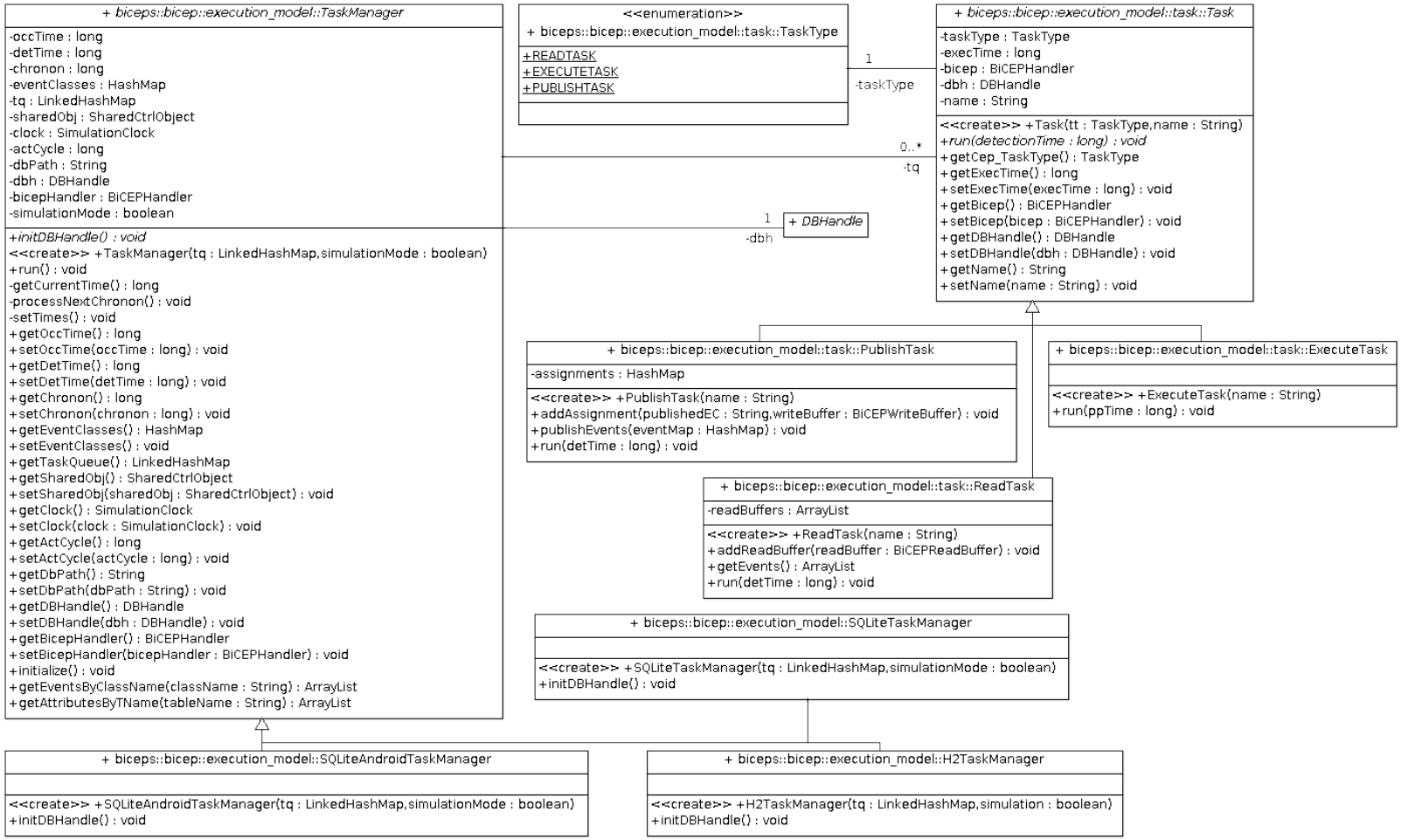


Figure A.12: The UML class diagram for the execution model of the CEP.