ulm university universität uulm

Institute of Distributed Systems
Ulm University

# Enabling Retroactive Computing Through Event Sourcing

Michael Müller

Master's Thesis
VS-M01-2016

Reviewers
Prof. Dr. Frank Kargl
Prof. Dr. Franz J. Hauck

Advisor
Dipl.-Inf. Benjamin Erb

# Abstract

Event sourcing is a style of software architecture wherein state altering operations to an application are captured as immutable events. Each event is appended to an event log, with the current state of a system derived from this series of events. This thesis addresses the utilization of retroactive capabilities in event-sourced systems: computing alternate application states, post hoc bug fixes, or the support of algorithms which have access to their own history, for example. The possibility of retroactively accessing and modifying this event log is a potential capability of an event-sourced system, but a detailed exploration how these operations can be facilitated and supported has not yet been conducted.

We examine how retroaction can be applied to event-sourced systems and discuss conceptual considerations. Furthermore, we demonstrate how different architectures can be used to provide retroaction and describe the prototypical implementation of an appropriate programming model. These findings are applied in the Chronograph research project, in order to utilize potential temporal aspects of this platform.

*This thesis was written by Michael Müller in the winter semester of 2015/2016 for the completion of the degree Master of Science. It was written at the Institute of Distributed Systems at Ulm University and was submitted on March 31, 2016.*

Michael Müller
michael-4.mueller@uni-ulm.de

Institute of Distributed Systems
Faculty of Engineering, Computer Science and Psychology
Ulm University
James-Franck-Ring
89081 Ulm, Germany

# Preface

This thesis is my own work, but without the discussions with friends and appropriate feedback it would certainly be in a different shape.

Firstly, I want to thank my advisor *Benjamin Erb* for coming up with the initial idea and his in every way valuable advice. Our discussions have been an opportunity for me to reflect on ideas and conceptual considerations. Without those, the results in this thesis would surely differ.

The remarks by *Markus Schnalke* provided great feedback of a reader unfamiliar with the topic. Thank you, Markus.

*Katja Rogers* had a look at the thesis and provided valuable suggestions on language and style. Thanks, Kate.

The group meetings of the *Chronograph research group* provided a forum for discussing ideas. *Gerhard Habiger* provided further valuable suggestions. Thank you all.

Last, *Valerie* deserves recognition for her support and understanding. Something that is equally important as technical advice.

Still Hänsel comforted Grethel, and said, »Wait till the moon rises; and then I shall be able to see the crumbs of bread which I have ſtrewed, and they will show us the way home.«

—Grimm's Household Tales
*Hänsel and Grethel*

# Contents

# Chapter 1

# Introduction

## 1.1 Motivation

*Event sourcing (ES)* describes a style of architecture wherein each state altering operation in an application is captured as an immutable event [1]. These events are appended to an *event log*. Once appended they can never be deleted, the event log possesses a write-once, append-only restriction. The current state of an application is derived from this series of events. When replayed, this series of events always leads to the exact same state of the application.

Systems following such an event-sourced architecture have been gaining popularity in recent years. There are a number of reasons for this; among them is that event-driven architectures enable loosely coupled, composable systems and possess an inherent asynchronous style [2]. As such they fit well for e.g. distributed systems with a need to scale. A distributed architecture which applies event sourcing can utilize the immutable nature of events and the append-only semantics of the event log beneficial for scalability. Event sourcing also fits well for systems with the requirement of an audit log, since events are immutable and the event log possesses an append-only restriction. It can always be clearly traced how a system reached its current state.

This thesis focuses on the *retroactive* aspect of event-sourced systems – exploring the past of existing or simulated event logs and the possibilities which can be gained hereby. Manipulations of the past contradict the append-only behavior of the event log, but can enable very interesting applications. Among other applications, the recorded history can be replayed in order to evaluate how other events or an alternate application behavior would have influenced the current state of the system [3]. These retroactive possibilities are potential capabilities of event-sourced systems, but a practical exploration of how these operations can be enabled and supported has not yet been conducted.

## 1.2 Overview

Restoring prior states of a system is already a well-established feature of event-sourced architectures. This thesis examines another very interesting aspect of this architectural style, which we think has not yet been considered sufficiently: Event sourcing is a perfect match for a computing model in which programmers can modify and interact with the application's history retroactively in the same environment in which the program runs.

We aim to provide a thorough examination of the retroactive capabilities of event-sourced systems. This work identifies issues and constraints that need to be taken into consideration when utilizing these capabilities. Furthermore, we identify benefits which systems can gain from retroactive features. The practicality of our ideas is analyzed in a programming model and a prototypical implementation. In order to gain further insights into the applicability of our ideas, we apply them in an existing research project.

## 1.3 Outline

In Chapter 2, we describe the terminology and background of event sourcing and describe work related to this thesis' objectives. Chapter 4 focuses on a conceptual view on the topic, identifies challenges, and details our ideas. For this, we describe an event-sourced architecture following Command Query Responsibility Segregation (CQRS), a style of system architecture well-suited to event-sourced systems. Next, we examine how retroactive computing features can be integrated into such an architecture. A number of issues and limitations, which need to be taken into account for this, are identified and we suggest two potential event-sourced architectures for retroactive computing. In Chapter 5, we outline a programming model which utilizes retroactive capabilities of an ES+CQRS architecture and takes retroactive issues into account. This programming model is implemented as a prototype with an example scenario. Chapter 6 summarizes our architectural considerations. In Chapter 7, we examine the applicability of our ideas in an event-sourced architecture that does not follow CQRS. The Chronograph research project is the foundation for this examination. Chapter 8 provides our concluding remarks and closes the thesis.

# Chapter 2

# Background and Terminology

This chapter details event sourcing and related terminology. We provide an overview on benefits and constraints of event-sourced systems and examine in which cases event sourcing can be applied beneficially as a style of architecture. Furthermore, we describe Command Query Responsibility Segregation (CQRS), a complementary architectural style for event-sourced systems, and illustrate which benefits can be gained from applying it.

## 2.1 Active Record

Many software systems require some kind of data management. A common approach is to retain the current state of an application within some kind of data storage (e.g. a database) and return it on request. *Active Record* is a software pattern which refers to the interaction with this current state (i.e. the active record) by applying create, read, update, or delete operations (CRUD) on objects in a relational database (Figure 2.1a) [4]. In this approach, only the current state of an object is maintained and manipulated; previous values are overwritten and everything not saved is lost. This approach fits well for many applications but has shortcomings in terms of traceability and support for operations on the history.

## 2.2 Event Sourcing

Most literature on event sourcing can be found online in blog posts, presentations, and software documentation. Academic literature on the topic is scarce. This section provides an overview on how event sourcing is defined in different resources and establishes the vocabulary used throughout this thesis.

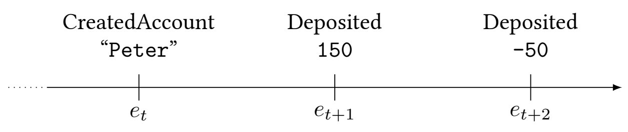
### 2.2.1 Different Definitions

There is no standardized vocabulary on the topic, the terminology and definitions vary depending on the author.

**Martin Fowler** The term event sourcing was first established in a blog post by Martin Fowler [1] in 2005. He describes events as a series of changes in the state of an

| Name | Balance |
|---|---|
| Peter | 100 |
| ... | ... |

(a) Active Record Example



(b) Event Sourcing Example

Figure 2.1: Two different ways of storing the same current state.

application. This series of events captures everything required to rebuild the current state. He views events as immutable and the event log as an append-only store. Events are never deleted, the only mean to revoke an event is a retroactive event [5]. Once appended to the event log, this retroactive event acts as an inverse event to a prior event and revokes its effects. Fowler does not distinguish clearly between events and the commands (i.e. actions) which triggered them. This is an issue which is addressed in various works [6, 7].

**Greg Young** Greg Young is another renowned author in the event sourcing domain. He describes event sourcing as "storing [the] current state as a series of events and rebuilding state within the system by replaying that series of events" [8]. In his view, the event log possesses an append-only behavior as well: "[events] have already happened and cannot be undone" [9]. What Fowler calls retroactive events, Young describes as reversal transactions [9, p.31].

**Udi Dahan** Udi Dahan is a further author of a number of relevant blog posts and articles concerning event-sourced systems. To him, event sourcing refers to "the state of the domain model being persisted as a stream of events rather than as a single snapshot [...]" [10].

**Martin Krasser** The articles and presentations by Martin Krasser concerning the persistence module in the Akka toolkit provide another view on event sourcing [11, 12]. In this context, actors in a distributed system communicate via messages, which trigger state changes. Event sourcing is used as the mean to persist changes to the state of an actor. State changes are "appended as immutable facts to a journal" [11]. A motivation for using it is that this approach "allows for very high transaction rates and efficient replication". Recovering the state of an actor (e.g. after a restart or crash) is done by reapplying (i.e. replaying) the persisted events.

The common ground of all these definitions is to publish every change to the state of an object (or system, or application) as an immutable event to an append-only event log. This results in a series of events, which when replayed always leads to the exact same state of the object (Figure 2.1b). Individual definitions vary or make no statement in terms of the editing semantics of the event log, the granularity of events, and the motivation for using it.

Where traditional architectures retain and maintain only the current state of an object, event sourcing maintains all state-altering operations. Where CRUD offers four operations to apply modifications, event-sourced systems are restrained to only one operation: append-only.

### 2.2.2 Benefits of Event Sourcing

An often found recommendation in literature is to apply event sourcing only in a clearly delimited part of a system and not force its usage upon a context where it does not appear beneficial [10]. A commonality often found in event-sourced systems, is that they possess a complex business logic. This is opposed to e.g. an application that provides "just" an editing frontend to a relational database. We now briefly describe some of the benefits which an event-sourced architecture can provide.

**Audit Log** In regulated areas (e.g. the financial industry), government regulations in many countries require companies to maintain records on the operation of a system. For example, the governmental regulations in the United States require brokers to keep their records in a non-rewriteable, non-erasable format [13]. Event sourcing, with its append-only event log and immutable event characteristic, fits this requirement very well. Technologies such as write-once-read-many (WORM) data storages can be used complementarily with event sourcing. If a WORM storage is used, the device prevents alterations of data in the hardware and allows only the appending of new data.

**Debugging** The captured events can be used to gain further insights into how the system reached its current states and which events were responsible for it. This is a strength in terms of traceability and debugging capabilities, since it makes it easier e.g. to retrace where a bug originated in a system.

**Scalability** The append-only nature of an event log is thought to be beneficial for scalable architectures. In such architectures it is common to have multiple replicated instances of the same data model. These replications need to be kept in synchronization, in order to provide a consistent view on the data. In event-sourced systems, the appending of events is the only mean to synchronize the replicated instances. Due to fewer necessary locks, an append-only architecture with immutable events is thought to scale easier for reads than e.g. an architecture with a need for model synchronization via updates [14, p.14-15][15].

**Informative Value** Another motivation for applying event sourcing can be the informative value (also sometimes referred to as "business value"[1]) of retrospectively examining the event log. All past states of a system can be reconstructed or queried. This can provide great value to systems, especially when the analysis of interaction with the system (e.g. customer behavior) is important. In such systems it is usually unknown what analysis one will execute in the future. An example to illustrate this is an online shop where a shop administrator wants to list all products which have at some point been removed from a shopping cart. In event-sourced architectures, the state of the shopping cart at each past point can be queried.

There are a number of documents describing how event sourcing has been applied in real-world systems. To highlight two information-rich documentations: The financial trading platform LMAX utilizes event sourcing [16] and a documentation published by the Microsoft Windows Azure team describes several case studies [10]. Furthermore, event-sourced frameworks like Akka[2], Event Store[3], or the Eventuate[4] toolkit by the Red Bull Media House provide the foundation for a number of real-world systems.

[1]http://docs.geteventstore.com/introduction/event-sourcing-basics/#business-value-of-the-event-log
[2]http://akka.io/
[3]https://geteventstore.com/
[4]https://rbmhtechnology.github.io/eventuate/

### 2.2.3 Thesis Terminology

Now that various authors' understanding of event sourcing has been described, we establish the terminology used throughout this thesis. Since authors in the field use varying terminology, our terminology may differ from the individual authors' understanding.

**Event** An event is a change to the state of an application. Each event is appended to the event log. It is important to note that the state change has already occurred to the entity. To reflect this, we name events as verbs in the past tense (e.g. `ProductCreated`). The granularity of events is defined through the context in which event sourcing is applied. Events can be replayed in order to rebuild the state of a system up to a certain point. Except for the first event, each event builds upon the previous event. A replay process always needs to start from the beginning, though snapshots may be used as substitutions.

**Snapshot** Snapshots are an established mean to optimize the process of restoring state. They represent the state of the system at a certain point in the event log and can be used instead of replaying the events prior to the snapshot.

**Event Log** The event log provides the stream of all events which belong to an application. An exchangeable term for the event log is the *event stream*. Figure 2.2 depicts an excerpt of a hypothetical event log.

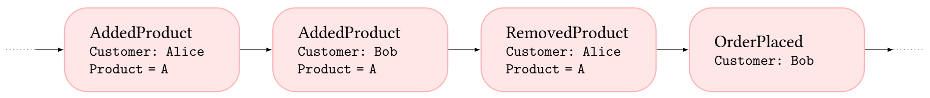


Figure 2.2: An excerpt from a hypothetical event log in an online shop.

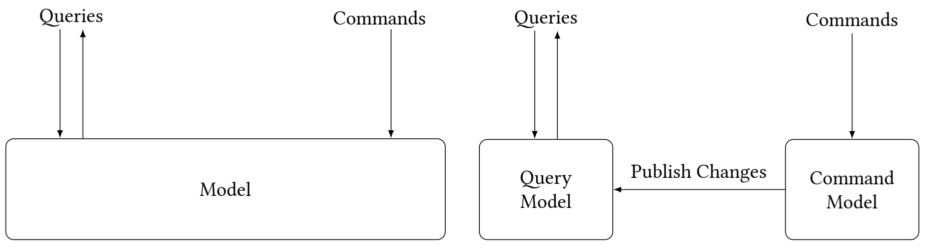
**Command** In the context of this thesis, a command resembles a requested change to an application. A command always results in an event, even in case of a failure. Commands are typically named in the imperative form (e.g. `CreateProduct`).

**Retrospection** We use the term *retrospection*, to describe the passive (i.e. read-only) access of a system's history and prior states.

**Retroactive Computing** With *retroactive computing* (or *retroaction*), we describe the interaction with an application's history. This includes applying passive retrospective operations, as well as actively modifying a history. Modifications can be applied through removal or insertion operations, as well as a retroactive change of the underlying application logic (the source code). This is examined in further detail in Chapters 4 and 5.

## 2.3 Command Query Separation (CQS)

The Command Query Separation principle was first described by Bertrand Meyer [17, p. 747] with the intention of improving the handling of side effects when creating a program or designing an API. A core idea behind this principle is to split access to objects into (1) *Queries*, which return information and (2) *Commands*, which modify state (Figure 2.3a). Queries should not produce side effects. An often-mentioned analogy is that a question should never change the answer.



(a) Command Query Separation (CQS) (b) Command Query Responsibility Segregation (CQRS)

Figure 2.3: Conceptual depiction of both patterns.

## 2.4 Command Query Responsibility Segregation (CQRS)

Command Query Responsibility Segregation is a pattern which was first described by Greg Young [9]. In the past, some resources have described it as an extension or special case of the Command Query Separation principle, but today it is considered a separate pattern with an origin in the CQS principle. CQRS states that a different model (i.e. component or object) is used to fulfill queries than the model used to fulfill commands (Figure 2.3b). CQS splits responsibilities on the code level, by dividing methods into queries and commands; CQRS takes this further and even divides objects into two kinds: reading or writing. Young has described this as follows: "CQRS is simply the creation of two objects where there was previously only one" [8].

Building an architecture following the CQRS pattern often introduces an *eventually consistent* behavior into the system. Since the query and command model are separate components, depending on their coupling they might not necessarily be in synchronization. If they are loosely coupled, the query model does not necessarily contain the same data as the command model at every point in time. This characteristic that the query model might contain a temporarily inconsistent (i.e. outdated) state, which at some point in the future "eventually" converges to a consistent state, is referred to as eventual consistency. This eventually consistent behavior becomes especially relevant in distributed systems, when the query model(s) are physically separated from the command model(s). Brewer's CAP theorem [18] postulates that a distributed system inevitably has to decide how to handle consistency, since it can only ever ensure two of the following three guarantees in case of a network failure:

1. *Consistency*: All participants view the same data.
2. *Availability*: Each request receives a response.
3. *Partition Tolerance*: The system continues to operate, even if message loss occurs between arbitrary participants of the system. This affects consistency, since participants need to cope with the loss of messages. It also affects availability, since each participant needs to be able to fulfill requests.

Thus, if a distributed system is built using CQRS as a style of architecture and chooses partition tolerance and availability over consistency, the system can only provide *eventual* consistency. Hiding the fact that the system behaves eventually consistent (e.g. by imposing optimistic assumptions) can be dangerously misleading, since developers and software components would then operate under false assump-

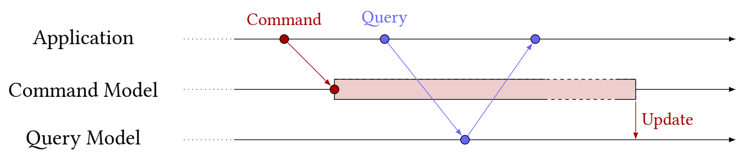


Figure 2.4: In a CQRS architecture, a command and a subsequent query do not necessarily possess a causal relationship, since commands are asynchronous and it is unclear when they are executed and finished. In the example illustrated above, the query model receives the status update from a finished command after the query has already returned. This leads to an eventually consistent semantics for queries.

tions. A progressive handling of consistency issues can be seen as a strength of a distributed ES+CQRS system, since it forces developers to take this very consistency behavior – which is such an integral part of the overall architecture – into account when developing an application. Queries might at all times return an outdated result and it is unclear when commands are executed. This progressive attitude to issues of large distributed systems has spawned a certain kind of attitude and a number of projects in recent years. The *Reactive Manifesto* [19] is a noteworthy document which surfaced in 2013. It summarizes key properties behind distributed system design, which correspond to event sourcing and CQRS.

A disadvantage of command and query segregation is that it yields a higher complexity when building a system and makes it hard to create a causally dependent series of commands and events. Since commands behave asynchronously and do not return a value, the application has no means of knowing when the result of a command is visible. Thus, a query following a command might return an outdated status. Subsequent queries, however, will at some point eventually return a consistent status. Figure 2.4 illustrates a typical sequence of commands and queries. There are possibilities to circumvent this behavior – delaying the fulfillment of queries until the query model is updated to a certain version number for example (as used in conditional requests in Eventuate[1]). But it should be noted that these are mitigation techniques which should only be used in rare, special cases since they come at the cost of efficiency and work against the characteristics of these systems. Instead it is desirable for the application to be inherently able to cope with an eventually consistent behavior.

## 2.5 Combining ES and CQRS

CQRS is a pattern which forms a symbiotic relationship with event sourcing and as such is often used in event-sourced architectures. The classification of operations in state-mutating commands and read-only queries aligns well with the ideas found in event-sourced systems. In CQRS, commands represent intentions, which are sent to a command processor. There they are processed and yield state changes – the events. Events are then appended to the event log and published to query models, where they eventually update the state as well. Query models fulfill read-only requests and can provide a certain view on the data (e.g. a table of accounts and their balances). Events are used as the mean to synchronize the models. Thus, commands are then clearly

[1]https://rbmhtechnology.github.io/eventuate/userguide.html#conditionalrequests

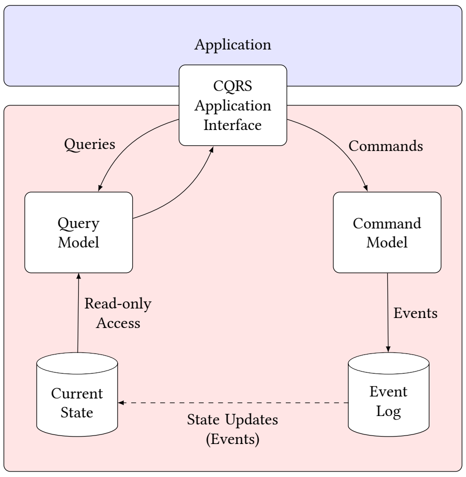


Figure 2.5: Architecture of a system which uses event sourcing in combination with CQRS. This illustration is based on a figure from [3].

separated from other operations which do not result in an append operation to the event log. Figure 2.5 illustrates a typical ES+CQRS architecture: the business logic resides in the high level application layer; commands and queries hide implementation details of how they are executed. Figure 2.6 depicts a conceptual view with focus on the internal workings.

The command/query segregation results in characteristics that work well with event sourcing. Since events are immutable and are only appended after a command has been executed they are the mean to keep data models in synchronization. The query models only need to receive new events in order to stay consistent with the command model. By splitting concerns on an object level, individual optimization of each model is supported. This is a crucial property for systems with a requirement of scalability, since the optimization techniques for reads and writes differ. Segregating the command and query model makes it feasible to scale them independently. One technique to optimize for reading operations in a distributed system is to replicate the database. These database replicates then have to be kept in synchronization, but each of them can then respond to requests. A contrasting possibility to optimize for writes, on the other hand, is to reduce redundancy (and thus writes) in a database. This can be achieved by organizing a single database accordingly, using e.g. normalization rules. Further details on the combination of event sourcing and CQRS can be found in a work by Hauser [20].

## 2.6 Domain-driven Design (DDD)

Both, event sourcing and CQRS, are closely related to the idea of domain-driven design. The terminology and ideas behind DDD were described by Eric Evans [21] in 2003. At its heart, DDD provides a set of guiding principles and building blocks for software development. The core idea here is to put the focus of a software project on the domain and the tasks within it. A central part when designing a system is the creation of a domain model. Among other components, this model describes tasks and events of the core domain. This domain model is created in collaboration with

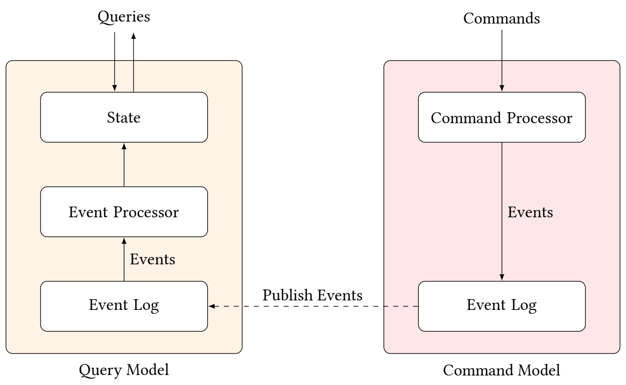


Figure 2.6: Conceptual view of an event-sourced architecture which takes advantage of the CQRS style.

domain experts. In this regard, DDD is similar to agile development models, where the collaboration with domain experts (or customers) is encouraged as well. DDD demands the definition of a common language, the *ubiquitous language*, to be used throughout the project. This ubiquitous language is thought to build a common view on the domain model amongst team members. Verbs and nouns in the language promote a clear understanding of the tasks which need to be fulfilled by the system. A common language is also thought to prevent misunderstandings and enable anyone in a project to talk comprehensibly to any other person in the project, thus bringing domain experts and developers together.

Applying DDD only in this bounded context as well as the usage of high-level concepts to abstract from low-level details offer commonalities with the event sourcing field. In an event-sourced system, one usually does not persist all low-level occurring events in a system, but rather only relevant domain events in a delimited context. DDD is thought to be suited well for systems with complex business rules and a clearly defined and delimited domain. In a simple system, however, DDD may increase complexity to an unnecessary and counterproductive level.

The ideas behind event sourcing and CQRS have developed out of DDD. Commands and queries in an ES+CQRS system can be modelled using DDD. This way they have a clear meaning in the domain model and are familiar to developers and domain experts. The same holds true for the events which are created – as domain events they fit into the domain model and its logic. A domain expert describes commands and queries and their internal logic; an application developer can then access these commands and queries over an API to implement an application. Since commands and queries abstract from domain-specific conditions, working with the system is made easier. An application developer does not have to take into account how a command is validated or know how it is implemented internally. Complex business logic is abstracted through the usage of simplified commands.

## 2.7 Summary

In this chapter, we detailed the concept of event sourcing. We laid out the terminology used in this thesis and described the Command Query Responsibility Segregation

(CQRS) pattern, a common style of architecture for event-sourced systems. Furthermore, we examined the symbiotic relationship formed by CQRS and event sourcing.

Although we described some of the benefits which event-sourced systems can provide, it is important to note that event sourcing is seldom applied to an overall system. Indeed, an often found recommendation is to apply this style of architecture only within a clearly defined context. The same recommendation is often found for CQRS, since the command/query model segregation comes with the implication of higher complexity and often an eventually consistent behavior. Thus, applying CQRS as a style of architecture is only beneficial when the system has requirements for CQRS properties, such as individual optimization of the read and write model.

# Chapter 3

# Related Work

This chapter provides an overview on related efforts to the objectives of this thesis. We aim to provide a clear distinction of this thesis' goals to existing works.

## 3.1 Event Sourcing

The idea of retroactively modifying existing event logs in order to observe alternate system behaviors is not unmentioned in the event sourcing field. Fowler describes a concept he calls *retroactive events* [5] – appending a special type of event to the event log to denote a retroactive change. What Fowler calls retroactive events, Young describes as reversal transactions [9, p.31]. Synonymous terms for this concept used by other authors are undo or anti events. Fowler also describes a number of benefits which systems could gather from retroactive capabilities: posteriori bug fixes or additional code which executes time-dependent logic, for example.

Potential modifications to the event log have also been mentioned by Erb and Kargl [3]: By deleting, inserting, or modifying events one may alter the current state of the application or retrieve former states; one could also alter the application logic (the "source code") at a past point and explore how the application would have behaved differently. There are other individual resources which mention potential retroactive capabilities of event-sourced systems as well; a conference presentation by Greg Young stands out among these [22].

But the scarce resources which touch on these retroactive capabilities offer only ideas or remarks on potential capabilities. The mentions often only refer to purely event-sourced systems, which do not persist commands. The benefit of sourcing commands as well is often not addressed. To the best of our knowledge, no comprehensive examination and exploration of retroaction in event-sourced systems has been conducted yet. This is the niche we address in this thesis.

## 3.2 History-aware Algorithms and Languages

History-aware algorithms utilize information about the past of a system. Noteworthy works in this direction are a history-aware self-scheduling algorithm [23], and a history-aware adaptive routing algorithm [24].

Exposing the history of variables as a programming language feature has been addressed by Proebsting et al. [25]. This work describes a programming language with language primitives which allow access of variable history. In the following example, the “`<x>`” exposes the sequence of values which were assigned to `x`.

```
print(”average %f\n”, sum<x>/length<x>);
```

The intention here is to reduce bookkeeping code (such as counting e.g. invocations of a certain function), in order to reduce errors, work, and overhead. The authors have filed their idea of program history in a computer programming language as a patent [26]. These approaches, however, only enable passive retrospection in the programming language and do not consider retroactive alterations of the application's history, nor the exploration of experimental execution paths.

## 3.3 Self-improving Algorithms

Self-improving algorithms in their original sense learn from the data upon which they work, in order to better perform their operation [27]. Such algorithms exist for a broad number of tasks: sorting, clustering, or scheduling, for example. The objective of most of these algorithms is to optimize individual parameters of an algorithm, with the logic itself remaining hardwired. Exploring entirely different algorithms, comparing experimental branches, issues of consistency, or side effects are not of subject in most of these algorithms.

An outstanding example of a self-referential and self-improving algorithm is the Gödel machine [28]. It rewrites its complete internal logic once a mathematically sound proof is found that another logic would perform better. When doing this, it is ensured that the examined logic is a globally optimal maxima rather than a local maximum. The Gödel machine is settled in the research area of *artificial general intelligence*, where further algorithms with learning abilities can be found. Neural nets, for example, can be iteratively fine tuned for machine learning purposes through the usage of new training data. There are other fields which utilize the idea of self-improving software as well. In graphical user interfaces it is quite common to adapt the interface based on the usage. An easy possibility to do so is to display frequently used functions first whilst hiding unused features. In large distributed systems, load balancers can be used to improve the overall system performance by spawning new virtual machines, in order to better cope with upcoming load[1].

## 3.4 Reversible Debugging

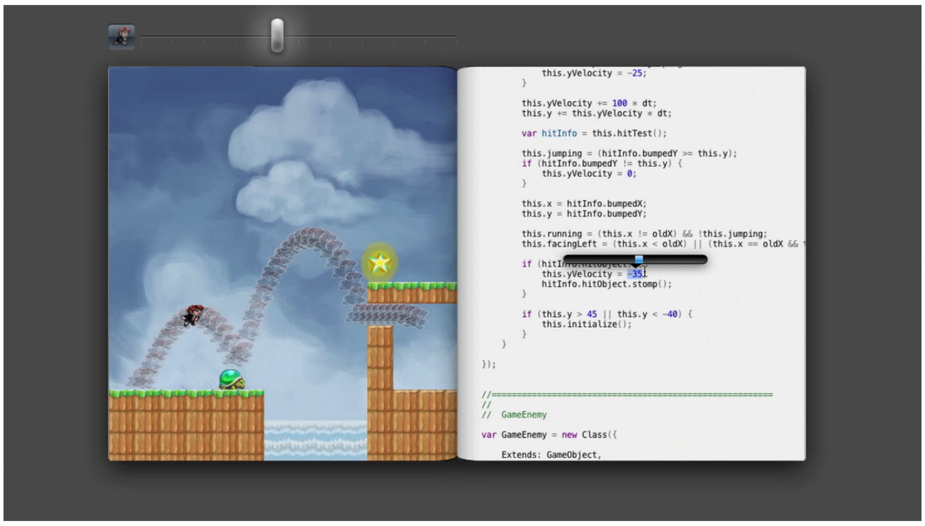
The concept of tracing a system for later analysis can be found in many areas. Execution traces of programs are commonly used for debugging or testing purposes. Tools for debugging software allow for interactive control over the execution of an ongoing program. These tools commonly only support stepping through the program execution, but there are some debuggers which also aim for the ability to step back. This concept of reverse debugging is also described as *omniscient debugging* [29] or *back-in-time debugging* [30]. One example of a debugger which allows for reverse debugging is the GNU Project debugger `gdb` [31]. Whilst a process is executed, registers and memory locations which are modified by individual machine instructions are

[1]https://aws.amazon.com/elasticloadbalancing/

recorded. In order to undo an instruction, these registers and memory locations are reverted. An insightful description of this process can be found in the GDB documentation [32]. Stepping backwards and forward is deterministic, as long as side effects do not trigger deviations from this process. As we view it, the influence of side effects is mostly ignored by reversible debuggers. There are individual tools which support recording side effects though [33].

An experimental graphical user interface which utilizes history-aware debugging was presented by Bret Victor [34] in 2012. Figure 3.1 depicts a screenshot of this interface. Above the left screen, a timeline slider is available. Using this element, it is possible to step through a program's recording. On the right side of the screen, a parameter which controls the jump strength of players is modified. The left side of the screen depicts the current program execution and extrapolates how this tweak would have influenced the further jumping curve of the player.

Delimitations of reversible debugging to our approach are that debugging is a decoupled process which happens in a separate environment. Furthermore, access to the program's history is not possible within the program environment (the source code) itself. Our objective is to examine an integrated system, where applications may modify and interact with their history in a single environment.



```
            this.yVelocity = -25;
        }

        this.yVelocity += 100 * dt;
        this.y += this.yVelocity * dt;

        var hitInfo = this.hitTest();

        this.jumping = (hitInfo.bumpedY >= this.y);
        if (hitInfo.bumpedY != this.y) {
            this.yVelocity = 0;
        }

        this.x = hitInfo.bumpedX;
        this.y = hitInfo.bumpedY;

        this.running = (this.x != oldX) && !this.jumping;
        this.facingLeft = (this.x < oldX) || (this.x == oldX &&

        if (hitI[illegible]
            this.yVelocity = -35;
            hitInfo.hitObject.stomp();
        }

        if (this.y > 45 || this.y < -40) {
            this.initialize();
        }
    }

});


//=====================================================
//
//  GameEnemy

var GameEnemy = new Class({

    Extends: GameObject,
```

Figure 3.1: An experimental graphical debugger, which allows for tweaking the recorded history of an application. The image (taken from [34]) depicts how the modification of a movement variable affects the recording.

## 3.5 Retroactive Aspects

Retroactive aspects have been described by Salkeld et al. in various works [35, 36]. Their work settles in the area of aspect-oriented programming (AOP). AOP has some advantages over an object-oriented approach. For example, it can be cumbersome to realize crosscutting functionalities in object-oriented programming models. Examples of crosscutting functionalities are logging or authorization features which affect multiple objects. A typical object-oriented approach to handle crosscutting functionality in multiple objects, is to call the e.g. authorization methods within each object. In AOP, on the other hand, crosscutting functionalities can be implemented by creating a new *advice* for an additional authorization functionality. Next, it is configured which methods this advice applies to by specifying *pointcuts*. The term *weaving* refers to the process of producing an executable program with the source code and the aspects "weaved in".

The authors aim to analyze and manipulate recordings of a program's execution post hoc (i.e. offline). For this analysis, they aim to extend an existing AOP language to support retroactive evaluation semantics. Thus, the analysis code is written in the same environment as the program. The authors make the argument that this eases the creation and evaluation of analyses for average developers, whose expertise lies in the language in which the program was written, rather than in the usage of external tools. Additionally, developers can use the familiar data types and functions from the program in the analysis code as well. They refer to their primary goal as "allow[ing] analysis code that is as close as possible to the program's source code". The term *retroactive weaving* refers to the evaluation of retroactive aspects "in relation to a previous execution", meaning retroactive aspects are weaved into an existing trace. *Retroactive advice* describes the analysis, replay, and modification of former states. From our understanding, retroactive aspects describe the interplay of retroactive advice and retroactive weaving used to conduct post hoc analysis of a recorded program execution.

The authors recognize the issues of dealing with side effects in retroactive contexts: "We are continuing to investigate the best approach for managing side effects. It is unclear how to rigorously permit intentional, safe side effects in retroactive advice: outputting results to the console or the file system should be permitted, for example [...]". For the analysis code they propose to avoid side effects, for the application code they suggest the usage of an operator which enable the suppression or redirection of side effects from retroactive code.

## 3.6 Retroactive Data Structures

Retroactive data structures record the series of operations which have been performed on them. It is then possible to retroactively insert, delete, or update an operation in the past. These data structures were first described by Demaine et al. [37], with a focus on how to implement them efficiently. The authors highlight a broad number of applications which can benefit from using retroactive data structures. One example which they describe consists of weather stations which lose their connection to a central control system. Once they are reconnected, their data needs to be fed back into the system. Ideally this is done retroactively, as if the stations had been connected the whole time.

The authors compare retroactive data structures to persistent data structures and highlight the differences: In persistent data structures, several versions of an object are held and modifying the object is achieved by applying an operation on the latest version. Retroactive data structures, on the other hand, focus on inserting, deleting, or updating operations which have been executed on an object *in the past*, rather than on the latest version.

Any data structure can be retroactively manipulated by rolling a data structure back to a prior state, apply a change, and replaying forward again. However, the authors argue that this often leads to inefficient manipulations with an expensive overhead. They examine if arbitrary data structures can always be transformed to a data structure with support for *efficient* retroactive manipulations and prove that it is not possible to find such a generic transformation. Realizing efficient retroaction depends on the use case: they describe a number of data structures which support efficient retroactive modifications (e.g. priority queues). The authors furthermore distinguish between *partially* and *fully* retroactive data structures. Partially retroactive data structures enable the insertion and deletion of operations in the past. Queries are limited to the present time. Thus the outcome of retroactive operations can only be observed at the present time. Fully retroactive data structures additionally allow the specification of arbitrary times in the past when placing a query. The result of the query will then be as if it were executed at that time in the past. The delimitations to our objectives are that we consider interactive event-sourced applications, opposed to data structures with no behavior by themselves.

## 3.7 Version Control Systems

Version control systems (VCSes) or source code management (SCM) tools, like Git or Subversion, provide capabilities for managing changes to files. Once a file is committed to the repository, a revision with this certain state is created. If an altered version of the same file is committed to the system, a new revision is created. The current state of a file is derived from this series of commits. Additionally, VCS tools allow changes to be traced and revoked, as well as the creation of branches. These branches contain the entire history of the repository up to a certain branching point. It is then possible to apply experimental changes in the branch, without affecting the main (or "master") branch. Results can be integrated back to the original branch by merging two branches. Most modern VCSes, however, provide much more than this basic set of operations (bisecting[1] the occurence of a bug, for example). Many modern VCSes save just the delta (the "diff") to the previous revision; the current state is thus defined by this series of deltas.

VCSes are typically agnostic in terms of the files which they administer in a repository. They are decoupled tools which operate on a meta level – they do not have any coupling with the files which they administer. Also the files in the repository do not have access to their own history, nor operations to manipulate their own history. A commonality with the event sourcing domain is that the repository can possess append-only semantics as well and the series of commits defines the current state. Thus, retrospective operations, such as restoring and analyzing previous states, are possible. Additionaly, a typical VCS provides features which this thesis aims to examine in event-sourced systems: creating branches and merging changes back to a main branch.

## 3.8 Time Travel Theory

Time travelling is a concept which describes the notion of moving forward and backward through time. The field has been subject to works of fiction, philosophy, and extensive scientific research. There are many parallels to the topic of retroaction in event-sourced systems: The event log can be considered a timeline and retroactive

[1]https://git-scm.com/docs/git-bisect

changes can be considered time travel. Furthermore, in time travel theory, a small change in the timeline can have enormous, unforeseeable changes as a consequence as well. An extensive summary of logical and philosophical questions on the topic was compiled by Smith [38]. A number of interesting hypothetical questions (e.g. "Where are the Time Travellers?") are discussed there as well.

Time travel theory can provide helpful insights on issues of retrocausality and causality violations which might occur. Causality violations in time travel theory occur when we observe an effect before its cause. One such issue concerns the emergence of temporal paradoxes, which can lead to an event log that is in contradiction to itself. Consider the *grandfather paradox*: one goes back in time to a point before one's own father was conceived and eliminates one's grandfather. This leads to a paradox: once the grandfather is dead, one could never have been born and never have travelled back in time to murder one's own grandfather. This is a logical paradox, each scenario contains its own exclusion. Transferred to an event-sourced system, a comparable scenario is an object which removes its producer from the log at a point before it was created.

But issues of retrocausality do not always require family members to die. Another problem of retrocausality concerns *causal loops* which emerge when one event is cause for another event, which is a retroactive cause for the first event. Or as the physicist Michael Rea puts it: "a causal loop is a causal chain in which at least one of the events in the chain lies in its own history" [39]. Without further considerations, it cannot be determined what originally triggered the event and causal loops can be responsible for loops in which "things come from nowhere" [38]. Various examples of causal loops have been constructed by philosophers, physicists, and fiction authors. These examples often have a similar pattern and the presumption of a single timeline (often referred to as a "closed timelike curve"). One such example is a time traveller who travels back with the copy of a mathematical proof. The traveller reveals this proof to the mathematician who made this very proof, at a time before he did it. Thus the information seems to come from nowhere. When computed, causal loops can create infinite loops. They can also break the traceability of a system, since it is not always clearly determinable how a system reached its current state. We examine these issues of causalities in retroactive systems in further detail in Chapter 4.

A number of theories postulate possibilities to avoid paradoxes. One such theory concerns *parallel universes*. Here, each time travelling action yields a new, parallel, universe. If the grandson eliminates his grandfather in this new, parallel, universe, his action will not impact his conception in the original universe. The elimination happened on a different timeline, thus there is no paradox. This idea is also commonly described as "many-worlds interpretation". Another possibility is the *self-consistency principle* [40, p.607], which was developed by Igor Novikov to solve the problem of time travel paradoxes. It postulates that actions which would lead to paradoxes simply cannot be executed – similar to the laws of physics which restrain our influence on the real world. In works of fiction this self-consistency is sometimes depicted as a restriction of the free will (one just cannot act in this way) or by creative avoidances, such as that one's grandfather was not actually the biological grandfather.

Whether time travel is possible remains disputed. As we view it, the scientific and philosophical consensus seems to be that time travel is assessed as unlikely. The argumentation often follows the lead that time travel raises a number of issues which cannot be explained with current scientific or philosophical understandings.

## 3.9 Other Related Domains

In the area of discrete-event simulation (DES), a system is modelled as a sequence of discrete events. Here, events are defined as "*instants in time when a state-change occurs*" [41]. Such a simulation is executed by tracing the path from one event to its consecutive event(s). State is represented as discrete events over time with the current state being derived from this series of events. This concept has a number of parallels to the ideas behind event sourcing. In fact, there are so many commonalities between the two research domains that it is possible to combine them. In a work by Erb and Kargl [3], both approaches are compared; they also describe a hybrid architecture.

There are a number of further research domains and technological fields with parallels to retroaction in event-sourced systems. Database management systems often possess transaction logs and rollback methods for fault or crash recovery [42]. Journaling file systems, such as ext4 or JFS, record the state mutating operation for crash or fault recovery. Message logging and checkpointing is another concept used in distributed systems for failure or crash recovery. Besides the already described retroactive data structures, there are more data structures with history support. The concept of unlimited undo or backtracking [43] is such an example. The blockchain technology [44] used in the Bitcoin cryptocurrency (and many other applications) has parallels to event-sourced systems as well. The blockchain utilizes cryptography to achieve an immutable property for blocks. In conjunction with the append-only behavior of the blockchain, this can be beneficial for scalability and traceability (audit log). A multitude of NoSQL databases view modifications of documents as incremental state changes, which are committed to an append-only data storage. CouchDB is such an example [14].

## 3.10 Summary

In this section, we described a number of related research domains and technological fields. We highlighted that a lot of the ideas behind event sourcing and retroaction can be found in other areas as well, mostly under a different terminology. Most systems, however, use a post hoc analysis approach, in which the analysis tool is in a separate environment, tool, or programming model. When the history of a system is utilized, this is mostly done by applying only passive retrospection, without retroactively altering the application's history.

The objective of this thesis is to examine an integrated system, where an application may modify and interact with its history in a single environment. To the best of our knowledge, the idea of using an event-sourced architecture to enable retroactive computing features has not yet been researched, nor examined within a prototypical implementation.

# Chapter 4

# The Conceptual Model

This chapter examines the ways in which retroaction can be applied to event sourcing. We discuss conceptual considerations and illustrate how different architectures can be utilized to enable retroactive capabilities of event-sourced applications.

## 4.1 Retroactive Computing

In order to examine how retroactive capabilities can be utilized in event-sourced systems, we explicitly examine event sourcing *in a CQRS context.* As described in Chapter 2, event sourcing and CQRS form a symbiotic relationship which is commonly taken advantage of. By assuming such a style of architecture, we can focus on how the event sourcing foundation can be utilized. This is not to say that we do not examine other architectures. Indeed, in Chapter 7 of this thesis we examine if and how the concepts described in this (and the succeeding) chapter can be applied to a different architecture. The Chronograph platform is the foundation for this second examination; it applies event sourcing in a context which has only few commonalities with the traditional CQRS style of architecture.

## 4.2 Conceptual Considerations

This section provides an overview on key challenges when utilizing retroaction in event-sourced systems. We propose possibilities of how to handle these issues, but do not claim to lay out the one true way in which retroactive computing should be done in event-sourced systems. We rather aim to illustrate that depending on the context of a system, different conceptual choices are best-suited. We deliberately restrict the context in which we examine retroaction and focus on three example scenarios of event-sourced applications with different requirements and issues. The topic of retroaction is very extensive and appointing these three scenarios as cornerstones of our examination provides a guideline and helps us illustrate complex issues.

**Weather Station Application** In this scenario, sensor readings are fed into the application via commands. These commands result in events, which do not possess causal relationships among each other. Based on this data, the application generates statistics and forecasts. The scenario is not complex and poses only few issues for

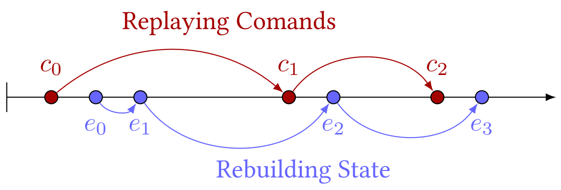


Figure 4.1: When commands and events are sourced, this results in a timeline of commands and subsequently their resulting events.

retroaction. Examples of retroactive operations in this scenario are the correction of faulty values, the retroactive addition of missing values, and the reconstruction of historic statistics and predictions. Comparing the outcome of different forecast algorithms based on the historic data is another use case.

**Internet of Things Controller** The application manages several independent Internet of Things (IoT) devices. These devices send status information to the controller, which maintains a global view on all devices and issues new commands to them. The global view of the controller is based upon the status information which it receives from the devices; commands are issued based on this constructed view. As such, this scenario possesses a close coupling between the application and its environment: The system state affects the environment, which in turn affects the system state. Retroactive use cases here include the analysis of incidents. Not only can incidents be reconstructed, changes to the controlling logic can be simulated as well. Furthermore, the historical application data can be used to optimize the control algorithm.

**Online Shopping Service** The online shop combines warehousing, order processing, pricing, and accounting. By utilizing retroaction, prices or inventory errors can be corrected retroactively. Simulations of changes to discount algorithms or price calculations can be performed based on past transactions. This scenario is heavily affected by side effects (e.g. sending order confirmations), hidden causalities (e.g. the impact of prices on actual orders), and a strong coupling (e.g. effect of orders on article availability).

### 4.2.1 Command Sourcing

In a strict event-sourced system, only events are sourced, but not the commands which triggered their creation in the first place. The current state of a system derives from this series of events and can at any time be rebuilt by applying each event successively again (starting from the first event). It is important to note that in such an event replay no commands are invoked, since the events are not newly computed. This is opposed to a command replay, in which each command is invoked again. As a result, the commands may yield the same events or different ones, but the events have been newly computed. A command replay poses a central issue to retroactive computing: If a command is invoked again, side effects might occur again. An event-replay, on the other hand, does not trigger any side effects, since only existing events are used to rebuild state.

By sourcing commands and events, the retroactive capabilities gain in expressiveness and a number of applications are possible, which are not feasible in a system which sources only events. For instance, it is possible to test how a system would

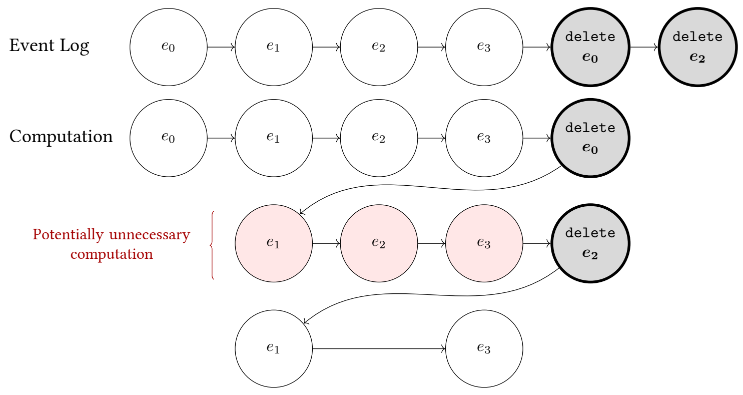


Figure 4.2: This figure depicts a possible course of computation when events are appended to denote retroactive modifications.

have reacted differently, if a different logic (i.e. a different command processing) had been in place. In order to evaluate how a different command processing would have affected the behavior of a system differently it is necessary (1) to source commands and (2) for the command processing to be decoupled from the command. Decoupling the command processing from commands can be modelled by a command processor component which receives a request to invoke a certain command, executes appropriate processing logic, and produces events (Chapter 2). One can then use the recorded set of commands and replay them with a different processing implementation. In relation to our three scenarios, commands can be modelled as sensor data streamed into the system (weather station, IoT) or as orders and inventory updates (online shop).

For the remainder of this thesis, we focus on systems which source commands and events. We refer to the log of such systems, the combination of command and event log, as the *timeline*. Figure 4.1 depicts an example of a timeline. Here, a command in the timeline can result in one or more events. If we write of “events” resulting from a command (as opposed to the singular “event”), we imply that the result could also consist of a single event.

### 4.2.2 Editing the Event Log

The append-only nature of the event log in event-sourced systems creates a lot of the advantages which one usually aims for with an event-sourced architecture (see Section 2.2.2 for a detailed explanation). For retroactive commands, this append-only restriction implies that retroactive changes to the event log need to be appended as well. We mentioned Fowler’s retroactive events [5] as a mean to apply retroactive changes to the event log. These retroactive events are appended to the event log to denote that the effect of a previous event should be revoked by modifying the *current* status in a way as if a certain event had never occurred. In the case of the online shop, a retroactive event which fixes a false inventory update could set the inventory stock back to a prior number. It is important to note that the originally false event remains in the event log. The usage of these retroactive events is not motivated by retroactive explorations of insert or delete modifications to the event log, the intention rather is to add fixes to the log. Fowler proposes three basic types of retroactive events for this: out of order events, rejected events, and incorrect events.

Retroactive events suit well for typical productive event-sourced systems, in which it is crucial that events are only ever appended and never deleted or retroactively injected. Otherwise a lot of the event sourcing characteristics (e.g. benefits for scalability, traceability, or the possibility to rebuild application state at arbitrary points in the timeline) would no longer hold. The motivation behind the research in this thesis, on the other hand, is to explore how modifications to the past of a timeline affect the current state of a system. Appending reversal events is not an appropriate tool for this purpose, since it cannot always be easily calculated how the current state of the system would differ. This calculation would be equivalent to resetting the system to a previous state, applying modifications there, and re-playing events (or replaying commands) from that point on. A series of retroactive changes would then lead to expensive and potentially unnecessary computations being executed (Figure 4.2).

A contrary possibility to apply retroactive modifications, is to directly edit a timeline without appending commands or events to the timeline. We refer to such retroactive modifications as *retroactive operations*. These are the insertion or deletion of commands (or events), as well as the exchange of command processing logic. Even though direct editing violates the append-only behavior of the timeline, this does not necessarily contradict the principles of an event-sourced system. We propose to apply two different views here. By distinguishing between a *main timeline* and *branches* of this timeline, we can utilize the advantages of both options. The main timeline resembles the application itself and retains the append-only behavior. It cannot modify its own past. It is though possible to create branches of this main timeline. Branches are copies of a timeline. They were created at a certain time and contain everything that was in the timeline at that point in time. Modifications on them do not affect the original timeline. In contrast to a timeline, they possess a direct editing semantics. If results from these branches are to be returned into the main timeline, this can be done using event sourcing primitives – by appending events.

This decoupling of "productive" system and retroactive explorations already prevents obvious issues of retroactive systems: A paradox could emerge once the application modifies its own timeline in an inconsistent way. This can happen once it introduces a contradiction to the current state into its own past – e.g. by modifying its history in a way that it could never have executed such a modification. Moreover, keeping the append-only characteristic for the main system flow does not break with benefits of event-sourced architectures for e.g. scalability and traceability. The following example illustrates this issue with a delete operation. For this, we assume that a system receives commands which influence and trigger the purchase of stock options. In the example, we use numbers as unique identifiers for commands (`c`) and events (`e`), but this is only for illustration purposes. Events usually do not have an incremental identifier.

```
c0: someComputation(): // ...
e0: buyStock = false

c1: newData(...): if (data.price < 100) buyStock = true;
e1: buyStock = true

c2: checkBuying(): if (buyStock = true) buyStock();
e2: boughtStock = true

c3: delete(”c1” and ”e1”);
```

The resulting timeline looks like this:

```
c0: someComputation(): // ...
e0: buyStock = false

c2: checkBuying(): if (buyStock = true) buyStock();
e2: boughtStock = true

c3: delete(”c1” and ”e1”);
```

It is no longer clear why stock was bought in the first place. In some corner cases, this may get clearer if the command model only possesses one possibility which could have triggered `buyStock()`, but this cannot be generalized.

In the case of insert operations the traceability of a system breaks as well. In the following example it can no longer be clearly traced what originally triggered the purchase, if an event “`bar = true`” is inserted after `e0`. There are certainly means to circumvent this behavior (e.g. saving timestamps with events), but this then still works against the append-only behavior of the event log.

```
c0: foo = false;
e0: foo = false

c1: if (foo == false || bar == true) buyStock();
e1: boughtStock = true
```

If a system applies retroaction only to branches and appends the results from retroactive computations on branches to the timeline using command/event primitives, then it is still possible to trace how certain states of the timeline occurred. From our point of view, Fowler's retroactive events are the only appropriate mean to apply retroactive operations on the main timeline without breaking traceability or the append-only property. Thus, in the case of the weather station, they are an appropriate tool to correct faulty values or retroactively insert later ones. Branches, on the other hand, fit very well to conduct e.g. experimental retroactive analyses and return their result to the application. In the IoT scenario this could e.g. be the analyses of an experimental device controller. For the remainder of this thesis, we focus on the direct editing *of branches*. As illustrated, it fits better to the objectives of retroaction in our context. Furthermore, we use the term *retroactive queries* when referring to queries which operate on a branch.

### 4.2.3 Consistency and Validation

An implication of directly editing a branch is that whenever a change is introduced into the past, these changes affect the subsequent part of the branch. Thus, in order to ensure a consistent branch, editing operations need to be complemented by further measures. Otherwise causal violations (i.e. a paradox) could occur when a contradiction is introduced into the branch. For example, an excerpt of the online shop's event log could look like this:

```
c0: Create Product A, Inventory Stock = 10
e0: Created Product A, Inventory Stock = 10

c1: Order 10 Items of Product A
e1: Ordered 10 Items of Product A
```

An example of an illegitimate change to this example would then be to inject the event “`Ordered 1 Item of Product A`” between `e0` and `e1`, since this would result in a negative inventory and depending on the domain model a negative inventory might be prohibited. The event `e1` would thus not be consistent with the domain model after this particular retroactive modification. There are a number of options how the consistency of a timeline can be ensured. In the related work chapter on time travel theory (Chapter 3), we already mentioned two possibilities to prevent paradoxes: the usage of parallel universes and the self-consistency principle. We now describe how the self-consistency principle can be applied to retroaction in event-sourced systems:

**Removal of Causally Dependent Events** Causally dependent events can be recursively removed, by forward deleting all events which have been directly or indirectly influenced by a specified event. After this cleanup process, all consequences of an event have been removed from the timeline. In order to realize this, causal dependencies between events need to be recorded. In the online shop scenario, this can be used to prevent paradoxes. When e.g. an event `PlacedOrder` is removed from the timeline, all causally dependent events (e.g. `ParcelShipped`) can automatically be removed as well. The concept of tracing causal dependencies among events will be detailed in the next section. Limitations of this approach are discussed there as well.

**Replay of Causally Dependent Events** A similar approach to ensure consistency is to recompute all events from the point of the retroactive change on. This can be achieved by reprocessing the commands belonging to these events, which ensures that succeeding events were computed with these retroactive changes taken into account. Transferred to time travelling, this means that everything happens again, though it might happen differently this time. This possibility requires capturing causal dependencies as well. Its limitations will also be detailed in the next section.

**Validation** The self-consistency principle from time travel theory can also be applied to restrain the possibilities of retroactive modifications. Invariants and pre-/postconditions are possibilities to validate the timeline and ensure its consistency. If the validation of a retroactive modification fails, this indicates that the timeline would no longer be consistent and the modifications thus are not allowed. Invariants fit well with the DDD approach of describing a domain model and the constraints within it. In the online shop, the invariant could describe domain-specific constraints (e.g. that product stock has to be a positive number) and event dependencies (e.g. that in a shopping cart, `ProductRemoved` events need a foregoing `ProductAdded` event). The challenge with validation conditions is that they can only be checked *after* the execution of a command. Only then it is clear in which events the command results. If the validation succeeds, the command and event can be appended to the timeline. If it fails, appending the command and event to the timeline can just be discarded. The state of the overall system then remains unchanged. But these validation mechanisms pose a challenge for a command yielding side effects. Then discarding the command and event from appending is not enough, since the side effects have already occurred. This issue and a possible solution is discussed in Section 4.2.5.

It can also be appropriate not to constrain modifications to the timeline at all. Inconsistent manipulations of the past might even be a fully intended effect. A use case for this is to define a certain application state as a desired target state. As a next step, the application's past is modified until the target state is reached. This process yields the necessary retroactive modifications as a result. Thus, constraining the set of possible operations on the timeline by imposing validation conditions and constraints might

– depending on the context – even weaken the retroactive capabilities. An expressive concept can allow to specify the validation behavior when manipulating the timeline: validate the timeline for consistency (e.g. by supplying an invariant with the intended modification) or execute the modifications with full knowledge of possible inconsistencies.

### 4.2.4 Tracing Causal Dependencies

With each operation, which modifies the past of a timeline and is not append-only, a nontrivial problem emerges: The state of the system from this point on might no longer be correct, since the reprocessed commands could yield different events. Thus, the succeeding commands would have had a different premise for their execution and they might have yielded different events as a result. Thus succeeding events might be causally dependent on prior events. But it is not sufficient to only take these *directly* affected events into account. Commands or events which have *indirectly* built on these direct events, may then have had a different premise as well. This trace of causal dependencies affects events which are directly and indirectly affected by a modification. Mathematically spoken, these are the events in the *transitive closure* of the modified event. In the previous section, we described the concept of self-consistency for a timeline by either recursively recomputing causally dependent events or by recursively removing them. This recursive process can lead to a cascade of rippling replays (or removals) which work through the timeline.

The concept which we just described lacks the information *how* it is possible to trace which commands are affected by a state change. This trace can be described using event sourcing primitives: For each command it needs to be clear what information it used to get to its result, i.e. upon which events it built to calculate the events which it yielded as an output. This approach enables the creation of a trace for each command, tracing which commands are affected by a retroactive modification, and recursively recompute them. A concrete algorithm for this is detailed in Section 5.2.1. A limitation of this process is that hidden causalities outside of a system might exist (e.g. through side effects). It is not possible to capture those and thus not all commands necessary might be replayed. This limitation is described further in Section 4.3.1.

We see multiple possibilities how the information upon which events the commands have built can be recorded. One possibility is for developers to manually annotate this information and e.g. return it from the command. Another option is the usage of getter and setter methods which record access to objects. In some cases, this annotation could also be done automatically through an underlying platform (e.g. utilizing source code analysis).

### 4.2.5 Side Effects

Dealing with side effects is an important and nontrivial issue, which needs to be solved in order to fully utilize the retroactive capabilities of event-sourced systems. Certainly one can prohibit applications to yield any side effects, but as other authors [45, 35] have recognized, many interesting applications are only possible because of side effects (e.g. input/output or networking).

In the context of our scenarios, the weather station application possesses no side effects. The application only gets data into the system via commands and its interaction with the outside system is limited. The IoT scenario and the online shop, on the other hand, possess a number of side effect afflicted operations: sending order confirmations to customers, validating payment details using external services, or issuing control commands to devices, for example. When examined closer, it becomes clear that the problem has two parts to it: *validation and control*. We examine these two issues subsequently.

### Validation of Side Effect Afflicted Commands

The timeline can only be validated for consistency after a – possibly side effect afflicted – command has been executed. Only then it becomes clear, in which events the command results. If this validation fails, the event is not persisted and the question arises how the already invoked side effects within the command can be revoked. For purely reading side effects this is not a problem, since they do not mutate state outside of the system. But for state mutating side effects this problem needs to be handled. One could see the solution in adding adding undo (or rollback) commands, which would reverse these side effects. But this idea hits upon problems as soon as the side effect is irreversible – which is the case with e.g. order confirmations (online shop) or actions sent to devices outside of the system (IoT). A better possibility is to delay the execution of state mutating side effects until the result of the command has been validated and it is certain that the resulting events are appended to the timeline. This approach has limitations as well, as soon as e.g. internal logic depends on their prior invocation. But it at least is a possibility to validate side effect afflicted commands.

### Controlling Side Effects in Replays

The second challenge concerns the question how the behavior of side effects can be controlled when a side effect afflicted command is replayed. When state is rebuild, this can be done by examining the series of events in the timeline (Figure 4.1). There are no side effects from such an event replay, since events denote only changes in state and their replay does not trigger their recomputation. But side effects can be caused by replaying (i.e. reprocessing) commands. A problem here is that one cannot find a general rule of how to handle side effects in a replay. If commands are replayed with a purely experimental intention, it can be undesirable to have some of the side effects triggered again. An online shop which replays customer interactions on the new backend for pure testing purposes will not want to have order confirmations sent out again. Console output or logging facilities, on the other hand, might be desirable side effects of such a replay for testing purposes.

Handling this ambiguity of side effects when working retroactively is an issue which Salkeld et al. [35] have described: "We are continuing to investigate the best approach for managing side effects. It is unclear how to rigorously permit intentional, safe side effects in retroactive advice [...]". Since the treatment of side effects is highly domain and context dependent it needs to be possible to specify how a side effect should behave when it is replayed. There are two behaviors which we aim for:

1. Reinvoke them and use the new result for the further processing (e.g. fetching a web page again).
2. Suppress their new invocation and reuse the result from their first invocation (e.g. reusing a generated random number or reusing the result from a prior dispatching of a mail message, instead of sending it again).

One could see the solution in adding the possibility for the command processing to know if it is currently replayed. If the command processing can be exchanged before a replay takes place, this would enable developers to specify how specific side effects within a command should be handled. This would solve some problems with side effects, by providing some form of control over them. But it does not solve all issues with side effects. For example, a command might have had a side effect which returned a value. In a command replay scenario it might be desirable for the side effect to return the same value as it did when executed for the first time. For example, in the online shop scenario a `PlaceOrder` command could have an internal credit card validation logic which uses an online service. If one were to replay the commands in the timeline, in order to find out if an optimized warehousing algorithm would have reacted better to inventory changes, it would be desirable to have the side effect return the same validation results as it did in the "original" recording. Otherwise it would be hard to compare the results of both algorithms. For this purpose, the result of the validation requests needs to be recorded somehow. Enabling commands to know if they are currently replayed does not solve this issue. Additionally, for commands to know if they are currently replayed contradicts a deterministic behavior of replays. A replay could then yield entirely different results, each time it is conducted. In order to gain full control over the behavior of side effects in replays, it is necessary to record the result of side effect afflicted operations.

A concept proposed by Fowler, as a mean to handle side effects in event-sourced systems, is the usage of gateways [1]. He describes gateways as special objects which encapsulate side effect operations [4, p. 466]. The behavior of these gateways in a replay is defined by the application which triggers the replay. An issue here is that it is unclear how gateways should record the operations which pass through them. Additionally, gateways do not suit well for the concept of branches: they would need special attention and handling when creating or replaying branches, since they do not tie in with the common event sourcing primitives.

Before we describe a possibility for handling these issues further, let us first examine what types of side effects can occur in an event-sourced system. There are three kinds of side effects which may occur in commands in event-sourced systems (Fowler uses a similar categorization [1]):

- *External Query*: Reading data by interacting with an outside system. In the online shop scenario this can e.g. be the reading of a currency exchange rate from an online service.
- *External Command*: Mutating data in an outside system. In the IoT scenario, the controller sends commands to devices outside of the system. These commands are irrevocable, as soon as they leave the system. Sent order confirmations in the online shop are a further example.
- *External Query + Command*: The combination of both. An example is an external payment service used by an online shop. The shop could request a payment, as part of a process to place an order. A notification could indicate if the payment was successful or failed.

For external queries in a replay scenario, it should be possible to specify if they should either be executed again or if the result which was originally returned should be reused. Reusing an already existing result implies that the new invocation is skipped. External commands are supposed to be strictly state mutating and to not

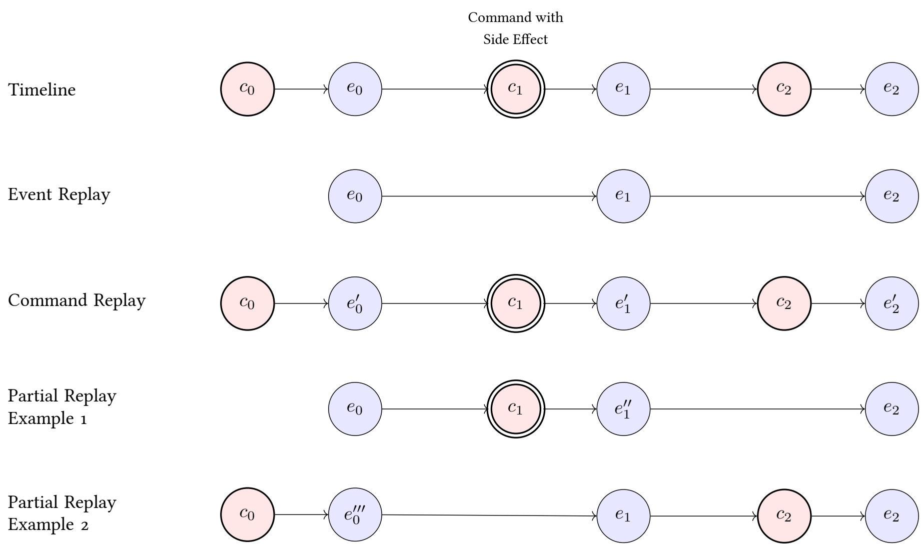


Figure 4.3: The figure depicts different possibilities for replays of a timeline. In an event replay, state is rebuilt based solely on events. In a command replay, each command is invoked again and events are newly computed. In a partial replay, it is possible to control which commands are re-executed. In "Example 1", the side effect afflicted command is re-executed. In "Example 2", the side effect afflicted command is not executed again, instead the event from the last invocation is reused.

return any state. It should be possible to specify if they should either be executed again or not be executed again (skipped). It should also be possible to handle the combination of both by specifying re-executions or the reusage of prior results.

To achieve this, we propose an idea which ties in with the characteristics of an event-sourced system – namely the event log and sourcing commands and events for restoring state or reprocessing commands. The concept of *partial replays* enables us to control the behavior of side effect afflicted commands in a replay. Partial replays form a combination of event and command replay. Here it is possible to specify the behavior of individual commands and events in a replay by controlling:

1. If a persisted event should be used instead of invoking the command again.
2. If a command should be re-processed.

Different replay sequences are depicted in Figure 4.3. The figure though only depicts small sequences. A problem which becomes visible in larger sequences, is that through the reprocessing of a command, a different event (or multiple different events) than in the original computation may be computed. The state of the system from this point on would then be different. Thus, subsequent commands may have executed a different logic, which possibly would have yielded other events. Section 4.2.3 discussed two solutions to this problem: recomputing causally dependent events as well or removing causally dependent events.

We propose to leave control over how fine-grained side effect afflicted commands can be controlled to the programmers and domain experts. We therefore propose to split side effect afflicted commands into multiple commands, with the side effects being isolated in individual side effect afflicted commands with few logic. In partial replays, all three types of side effects result in command-events pairs with no further distinction if they have a reading or writing side effect, or none at all. Side effects then are encapsulated in individual commands and events. This concept enables us to view side effects as isolated and thus we can control their individual behavior in a replay. It is then possible to e.g. specify that results of a command `SendOrderConfirmation()` should always be reused. Thus the order confirmation side effect will not be invoked again. This also enables us to specify that e.g. a command `FetchStockPrice()` is invoked again. As a consequence, all events which are causally dependent on this result can be reprocessed as well (i.e. the respective commands are reinvoked). This can be achieved by examining the trace of causalities among events as described in Section 4.2.4. But this concept implicates that it needs to be possible to split commands. Three possibilities how this can be achieved are:

1. For the application, which issues commands to the API, to split a command into an ordered series of multiple, smaller commands. These commands are issued sequentially. Each command in the series needs to complete, before the subsequent command is issued.
2. For the API to provide a thin logic layer which splits an incoming command into multiple internal commands which are invoked sequentially, one after another.
3. For the command processor to invoke other commands during the processing of a command.

The implication of (3) is that the definition of the timeline as a series of atomic command and succeeding affiliated events would break up. Instead the timeline would consist of nested series' of commands and their respective events:

$$(command_a \rightarrow (command_b \rightarrow event_b) \rightarrow event_a)$$

Such nested series' break with the event sourcing primitives of pairs of commands and event(s). We thus do not consider (3) further. The requirements for (1) and (2) are:

- When executing a series of commands, this series is fixed. The component which issues the individual commands (the API or the application) does not deviate from this series. No further logic is executed during the execution of the series. Otherwise the determinism of replays is endangered, since this "logic layer" would not be persisted in the timeline and could not be taken into account in replays. Thus, replays might not perform the operation as it would have really occurred.
- Each command is persisted in the timeline, independent of its success or failure. This enables later replays with possibly different results.
- The execution of this series is done sequentially (or in a causally equivalent, serializable order). Each command has to finish its execution before the subsequent command in the series is invoked. After each command returns, it needs to be checked if the command succeeded. If it did, the command and the resulting event are persisted to the timeline and the next command in the series is invoked. If the command failed, the commands in the series will not be invoked further. But the commands will still be persisted to the timeline, in order to enable later command replays.

Without those requirements, a causally equivalent (i.e. deterministic) replay of persisted commands in the timeline is no longer possible. This issue is detailed further in Section 4.3.4.

We now describe a brief example of this concept. Consider a hypothetical command `ValidateCreditcard` as part of the logic behind the online shop scenario. The command authorizes a credit card using an external service, which it accesses over a web service:

```
CommandModel.process("ValidateCreditcard"){
  /* side effect */
  var validation = fetch("https://.../check-card", ...);
  if (validation == true) {
    return Event("OrderPlaced");
  } else {
    return Event("OrderCanceled");
  }
}
```

This command can be broken into two succeeding commands, `ValidateCreditcard` and `PlaceOrder`. The `ValidateCreditcard` command executes the side effect afflicted operation. This results in an event being persisted and the side effect effectively becoming part of the state of the system:

```
c0: ValidateCreditcard()
e0: ValidationSuccessful

c1: PlaceOrder()
e1: OrderPlaced
```

The `PlaceOrder` command can then execute logic based on the fetched validation result. In a replay it can be controlled if the side effect should be executed again (i.e. the command reprocessed) or if the already persisted stock price should be used.

The advantage of such a series of (possibly side effect afflicted) commands and their events, over the concept of gateways is that they specify clearly how the results of side effects are recorded and how their behavior can be controlled when a replay is executed. Also this concept ties in with event-sourced primitives: commands, events, and the timeline. This comes in helpful when we use the concept of branches on the timeline, since we do not have to pay special considerations to gateways and their behavior when branching. As detailed earlier, this splitting can be done by the API or by the application. As long as the component adheres to the described requirements, a deterministic replay is possible.

### 4.2.6 Separated Query and Command Model

When retroactive explorations are conducted, the goal usually is to determine *what* the current state would be *if* the past would have differed. Such *what-if questions* typically consist of a similar sequence of actions: a series of operations on the timeline and a concluding query (or multiple queries). An example of such a series is:

1. *Branch* timeline at a certain point in the past.
2. *Insert* an event at a certain point in the past of the newly created branch.
3. *Query* current state of the branch.

What-if questions are hard to realize in a standard CQRS architecture, since they require to first execute a series of operations (in commands), before a subsequent query can be deposed. A separated command and query model with an eventually consistent semantics works against such a causally dependent series of commands and queries (Chapter 2). Without further measures, an application cannot safely assume that the branch and its modifications have already been replicated to the query model(s). Because of this eventually consistent behavior, it is entirely possible for an application to address a query model where not all what-if modifications have yet been applied to a branch. As described in Section 2.4, an eventually consistent behavior can be seen as a positive artifact of a CQRS architecture. For what-if questions, however, this behavior is disadvantageous, since a causally ordered behavior fits better to series' of causally dependent operations and queries. A possibility to address this, is to mitigate the eventually consistent behavior and delay the fulfillment of queries until the query model is updated to a certain version number (a concept used for conditional requests in Eventuate[1]). Another option is to dissolve the model segregation for retroactive operations and retroactive queries, so that they operate on the same model. Causality can then be enforced through the model. A separated retroactive model for retroactive operations and queries offers the same benefits as a separated model for application commands and queries: individual optimization of each model and the creation of applications with inherent eventual consistency in mind. On the other hand, what-if questions are easier for developers to write, if retroactive operations and retroactive queries can be issued causally dependent. Developers then do not have to take on the task of ensuring that queries are only issued, once all retroactive operations have been applied.

## 4.3 Constraints and Limitations of Retroaction

Before we address retroactive architectures further, limitations and constraints of retroaction are described in this section. The degree to which they apply depends on the actual context of an application. We do not claim to have solutions for them, they should rather point out the limitations of retroaction.

### 4.3.1 Hidden Causalities

In an ideal scenario, no hidden causalities exist in an event-sourced system. Thus, when a retroactive change is made to the timeline and a subsequent replay is conducted, the outcome is exactly as it would have been if the change had originally been in place. But this ideal scenario is not the case for many systems, as they have some coupling to the real world or to other systems. A danger in the informative value of retroaction lies in hidden (i.e. untracked) causalities of events. One event may influence – directly or indirectly – the occurrence of subsequent events. This is even more problematic if these causalities occur outside of the event-sourced system, since they then cannot be tracked. These hidden causalities might weaken the informative value which can be derived from the results of retroactive changes.

To illustrate this on the online shop scenario: The retroactive manipulation of the inventory could result in only three car tires being available for purchase. This would affect existing car tire order commands in the timeline, since most customers would

[1]https://rbmhtechnology.github.io/eventuate/userguide.html#conditionalrequests

probably only have issued an order if *four* tires had been available. The issue can get even more complex. Maybe there were only three tires in the inventory all along and through retroactive changes there are now four. A customer might have taken this opportunity to place an order of four tires. On the other hand, a car dealer might have ordered four tires with the intention of supplying them as spare tires to four different cars, one for each. Ordering three tires would have still been fine then. Without knowledge about these causalities in the real-world, the system cannot distinguish these cases. The influence of prices on placed orders is a similar issue. Thus, caution should be exercised when deriving informative value from retroactive results. One should at least take domain knowledge and the possibility of causalities outside of the system into account.

These issues are smaller, when an outside system does not further process the results of queries. Furthermore, an outside system should not save any state based on the result of queries. This can otherwise lead to a (possibly outdated) copy of the data outside the system. If the outside system makes decisions based on a copy of the data, the informative value and expressiveness of retroaction is heavily reduced. This issue is especially relevant when the event-sourced system is used as a component within a larger system. Such shadow state caches in the application are discouraged. Hidden causalities can also occur through side effects. These effects take place in an outside world and their behavior may be influenced by factors outside the scope of the event-sourced system.

### 4.3.2 Causality Violations

A further danger of retroaction lies in retroactively induced causality violations, such as paradoxes or causal loops. Causality violations occur when we observe an effect before its cause. In Section 3.8, we have described temporal paradoxes where scenarios contain their own negation (grandfather paradox). In event-sourced systems, a logical paradox can e.g. emerge once an object removes its own producer at a point before it was created. We have described the emergence of domain model contradicting paradoxes in the previous section (e.g. placing orders when an item is out of stock). Another problem of retrocausality concerns causal loops. Causal loops emerge when one event is cause for another event, which is a retroactive cause for the first event. Without further measures, it can then no longer be determined which event was the initial cause. If not handled properly this can break the traceability (audit log) of event-sourced systems and when computed can lead to infinite loops. Figure 4.4 depicts an example of a causal loop.

These issues are a direct consequence of breaking with the append-only behavior of the timeline – they emerge when the own past is modified. Issues of retrocausality limit the possibilities of retroaction and can be source for a number of problems. Nevertheless, the possibilities of retroaction can enable powerful retroactive capabilities and there are options to limit the impact of these issues. Obvious issues of retroaction can already be prevented by *allowing retroactive modifications only on branches of the own timeline.* This resembles the many-worlds interpretation from time travel theory. Instead of a single timeline, retroactive modifications yield a new, parallel, universe. In the context of retroaction in event-sourced systems, we proposed to keep the append-only semantics for the main timeline and feed results from retroaction back using commands/events (Section 4.2.2). This also complements event sourcing characteristics of traceability and scalability, which emerge from an append-only restriction.

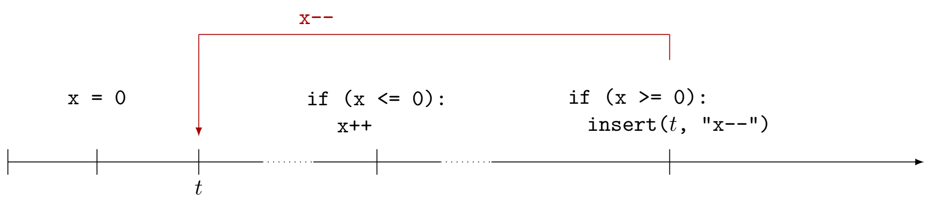


Figure 4.4: A causal loop is caused when one event is cause for another event, which itself is cause for the first event.

### 4.3.3 Command Semantics

When a command has an effect independent of the current state of the system, this can have an annihilating effect on retroactive modifications. For example, a command `EmptyShoppingCart` might remove all products from the shopping cart. Thus, the effects of a retroactively injected command `AddProductToCart` prior to an `EmptyShoppingCart` command will be overwritten. This can lead to unintended reversals of retroaction. This is not necessarily a limitation of retroaction, but it can have an unintended limiting effect if not taken into account.

### 4.3.4 Causal Equivalence of Replays

In terms of performance, it can be desirable to optimize a command replay (using e.g. parallelization in threads or distributed processing). But this cannot be achieved trivially by e.g. arbitrarily distributing the command processing among threads. Due to differences in e.g. thread scheduling or network transmission times, a replay could then result in a non-deterministic order of processing. If the persisted order is deviated from in a replay, the resulting events can break the deterministic cause of subsequent events. Thus, an entirely different application state may emerge each time a replay is conducted. To illustrate this on the online shop scenario: If `PlaceOrder` commands are issued in a different order, items in the replay could at different times be out of stock instead of available. Orders which have previously been executed successfully could then suddenly fail in a replay, due to a different order of processing.

A causally equivalent replay also makes it feasible to compare the impact of different command processing implementations. For this, commands are replayed in a serializable order with a different implementation. The results can then be compared against each other. In a non-deterministic replay, on the other hand, unintended variations in e.g. processor utilization could as well be responsible for the difference in the outcome.

Thus, an event-sourced platform applying CQRS and supporting retroactive computing capabilities needs to either (1) provide the guarantee of determinism for replays by issuing commands in a serializable order or (2) consciously decide to break determinism to gain a better replay performance. This decision depends on the context of the application and either behavior can be desirable.

### 4.3.5 Performance

As examined in Section 4.2.3, the validation of a retroactive modification can require checking the foregoing and subsequent timeline from the point of the modification for consistency. Removing or recomputing direct and indirect causally dependent events

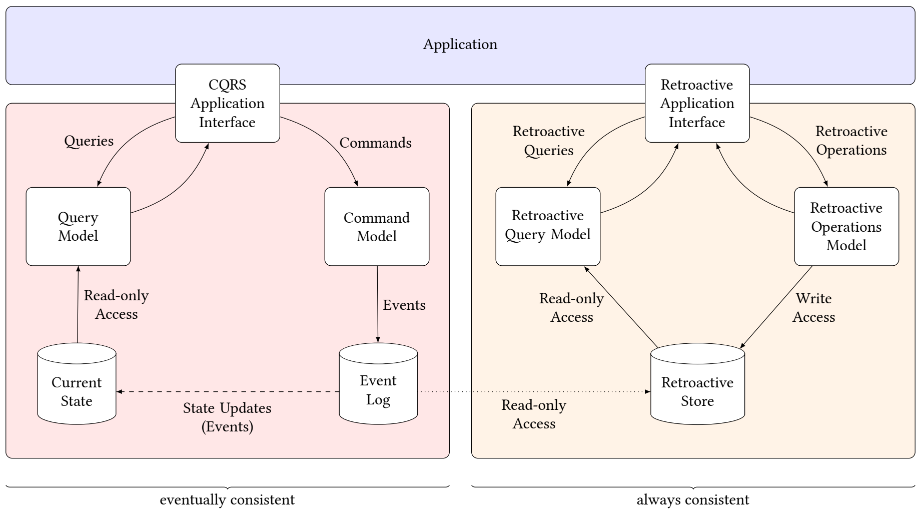


Figure 4.5: The retroactive computing features are introduced as a plug-in. The dotted connection marks the only connection from the component to the original model: the retroactive store needs access to the event log.

might be further performance issue. Thus, separating retroactive computations on branches from the computations of the live system, on the main timeline might even become necessary. When retroactive computations are decoupled in a separate, asynchronous environment, the main timeline might advance whilst retroactive computations are conducted. When computations are expensive and take hours, days, or even weeks, both systems might diverge and the computed results might no longer be relevant to the live system. Hence, performance issues can have a limiting effect on the value of retroaction and it can be important for retroactive computations to be efficiently computable.

## 4.4 Retroaction-enabled Architectures

In this section, we examine two contrasting possibilities how retroaction can be integrated in an event-sourced architecture following CQRS: (1) as a separate component with no modifications to the original architecture (i.e. as a plug-in) or (2) designed right into the architecture (i.e. unified). This is not to say that these are the only two possibilities. In fact, there are a lot of possibilities how this can be achieved, but we focus on two outstanding possibilities.

The idea of CQRS is that commands and queries operate on separate models. As detailed in Section 4.2.6, this is well suited when building systems in which an eventually consistent behavior is inevitable, but not appropriate for our use cases of retroaction. Following our argumentation from Section 4.2.6, we thus do *not* apply CQRS for the retroactive computing features in the following architectures. Instead, we propose for retroactive operations and retroactive queries to both operate on the same data model. This way, the data model can enforce causality.

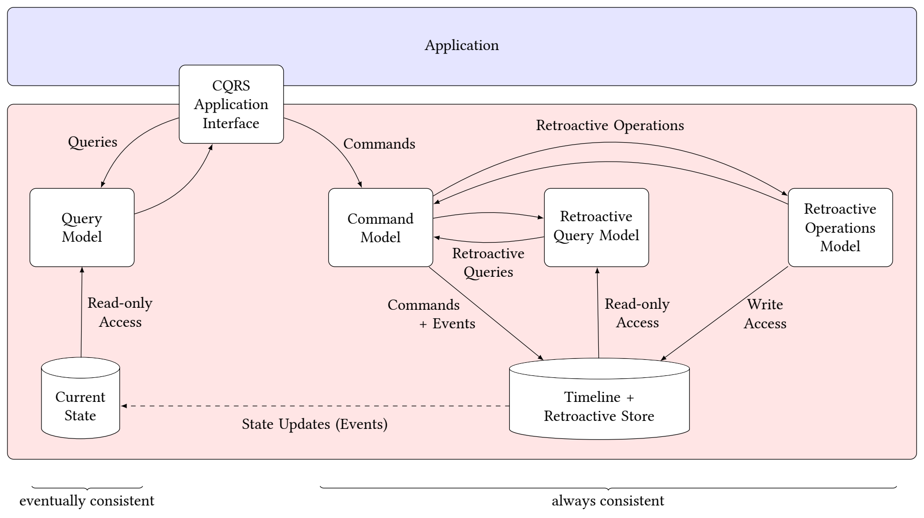


Figure 4.6: This architecture design integrates the retroactive computing capabilities into the ES+CQRS architecture in a more unified approach.

### 4.4.1 Plug-in Architecture

The first option which we describe introduces retroactive computing as an additional, separate component into an existing ES+CQRS system. The system is strictly event-sourced: commands are not persisted, solely events are sourced. Retroactive computing in this architecture takes place on a high level, in the application. This is how CQRS is usually applied: The business logic resides within the high level application layer and commands merely hide implementation details of how commands are executed. This architecture is depicted in Figure 4.5. The retroactive features can be accessed within the application – which already has access to the CQRS API over a Retroactive API. The application using the component can execute retroactive queries and operations over this API. The results from retroactive queries and operations are returned back to the application, which can then decide how to use them. A "Retroactive Store" component holds all branches and their modifications in relation to the timeline. The "normal" event log possesses an append-only semantics and may not be retroactively modified. Branches in the retroactive store, on the other hand, can be edited directly. Causality of retroactive operations and retroactive queries can be enforced by accessing the same model (marked as "always consistent" in the figure). This follows our ideas described for what-if questions in Section 4.2.6.

We have created a prototypical implementation of this architecture with JavaScript as the scripting language for the underlying runtime engine and JSON as the data format for events and commands. The scenario implemented in our prototype is an online shop with retroactive capabilities. The online shop exposes functionalities via a web service API (displayed as the "CQRS API" in the figure). Through this interface it is possible to add products to a cart or place orders. Over the "Retroactive API" it is possible to invoke retroactive commands, such as calculating hypothetical discounts, if orders would have been executed as a privilege customer.

### 4.4.2 Unified Architecture

A contrasting possibility to the described plug-in architecture, is to integrate retroactive computing as an integral part of the ES+CQRS architecture in a way that retroactive operations or retroactive queries can be invoked within commands. Retroaction thus can be viewed as an integral part of the programming model. As in the plug-in architecture, causality of retroactive operations and retroactive queries is enforced in this architecture as well. This is achieved by accessing the same model ("always consistent" in the figure). The only way for the application to access retroactive capabilities is by issuing commands. It is not possible to access retroactive features directly from the query model. In order for the application to query results of retroactive operations, an event needs to be appended to the log. This is not ideal for all use cases, since the event is replicated to all query models. Especially in large systems with many replicated query models, this might be an unnecessary overhead. But this concept of persisting retroaction via events, resonates with the idea that state mutations are persisted and thus the state of a system can always be traced or rebuilt. There exists only one store in this architecture, it contains the timeline and its branches. A conceptual and prototypical implementation of this architecture is described in Chapter 5.

### 4.4.3 Comparison

Each architecture addresses a different layer of interaction with the retroactive capabilities. In the plug-in architecture, retroaction is added as an additional feature to a traditional CQRS architecture. Application commands can even be issued based upon retroactive insights. Though this should not have the consequence that the application creates its own commands by querying the retroactive store and issue application commands based upon the results. The idea behind CQRS for the application is to issue commands, but not to define commands. If this happens, this results in an architecture where commands (i.e. application logic) and state resides in multiple components. Thus commands and state are not always persisted in the timeline. This breaks the traceability of a system and restoring arbitrary states is no longer possible.

In the unified architecture, on the other hand, application commands access retroactive features indirectly. Retroaction is integrated into the programming model, which thus gains the ability to access and manipulate past states of objects and explore alternate states. This enables a retroactively expressive programming model and makes certain application commands and queries feasible, which are not possible otherwise (e.g. calculating a discount based on previous orders, in a `PlaceOrder` command). A further advantage of this unified approach is that the access to retroactive features happens in the same environment as the programming takes place. This enables developers to use data structures, functions, and libraries from the program in the retroactive code.

Introducing retroactive computing as a plug-in requires no modifications to an existing CQRS architecture and can thus be advantageous when enhancing an existing architecture. There is only one connection from the plug-in to the original model: the plug-in needs access to the event log. The integration of retroactive computing as a core part of an ES+CQRS architecture, on the other hand, does not suit well with existing event-sourced systems, since it requires profound architectural modifications. Nevertheless, it yields more expressive commands and has possibilities which cannot be reached with a plug-in.

In the separate architecture, modifications to the event log are recorded in the retroactive store, but not in the event log. Restoring state based solely on the event log thus is no longer sufficient, the retroactive store needs to be rebuild as well. In the unified architecture, on the other hand, the commands using retroaction are sourced, as well as their resulting events. Retroaction is part of a command and results in events. Thus, the timeline always contains all retroactive operations ever applied – it is still the only source of application history.

Retroactive operations can be quite costly, since they might require a lot of re-computations. In the plug-in, these costs arise only when retroactive operations are applied; otherwise they are not present and do not influence the performance of the system. The unified architecture, however, imposes more costs to the overall system performance, since commands need to be sourced and the effects of retroactive operations are replicated. Furthermore, if a command in the unified architectures applies complex retroactive operations, the further execution of sequential commands is blocked. Thus, if a command starts with expensive retroactive calculations, the execution of subsequent commands halts. The eventually consistent semantics of commands and queries mitigates this issue, but it is still not ideal.

Many of the ideas introduced in this chapter only play out their full strength if an ES+CQRS system is built with them in mind from the ground up. For example, splitting the side effects from larger commands into individual commands for better control in replays is hard to do in existing systems.

## 4.5 Summary

In this chapter, we described a number of conceptual issues when utilizing retroaction in event-sourced systems. The two most pressing issues concern the dealing with side effects and issues of keeping a consistent timeline. We illustrated that restraining direct editing operations to branches of a timeline can prevent causality issues. Furthermore, it is possible to retain traceability and append-only behavior in event-sourced systems if results from retroaction are integrated back into the system as events.

In order to impose control over side effect afflicted commands in replay scenarios, we discussed several possibilities and concluded the discussion by introducing the idea of partial replays which require tracking causalities among events. If side effects are outsourced into separate, individual commands they can be partially reused or reinvoked. The command and event primitives of event sourcing are an ideal match for imposing control over side effects. Through events, side effects can be recorded and re-used. Through the sourcing of commands, side effects can be reinvoked.

Next, we discussed a number of ideas for keeping a consistent timeline after a retroactive modification. For this, we transferred ideas from time travel theory to retroaction in event-sourced systems. As correspondences to parallel universes we proposed to prohibit editing the own timeline and allow direct retroactive modifications only for branches. Analogue to the self-consistency principle, we proposed validation conditions and the replay (or removal) of causally related events. We examined, that validation conditions can only be imposed once the result of a command is clear, for which the command needs to be invoked. This problem is especially challenging once the command yielded side effects, which cannot be revoked. Furthermore, this chapter contributed an overview on constraints and limitations of retroaction in event-sourced systems. We highlighted the central role of hidden causalities through real-world coupling or through side effects. Further limitations, which we identified, were the influence of causality violations, causally equivalent replays, and the semantics of commands which may annihilate retroactive modifications. The performance of retroactive computations can be a further limiting factor.

Our conceptual considerations were concluded with the suggestion of architectural modifications (e.g. resolving the command/query segregation for retroaction) and the demonstration how different architectures can be used for enabling retroactive capabilities of event-sourced applications.

# Chapter 5

# The Programming Model

The previous chapter has examined the application of retroaction in event-sourced systems. We described conceptual considerations and challenges which emerge from this combination. Furthermore, we examined two very different architectures which utilize retroaction. The objective of this chapter is to illustrate how a programming model for the unified architecture can work. For this, we outline an appropriate programming model and its prototypical implementation.

## 5.1 System Model

We describe the *outline* of a programming model and make many simplifying assumptions in order to focus on relevant concepts. The objective of our programming model is to illustrate how retroaction in event-sourced systems can work. We do not claim to lay out a programming model which withstands challenges such as performance, scalability, parallelization, or distribution. We make the assumption that the programming model runs in a single-threaded environment with one timeline, that sufficient resources are available, and that no error handling is required. In the remainder of this chapter, the term *runtime engine* is used when referring to the system which provides underlying functions (such as the persistence of events). We will not deepen details of such a runtime engine, since this is not subject of this work. The code examples in this chapter are written in the scripting language JavaScript. They are excerpts from our prototypical implementation, which is detailed in the second part of this chapter. The code examples have in most cases been shortened to focus on the concepts, but they are otherwise in correspondence to our implementation.

## 5.2 Primitives of Retroaction

The objective of the programming model outlined in this chapter, is to provide programmers with access to an application's history in the programming environment. We aim for the possibility of modifying and interacting with the application's history retroactively in a single environment. The primitives of retroaction which we require for this are:

1. *Branching:* It needs to be possible to create a branch at a certain branching point in the timeline. The subsequent timeline from this point on is copied to the branch and can be modified. Modifications on the branch do not affect the timeline. Through the specification of a branching point, the retroactive operations can be restrained to the part of the timeline from the branching point on.

2. *Access to Prior States:* The application can access its own history and the history of branches.

3. *Interaction with Branches:* The state of branches can be queried and results from computations on branches integrated back into the timeline.

4. *Delete/Insert Commands or Events:* Specified commands or events can be retroactively deleted or inserted.

5. *Exchange Command Processing:* The command processing implementation can be retroactively exchanged and evaluated by replaying the application's command history.

For our further considerations we build upon the principles of the unified architecture described in Section 4.4.2 and illustrated in Figure 4.6 (both on page 37). In short, we retain the overall CQRS architecture with a segregated command and query model. Commands are issued from an application layer and result in events, which are published to the query model. In commands, developers have access to retroactive operations. They can create branches, as well as insert (or delete) commands and events. The only mean to read state is by issuing queries. A close coupling of queries and subsequent, dependent command to the actual CQRS interface is discouraged for the higher application logic.

### 5.2.1 Events

#### Events as State Modifications

In a strictly event-sourced architecture, events are viewed as *implicit* changes to an abstract system state (e.g. `AddedProductToCart(id=533)`). In order to derive the current state of the application (e.g. the list of items in the shopping cart), state has to be constructed by interpreting events. For our programming model we propose a different point of view: system state as an object, accessed and modified by commands. Events are created by an underlying runtime engine after modifications have been applied; they contain the modifications which were applied to the state object. The event is created by the runtime engine, which computes the differences to the prior state object. This point of view eases us to record which persisted events and commands in the timeline possess a causal relationship, by annotating commands with the information which properties of the state objects were used to compute the state modification. Among other things, tracking these dependencies can be used to determine what needs to be replayed after a retroactive modification has been applied (Section 4.2.4). This can be achieved by tracing which state properties in the future are affected by a retroactive modification. A retroactively inserted command `AddProductToCart` could e.g. require the recomputation of a subsequent `PlaceOrder` command, in order to keep the timeline consistent. Similarly, when removing an event, all causally dependent events can be removed as well. This can prevent issues of causality violations (e.g. temporal paradoxes).

The following algorithm can be used to compute the causally related events to a selected event. The underlying concept is that a causal relationship exists when the computation of an event has built on a prior event. The algorithm is initialized with a selected event and returns all forward events in the timeline, which have a causal relationship with it.

1. Lookup which state properties were modified during the computation of `selectedEvent`. Save those as `modifiedProperties`. Create an empty array `causalityTrace`.

2. Examine the subsequent event in the timeline (`nextEvent`), until the algorithm returns or the end of the timeline is reached:

    - Was `nextEvent` computed by reading any of these `modifiedProperties`?
        - Yes? The event has a direct causal relationship to `nextEvent`, add the event to `causalityTrace`. To find the indirect causally dependent events, apply this algorithm recursively for `nextEvent`.
        - No? Check if the computation of `nextEvent` modified any of the `modifiedProperties`.
            - Yes? The computation replaced certain properties without reading anything originally used by `selectedEvent`. These properties can then be removed from `modifiedProperties`, since subsequent events will not have a causal relationship to `selectedEvent` based on these properties. The next step is to check, if there are any properties left in `modifiedProperties`.
                - Yes? Process to the next event in the timeline.
                - No? The algorithm returns its results.
            - No? Process to the next event in the timeline.

### Capture State Changes in Events

Events capture state changes to the system state object. In the context of retroaction, the form in which state changes are recorded (i.e. the delta encoding) is important to consider. The delta encoding heavily impacts the possibilities of retroaction. Let us examine two options how data differences can be encoded: Unix diffs and JSON Patches.

In Unix, the `diff` tool can be used to output differences between two files. This output can be used to transform matching parts of a file using e.g. the `patch` tool. Both tools are commonly used for software development and utilize basically this format:

```
-state.shoppingCart = [1, 2, 3];
+state.shoppingCart = [1, 2, 3, 4];
```

This so called “diff” describes the difference from one file to another on a line-by-line basis. Diffs are commonly applied to replace parts of one file (the ones prefixed with “-”) with other parts (prefixed with “+”). Thus, the diff can only be applied to exactly the first line, prefixed with “-”. If we modify line 1 of the original file to “`state.shoppingCart = [0, 1, 2, 3]`”, the above patch then could no longer be applied to the file, since line 1 of the above diff would no longer match the file. This is a desirable behavior when working with source code, where minor differences can have a big impact.

For retroaction, however, a different behavior can be more appropriate. If we introduce a retroactive change, it is desirable to retain the possibility of applying all subsequent changes (i.e. diffs or patches) in the timeline. For example, adding an id to the above `shoppingCart` array in the original file should not annihilate previously created patches which are placed later in the timeline. If these later patches add other products to the cart it would be desirable to retain the possibility of still applying them. This can be achieved by capturing the *operations* which were applied to the state object. A technology which uses such a mechanism are JSON Patches [46]. They encode modifications to JavaScript objects and provide the ability to rebuild the state of an object by applying these modifications successively. The JSON Patches specification defines six operations which can be applied to properties of objects. Among them are operations which encode that properties were added, removed, replaced, or moved. Transferred to the above example, an according JSON Patch can be of this form:

```
{ "op": "add", "path": "/state/shoppingCart/-", "value": 4 }
```

The "`-`" suffix in this case specifies that the value 4 is appended to the array `state.shoppingCart`. In this case, changes to the original file do not conflict with the patch – it can still be applied. But the issue is not solved completely; in the case of direct value assignments to a variable, or the direct modification of a value at a certain index in the array, the patch still annihilates retroactive modifications. This is not necessarily an issue, but it can have a limiting impact on retroactive modifications. In Section 5.2.4, we describe how retroaction-aware programming can be utilized to mitigate this issue. For the programming model in this chapter we focus on *events as recordings of the operations behind state changes*, as used by JSON Patches.

### 5.2.2 Commands

We apply the view on commands as described for CQRS (Chapter 2): commands as requests to invoke a certain logic. This decoupling enables us to separate intended modifications from the command processing implementation which they trigger. Separating these concerns enables us to retroactively replace the command processing implementation at an arbitrary point in the timeline and reprocess commands with a different processing implementation. This allows for the observation of alternate application states.

In our programming model, the runtime engine splits "external" requests into a series of multiple "internal" commands. The commands in this series then are sequentially processed. After the processing of a command is finished, the runtime engine checks if it succeeded. Therefore the processing function returns information on its success or failure. If the processing succeeded, an event is computed and persisted to the timeline. Next, the subsequent command in the series is processed. If the command failed, the commands in the series will not be processed further. But the subsequent commands – which were not invoked – are still persisted to the timeline. This serves the purpose of enabling later command replays.

The resulting event from a command processing is created by the underlying runtime engine. This event contains modifications of the state object which occurred during the command processing. Together with the command which triggered the processing, it is appended to the timeline. This concept follows our conceptual considerations from Section 4.2.5: We argued that it is important for expressive retroaction to view side effects as isolated, for they can be controlled individually in a replay. Furthermore, we argued that each of the commands must be persisted in the timeline, independent of their failure or success.

In the programming model in this chapter, a developer writes individual processing functions for each command. Listing 5.1 displays the processing of a simplified command. The depicted command applies a discount to an order, and annotates what it used for its computation. Furthermore, the command and the resulting event are tagged. These tags can be used to reference commands or events (e.g. for removals or as branching points). The concept of tagging commands and events will be detailed in Section 5.2.3 of this chapter.

Listing 5.1: Example of a command.

```
commands.tags.applyDiscount = ["discount-modifications"];

commands.applyDiscount = function(request, state) {
  /* find the total order price in the global state object */
  var totalPrice = state.orders[request.orderId].totalPrice;

  /* apply 10% discount */
  state.orders[request.orderId].discount = totalPrice * 0.1;

  /* if the order is changed retroactively at a past point this
     command needs to be replayed, since a different discount
     would then be computed */
  var readDuringComputation = [
    "state.orders[" + request.orderId + "].totalPrice"
  ];

  /* specify tags for this event */
  var eventTags = ["applied-discount"];

  return {
    success: true,
    newState: state,
    read: readDuringComputation,
    eventTags: eventTags
  };
}
```

### 5.2.3 Retroaction

#### Scope of Branches

There are a number of possibilities concerning the duration of a branch's existence. Two possibilities are (1) for branches to be persisted as part of the system's state. This enables the usage of a branch in multiple commands. An option (2) is for branches to be only existent temporarily, during the processing of a single command.

If branches are available over the course of processing multiple commands, it is important to consider that this yields the possibility of nested branches. Nested

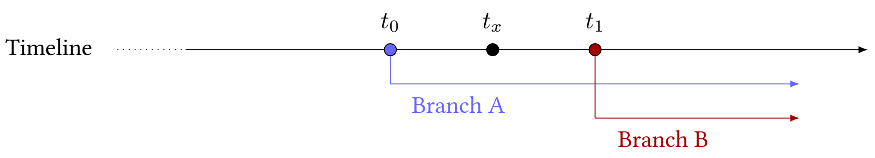


Figure 5.1: The figure depicts a scenario, in which branches exist as part of the system state. The system at $t_0$ creates a branch A at an arbitrary former point. Branch A exists from then on within the system state (i.e. the timeline). At $t_1$, a further branch B, with its branching point at $t_x$, is created. Branch B then contains the *nested* Branch A.

branches can then occur when a branch is created within a command and succeeding commands create further branches (Figure 5.1). The further branches would then contain the system state – and thus the earlier created branches as well. To avoid problems here, the same branch object should not be referenced in multiple branches. Each event and command should be assigned to exactly one branch This allows for the coexistence of a branch in various branches. If the same branch object is nested in multiple branches, unforeseeable behavior can occur. For our programming model we view branches as only temporarily existent, for the scope of a command. Persisting branches as part of the system state would imply that they are replicated to all data models. For the use cases in this chapter this is unnecessary.

### Timeline Pointers and Tags

In order to enable expressive retroactive operations, it needs to be possible to reference points in the timeline. These *timeline pointers* can then be used to reference events and commands as branching points, insertion marks, or for removal. To achieve this, we propose the possibility of annotating events and commands with a list of tags. These tagged events and commands can then be used as pointers to locations in the timeline.

Enabling the tagging of commands *and* events makes sense, since they both serve a different purpose. A request to invoke a command is always issued by the application, whereas an event describes how the system reacts to it. The resulting event, triggered through the command, depends on the command processing implementation of the system. Therefore it makes sense to annotate the resulting event with tags *in the command processing*. Annotating commands enables us to filter for commands, independent of how they are processed. For commands, tagging allows to dynamically exclude message categories (or individual messages) in replays. This can be used to exclude e.g. side effect afflicted operations, in order to prevent side effects from being reinvoked.

### Access to Retroaction

The retroactive capabilities in our programming model can only be accessed from within commands – not within queries. This characteristic adheres to the design of the unified architecture as described in 4.4.2. In this section we examine individual operations of our API in detail. In order to utilize retroaction, the first step in our programming model is always to create a branch of the "main" timeline. For this, a branching point needs to be specified. The following retroactive operations can be applied to create branch objects and gain retrospective information:

(i) `branchObject = createBranch(:tag, ”before” || ”after”)`
A branch of the timeline can be created by invoking this function within a command. The `:tag` is a unique tag of either a command or an event. By supplying “`before`” or “`after`”, it is possible to specify if the branching point should precede the tag or be placed after it.

(ii) `stateObject = branchObject.getState()`
This method can be applied to a branch object, it returns the latest state object.

(iii) `eventArray = branchObject.getEvents([:tag])`
Returns an array of events, which match the supplied tags. For each event the state of the timeline at the time of the event is supplied as well. This state can be accessed under the property `:event.state`. When called on a branch object, the events are returned from this branch.

Retroactive modifications of an application’s history can be removal or insert operations, as well as a retroactive change of the underlying application logic (the command processing). Following our argumentation from Section 4.2.2, retroactive modifications can only be applied to branch objects. Furthermore, a branch cannot apply retroactive modifications to itself. The following operations can be used to apply modifications to the branch object:

(iv) `branchObject.insertEvent(:tag, :event, ”before” || ”after”, :validatorFn)`
An event in the form of a JSON Patch can be inserted at a unique, tagged point. The validator function `:validatorFn` is optional and will described in more detail in this chapter.

(v) `branchObject.deleteEvents([:tag], :dependentOnes)`
If `:dependentOnes` is set to true, the events which have a causal relationship to the ones supplied are removed as well.

(vi) `branchObject.insertCommand(:tag, :command, ”before” || ”after”)`
A command can be inserted by specifying an insertion point and a command. The listings in this chapter contain examples for this command.

(vii) `branchObject.deleteCommands([:tag], :dependentOnes)`
If commands are deleted, the respective event is deleted as well. Otherwise it would no longer be possible to conduct a command replay for a corresponding event.

(viii) `branchObject.changeCommandFunction(:old, :new)`
`changeCommandFunction(:old, :new)`
This function can be invoked either for branch objects or for the timeline. It inserts a special type of “runtime engine command-event pair” in the timeline. This pair takes a special role in command and event replays. It denotes that for the processing of the `:old` command, the `:new` function should be used from this point on. We utilize call-by-naming for this and supply the function names as arguments. It is necessary to insert a command-event pair to denote this

change in the processing function. Inserting solely an event, would pose an issue to command replays, since this event would not be considered in a command replay.

Another possibility for persisting changes in the command processing, is to understand the command processing implementation of a system as part of the state. This concept is used by the Chronograph platform and will be detailed in Chapter 7.

Our programming model exposes additional operations, to manually trigger replays. An alternative would be an immediate evaluation strategy, where the computation of changes takes place immediately after e.g. an insert operation has been invoked. But this does not suit well, for the reasons described in Section 4.2.2: using immediate evaluation in conjunction with a series of multiple retroactive operations then leads to potentially unnecessary computations.

(ix) `branchObject.rebuildState()`
State on the branch is rebuilt based upon the persisted events.

(x) `branchObject.commandReplay([:tag])`
This method triggers a replay of the commands supplied in the tags array. The recomputed commands yield newly computed events. For these "new" events, causal dependencies to subsequent events are checked. If subsequent events with a direct or indirect causal relationship exist, these are recursively recomputed as well (by re-processing their respective commands).

(xi) `branchObject.partialReplay([:reuse], [:recompute])`
The `:reuse` and `:recompute` arrays contain tags. If causally dependent events exist, they are recomputed as well.

The following listing (Listing 5.2) depicts an example of a command which creates a branch, applies retroactive operations to it, and integrates results from the branch back into the main timeline. The branch is encapsulated in an object, which exposes access to retroactive modification methods, replay mechanisms, and the branches' history, and current state. It is then possible to create history-aware algorithms, which utilize the application's history as part of their computation. We describe examples of such commands in the later part of this chapter.

Listing 5.2: Example command, which utilizes retroaction.

```
var newCommandFn = function(command, state) {
  // ...
  return { newState: ..., read: ... };
}

var exampleCommand = function(command, state) {
  /* the branch is an object which exposes retroactive operations */
  var branchingPoint = "timeline-tag0";
  var b = createBranch(branchingPoint);

  /* an event is created (in the form of a JSON Patch) and inserted */
  var evt = createEvent({"op": "add", "path": "/state/-",
            "value": {"name": "Alice"}});
  b.insertEvent("timeline-tag1", evt, "before");
```

```
    b.rebuildState();

    /* get the state from the branch */
    var s = b.getState();

    /* retroaction */
    b.changeCommandFunction(”exampleCommand”, ”newCommandFn”);
    b.insertCommand(”footag”, ”AddToCart({ product: 321 })”, ”before”);
    b.commandReplay([”AddToCart”], true);

    /* analyze state of the branches and save result */
    var s2 = b.getState();
    if (...) state.retroactiveAnalysis = ...;

    return { newState: state, read: [] };
  }
```

After the function has returned, the underlying runtime engine creates an event consisting of the command, the command's arguments, and the state modification. This event is published to the query models and then visible to queries.

The above listing described only command and event replays. As illustrated in Section 4.2.5, the concept of partial replays gives us fine-grained control over when to newly compute an event and when to use an already persisted one. This feature can be used to control side effects. In a partial replay, causal relationships between events are taken into account. Thus, if only one command is replayed this has the recursive effect of causally dependent commands being replayed as well (Section 4.2.5). For the events which are specified to be reused, the reprocessing of the command is skipped. Listing 5.3 illustrates how a partial replay can be conducted.

Listing 5.3: The listing depicts how a partial replay can be conducted.

```
/* newly compute the commands in this array. instead of reusing
   the already persisted events, the command is newly invoked.
   events which possess a causal relationship to the newly
   resulting events are recomputed as well. */
recomputeCommands = [ ”stock-price” ];

/* for the tags in the array use the persisted events instead of
   newly processing the command */
reuseEvents= [ ”stock-price” ];

/* conduct the partial replay on a branch */
var b = createBranch(”timeline-tag0”);
b.partialReplay({ recompute: recomputeCommands, reuse: reuseEvents });
```

As described in Section 4.2.3, the recordings of causal dependencies between events cannot only be used to recompute subsequent, causally dependent events, they can also be used to remove subsequent, causally dependent events. The following listing depicts an example of such a modification:

```
var b = createBranch(”timeline-tag0”);
var deleteCausallyDependentOnes = true;
b.deleteEvent(”some-event-reference, deleteCausallyDependentOnes);
```

### 5.2.4 Retroaction-aware Programming

Programming constructs in commands can conflict with retroactive modifications. For example, a developer might always decrement a certain state property by one in a command processing implementation, whilst relying on some knowledge about the context of the application. An assumption could be that a value $x$ is defined as $x \in \{1, \dots, n\}$. The developer could apply the decrement, in order to shift the value to reference array indices so that $x \in \{0, \dots, n\}$. Through retroactive changes, this “one off” assumption may no longer hold and the decrement operation thus suddenly have a different semantics.

A first idea to cope with this issue, is to provide more semantics to certain operations. This can be done by providing helper functions (or an API) to the developer, in order to resolve the ambiguity of operations. For example, a helper function `ConvertToIndices` could be provided to developers. This function would convert numbers to array indices and could be used instead of the decrement operation. Thus, the intention behind the operation would be clear. In a replay, the helper function could be adapted, in order to not break programming instructions. Another example is the usage of `getLast()` over “`var last = state.foo[99]`”, where 99 is a presumed last index.

The delta encoding of events can also be used to persist the semantics of state changes. If the appending of a value to the just mentioned array is done whilst relying on the knowledge that the last element will “always” be 99, this could be a direct assignment of the form “`state.foo[100] = 'bar'`”. The delta encoding would then not capture that the intention was to append a value. An array push instruction – e.g. “`state.foo.push('bar')`” – is a better alternative, since this allows the runtime engine to capture that the intention was to append a value to the array. There are various possibilities how the intention of a state change can be captured: for the developer to capture this intention manually by creating the patch by himself, through source code analysis, or through helper functions which provide more semantics. The decision depends on the context of the application and either behavior can be desirable.

### 5.2.5 Validation

Validation conditions can be written as a validator function. Such a validator function can be supplied along with the retroactive operations. The validator function in this case gets the state before the modification and the state after the intended modification supplied as arguments. The NoSQL database CouchDB uses a similar mechanism for validating changes to documents[1]. If the validation is successful (i.e. it returns `true`), the event is persisted to the timeline. If not successful (returns `false`), the event is discarded. The following listing illustrates an example of a retroactive modification, where a validator function is supplied.

```
var b = createBranch(branchingPoint);

var validatorFn = function(oldState, newState) {
  if (newState.price < 0) return false;
  else return true;
};
b.insertEvent(”timeline-tag1”, ”{ items: 0  }”, ”before”, validatorFn);
```

[1]http://guide.couchdb.org/draft/validation.html

As detailed in Section 4.2.3, state mutating side effects require special attention when validating the timeline for consistency. They need to be delayed, until it is clear if the validation of the command's processing results succeeded. Otherwise, the side effect would already have occurred and could not be revoked. This can be solved by returning an array with functions which need to be delayed from the command processor. If the `validatorFn` succeeds, the functions in the array are invoked subsequently.

```
var cmdWebRequest = function(cmd, state, validatorFn) {
  // ...
  var writingSideEffectFn = function () { http.post(...); };

  return {
    state: state,
    read: [],
    delayUntilValidated: [ writingSideEffectFn ]
  };
}
```

A possible scenario to apply this concept, is to retroactively reprocess a command which executes a reading side effect. Based on the return value, the command modifies the system state and executes a writing side effect. Using the described concept, it is possible to validate the modified state, before persisting the event and triggering the side effect. The expressiveness of this concept is limited to simple scenarios though, but it provides a – admittedly narrow – possibility to use validation in combination with side effect afflicted commands. This basic concept could be enhanced by supplying e.g. the entire branch with all its events and commands. Other possibilities are to supply the command, which triggered the processing, as an argument.

### 5.2.6 Return-values of Commands

As illustrated in Section 2.4, commands in a strict ES+CQRS architecture possess no return value. But the opinion on the question whether commands may still return a value, is disputed in the CQRS domain. Some projects which apply CQRS allow for replies[1], whereas authors like Greg Young promote an explicit *no*: a command should never return a value [9, p.17]. Commands are described as possessing a void return type. They mutate state, but do not return a value. Instead, solely queries can be used to gather data. If commands never return anything, there is no way of knowing when a command has been executed or when a corresponding event has been appended to the timeline. But for certain use cases it is desirable to have this information returned to the application. This information can be used to e.g. delay queries, until a certain command has been processed.

As we view it, the overall intention behind th no-return value nature of commands, is to prevent bilateral commands which return *and* mutate state. But this does not necessarily imply that commands cannot return any value at all. We propose for commands to asynchronously return a reference to an event, once it has been appended to the event log. From our point of view this does not break with the intention of commands not returning state, since the return value of commands in this case is a *meta value* – not comparable to returning application state. Integrating the reporting of this reference could be done by supplying a callback method with the command,

[1]http://rbmhtechnology.github.io/eventuate/architecture.html#event-sourced-actors

setting a listener, or by polling the visibility status. Our prototypical implementation utilizes a further possibility, which will be detailed in Section 5.3.5.

## 5.3 Implementation

In this section we describe a prototypical implementation of the previously described programming model. As a scenario for this, we implement a simple online shop. This scenario enables us to illustrate the concepts described in the foregoing chapters of this thesis, as it poses a number of interesting challenges – such as side effects or hidden causalities – to retroaction. We deliberately do not take functionalities necessary for a productive "real world" online shop into account: authentication, authorization, or validation, for example. Considering these features would distract from the concepts which we want to illustrate and make the implementation unnecessarily complex. Furthermore, we make the assumption that the online shop has only one customer. This is certainly not realistic, but supporting multiple customers would merely make the code less clear and comprehensible. Since we aim for the code and API to illustrate conceptual matters, we thus deliberately leave out these features and focus on a simple scenario.

The prototype was implemented in the scripting language JavaScript, an interpreted language with functional aspects. Functions in this language are considered higher class citizens and can be used like other variables. These functional aspects work in favor of our concepts, since they enables us to e.g. dynamically switch the command processing implementation, supply validator functions with retroactive operations, or delay the invocation of side effects. Moreover, JavaScript forms a symbiotic relationship with the lightweight data format JSON [47], which we use in conjunction with the already mentioned JSON Patches for the events. JSON is also used as the underlying data format when issuing commands and queries, or receiving responses. The prototype was built with node.js[2] as the underlying JavaScript platform. Our prototype runs in a single-threaded loop, in which incoming commands are processed.

In the remainder of this chapter we describe further internals of our prototype. For this, we focus on the interesting parts and leave some of the underlying implementation details (such as persisting events) out. The code examples are excerpts from our prototypical implementation. In some cases we have shortened them to the relevant parts, but they concur with our implementation.

### 5.3.1 The Application Interface

Representational State Transfer (REST) is a resource-oriented style of architecture [48], which corresponds to the principles of the World Wide Web. This architectural style was introduced by Roy Fielding and has gained popularity in recent years. Today, many large web services – Twitter, Facebook, or Google, for example – utilize REST to provide APIs over the web. There are numerous reasons for this. Among them is that REST promotes statelessness – each request to an API holds everything needed to fulfill it. The server does not hold session state, which makes it easier to build scalable systems. The most common application of REST is the usage of HTTP to provide

[2]https://nodejs.org/

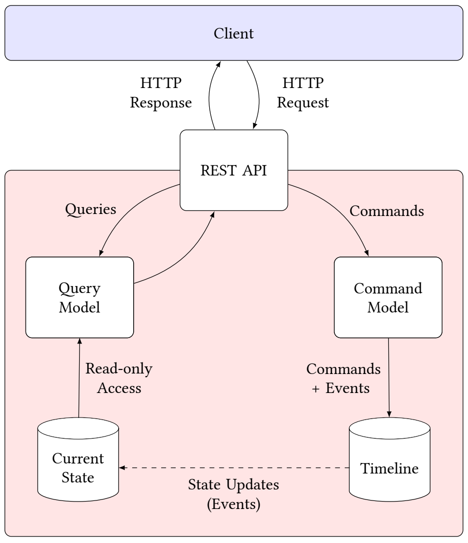


Figure 5.2: The figure is a schematic illustration of our prototype. A client sends HTTP requests to the REST API and receives a response. The API maps these requests to application commands and queries.

| | Semantics | Method | Resource | Payload | Application Mapping |
|---|---|---|---|---|---|
| Q | List all products | `GET` | `/products` | | `getProducts()` |
| Q | List details of product | `GET` | `/products/:id` | | `getProduct(:id)` |
| Q | Display items in cart | `GET` | `/cart` | | `getCart()` |
| C | Empty shopping cart | `DELETE` | `/cart/` | | `emptyCart()` |
| C | Put item in cart | `POST` | `/cart/:id` | `{qty: 1}` | `addToCart(:id, :qty)` |
| C | Update item quantity | `PUT` | `/cart/:id` | `{qty: 5}` | `updateQty(:id, :qty)` |
| C | Remove item from cart | `DELETE` | `/cart/:id` | | `removeFromCart(:id)` |
| C | Place order | `POST` | `/order` | | 1) `fetchCurrencyRate()` 2) `createOrder()` 2) `sendConfirmation()` |

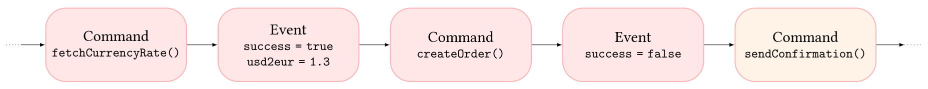
Table 5.1: The table lists resources and operations of the online shop. The REST interface maps and distributes HTTP requests to system commands (C) and queries (Q). Two application commands yield side effects: `fetchCurrencyRate()` is an external query and returns data, `sendConfirmation()` is an external command, which triggers the transmission of a mail message.

web services: Uniform Resource Identifiers (URIs) are used as identifiers when accessing resources, HTTP methods are used as verbs to operate on these resources. The HTTP verbs possess a clearly specified semantics [49]: `GET` and `HEAD`, for example, are considered safe and should not mutate state, whereas e.g. `POST`, `PUT`, or `DELETE` may change state.

Our prototype provides a RESTful API to clients. We chose to use REST, since it already provides a clear segregation in queries (safe verbs) and commands (not safe verbs). This inherent segregation works in favor of our CQRS architecture. The API can be accessed using e.g. a browser or command-line tools, such as cURL or wget. Table 5.1 lists all resources which are exposed over REST and the HTTP verbs which can be used on them. A shopping cart in a REST architecture is typically modelled as one resource `/cart`, which can be updated using the `PUT` verb. But in HTTP, `PUT` possesses the semantics of replacing a resource: "*The PUT method requests that the enclosed entity be stored under the supplied Request-URI. [...] the enclosed entity SHOULD be considered as a modified version of the one residing on the origin server.*" [49]. This can conflict with retroaction. As described in Section 4.3.3, the semantics of commands can lead to unintended reversals of retroaction. Here, a `PUT` command overwrites an entire resource each time it is invoked. Thus, retroactively injected updates of the cart would be annihilated by the next cart update in the timeline. As a consequence, we modelled individual items in the cart as separate resources. By applying `POST`, items can be added to the cart. Their quantity can be updated using `PUT`. By applying `DELETE` on individual cart item resources, they can be removed from the cart.

Furthermore, we provide a command with the semantics of emptying the shopping cart. The operation `DELETE /cart` can be used to empty the cart. This command will annihilate previous `addToCart()` commands (and events), but in this case the semantics is different: delete all items from the cart. Thus, annihilating prior events is an intended consequence.

In our prototype, HTTP requests are sent to the REST API. This interface contains a thin logic layer, it maps and distributes requests to system commands and queries. Figure 5.2 depicts a schematic illustration of this process. The mapping of application commands and requests is visible in Figure 5.1. In order to fulfill e.g. the `POST /order` command, a `fetchCurrencyRate()`, a `CreateOrder()`, and a `SendConfirmation()` command are issued subsequently by the API. This is done sequentially; the runtime engine waits for each command to finish its execution. After each command returns, the runtime engine checks if the command succeeded. If it did, the command and the resulting event are persisted to the timeline and the next command in the series is processed. If the command failed, the commands in the series will not be processed further. But the commands are still persisted to the timeline, in order to enable later replays. The following figure depicts a possible course of computation for the mentioned series. First, the currency is fetched, then the order creation is attempted. This fails and hence the subsequent `sendConfirmation()` command is not processed (highlighted in yellow). It is persisted to the timeline though and can be recomputed in a replay.



Listing 5.4 displays a relevant excerpt from the runtime engine, which depicts this process. The `fetchCurrencyRate()` has a purely reading side effect (an exter-

nal query). It fetches current currency exchange rates and persists them to the system state. These exchange rates are used during the calculation of an order price. The `sendConfirmation()` command triggers a purely writing side effect (an external command) which sends irrevocable confirmation messages via mail.

Listing 5.4: Shortened and simplified excerpt from the REST API.

```
restAPI.commands = function(request) {
  /* map HTTP requests to commands */
  if (request === "POST /order) {
    var commands = [
      "fetchCurrencyRate(request, state)",
      "createOrder(request, state)",
      "sendConfirmation(request, state)"
    ];

    // 1) invoke each command in the series sequentially.
    //    this enforces a causal order by waiting for each
    //    command to terminate, before the next is invoked.
    // 2) for each comand, calculate event from state changes
          and persist command + event.
    // 4) publish events to query model(s).
    runtimeEngine.processCommands(commands, request);
  }
}
```

## 5.3.2 Command Implementation

Commands are implemented as modifying the system state object. The runtime engine generates an event after a command has been executed. This event captures the change to the prior state object. Within commands, the programmer annotates which state properties were read during the computation. Modified state properties are automatically captured by the runtime engine. This enables the runtime engine to track causalities between events (Section 4.2.4). Thus, if we e.g. remove a `ProductAddedToCart` event from the timeline, subsequent causally dependent events (e.g. `PlacedOrder`) can be automatically removed (or replayed) as well. Listing 5.5 depicts two command implementations.

Listing 5.5: Two examples of online shop commands.

```
/* command, which adds items to the cart */
commands.addToCart = function(request, state){
  /* extract parameters from the HTTP request, update state  */
  var productId = request.params.productId;
  var quantity = request.params.qty;
  state.cart[productId] = quantity;

  return {
    success: true,
    newState: state,
    read: []
  };
}
```

```
/* command, which places an order */
commands.placeOrder = function(request, state){
  var order = { items: state.cart, sum: ... }
  state.orders.push(order);
  state.cart = {};

  return {
    success: true,
    newState: state,
    read: []
  };
}
```

### 5.3.3 Events

The already mentioned JSON Patches serve as the data format for events. Events in our implementation are created automatically by the runtime engine. We implemented this feature using an open source library[1], which offers the possibility to output changes to objects in the form of a JSON Patch. For this, it utilizes the `Object.observe()` feature from ECMAScript 2016 (ES7), the most recent specification of the JavaScript language. This feature provides a mean for recording changes to an object[2]. We applied this to record modifications of the system state object. This enables us to automatically generate the patch after a command has been processed. Together with event tags, state properties which were used, and a preliminary id (explained later in this section), this patch forms the event. After the event has been created, it is persisted to the timeline together with the command. The event is furthermore published to the query model(s). Listing 5.6 displays the patches of some events, which were automatically generated by the prototype runtime engine.

Listing 5.6: Events in the form of JSON Patches, from our prototype. They depict modifications of the system state object.

```
/* add product id 172 to the ”state.cart” object with quantity 1 */
[ { ”op”: ”add”, ”path”: ”/cart/172”, ”value”: 1 } ]

/* update the quantity of the product ”state.cart[172]” */
[ { ”op”: ”replace”, ”path”: ”/cart/172”, ”value”: 3 } ]

/* add an object to ”state.orders” array and remove it from cart */
[{
  ”op”: ”add”,
  ”path”: ”/orders/-”,
  ”value”: {
    ”timestamp”: 1458061070313,
    ”content”: { ”172”: 3 }
  }
},
{
  ”op”: ”remove”,
  ”path”: ”/cart/172”
}]
```

[1]https://github.com/Starcounter-Jack/JSON-Patch
[2]https://developer.mozilla.org/de/docs/Web/JavaScript/Reference/Global_Objects/Object/observe

### 5.3.4 Queries

Queries are sent as HTTP requests to the REST API, using safe HTTP verbs. The thin logic layer in the API then maps the requests to a query model, which responds with the state visible in the model. The method in the following listing returns the shopping cart content, when queried.

```
queryModel.getCart = function(request, state) {
  request.send(state.cart);
}
```

We apply CQRS here: The query model is decoupled from the command model and there might exist a multitude of replicated query model instances which could be used to conduct e.g. load balancing. As examined in Section 2.4, this results in an eventually consistent behavior of queries. Listing 5.7 displays an excerpt of this routing, using a simple round robin algorithm.

Listing 5.7: Simplified excerpt from the query routing in the API.

```
restAPI.queries = (request, state) {
  var roundRobin = ++roundRobin % queryModels.length;
  if (request === ”GET /cart”) {
    queryModel[roundRobin].getCart(request, state));
  }
  // ...
}
```

### 5.3.5 Preliminary Command IDs

Following our argumentation from Section 5.2.6, we implement commands in a way that the HTTP request receives a preliminary command id as an immediate response. This preliminary id can be used by clients to ensure that queries are only fulfilled, if the result of a certain command is visible in the query model. We propose a solution which ties in with the principles behind REST. Commands in our prototype return the HTTP status code "`202 Accepted`" to indicate that the request has been accepted and will be processed asynchronously. The preliminary id is returned as the payload:

```
HTTP/1.1 202 Accepted
Content-Type: application/json;charset=utf-8
...

{ ”preliminary_id”: 533 }
```

Queries can be restricted to a certain preliminary id. If this id is not yet visible in the query model, the query is rejected. To achieve this, we utilize the `If-Match` field available in HTTP/1.1 as a mean for conditional HTTP requests. The specification matches our use case: "*The 'If-Match' header field makes the request method conditional on the recipient origin server either having at least one current representation of the target resource [...]*" [50]. Listing 5.8 depicts a simple script which we used to test our prototype (among other test scripts). The script sends two succeeding HTTP requests: a command (line 6-10) and a query (line 14-17). The query is sent as a conditional request using the preliminary id which was returned by the prior command.

Listing 5.8: Simple Test Script for our Prototype.

```
#!/bin/ksh
URI=http://localhost:8000

# add item to cart
PRELIMINARY_ID=$(
  curl  -X POST                                     \
        -H ”Content-Type: application/json”         \
        -d ’{ ”qty”: 3 }’                           \
        $URI/cart/724                               \
        | grep -o ’[0-9]*’
)

# list items in cart
curl    -v -X GET                                   \
        -H ’Accept: text/plain’                     \
        -H ”If-Match: $PRELIMINARY_ID”              \
        $URI/cart
```

The command adds a product to the cart and obtains a preliminary id. The HTTP exchange looks roughly like in the following listing. Lines starting with “>” indicate a HTTP request from the client to the server, lines starting with “<” mark the response.

```
> POST /cart/724 HTTP/1.1
> Content-Type: application/json;charset=utf-8
> ...
>
> { ”qty” : 3 }

< HTTP/1.1 202 Accepted
< Content-Type: application/json;charset=utf-8
< ...
<
< { ”preliminary_id”: 1458171779004 }
```

The second HTTP request (from Listing 5.8), obtains the content of the shopping cart. In accordance to the HTTP/1.1 specification for conditional requests, the server returns “`304 Not Modified`” if the condition evaluated to false and “`200 OK`” if it succeeded [50]. The following listing depicts this rejection (line 5-9) and the alternative response (line 13-19).

```
1   > GET /cart HTTP/1.1
2   > If-Match: 1458171779004
3   > ...
4
5   < HTTP/1.1 304 Not modified
6   < Content-Type: application/json;charset=utf-8
7   < ...
8   <
9   < { ”visible” : false }
10
11  OR
12
13  < HTTP/1.1 200 OK
14  < Content-Type: application/json;charset=utf-8
```

```
< ...
<
< [
<   { ”product1” : { ”title”: ”Lorem Ipsum”, ”price”: 99 }  }
< ]
```

### 5.3.6 Retroactive Use Cases

In this section we describe some of the use cases for retroaction, which can be applied to our prototype.

#### History-aware Algorithms

Through the programming model which we described, the history of the application is exposed to commands. This can be utilized to create history-aware algorithms. An example for this is a `PlaceOrder` command which calculates an order discount based on previous orders by this customer. Listing 5.9 depicts the commands' implementation. First, a branch of the timeline is created, then all events tagged with `placed-order` are searched for. Next, these events are examined in order to compute a discount for the current order.

Listing 5.9: When placing an order, a discount based on previous orders is calculated.

```
commands.placeOrder = function(request, state) {
  /* ’big bang’ is a pre-defined tag of the runtime engine which
     refers to the start of the timeline. */
  var b = retroactive.createBranch(”big bang”);

  /* find all events tagged with ’placed-order’ */
  var orderEvents = b.getEvents([”placed-order”]);

  var allOrdersAmount = 0;
  for (var o in orderEvents) {
    /* get the system state object at the point of this event
       in the timeline */
    allOrdersAmount += orderEvents[o].getState().totalAmount;
  }

  var discount = 0;
  if (allOrdersAmount > 1000) discount = 0.1;      /* 10% */
  else if (allOrdersAmount > 100) discount = 0.05; /*  5% */

  var order = {
    items: ...,
    totalAmount: ... * discount
  };
  state.orders.push(order);

  return { success: ..., newState: ..., read: ... };
}
```

### Partial Replay with Control of Side Effects

The concept of partial replays allows for control of the side effects in a replay (Section 4.2.5). It is possible to either reuse prior results or reinvoke them and continue the processing with these new results. As we described it, the shop fetches current currency exchange rates before creating an order. A use case is to replay the shop timeline and reprocess each `fetchCurrencyRate()`. In such a replay, the shop fetches each currency exchange rate from an external web service again. Each causally dependent command (`placeOrder()` in this case) will then be reprocessed with the new result. In order to prevent confirmations from being sent out again, we suppress them. In this case, suppressing the invocation means that we reuse the event from the last `sendConfirmation()` invocation (if an event is available).

Listing 5.10: Partial Replay Example

```
/* newly compute the events in this array, instead of reusing the
   already persisted ones */
var recomputeCommands = [ ”fetch-currency-rate” ];

/* for the tags in the array use the persisted events instead of
   newly invoking the command */
var reuseEvents= [ ”confirmation-mail” ];

/* normal event replay, but for the supplied ’newlyComputeEvents’
   array the belonging command is newly invoked. Events which
   possess a causal relationship to the newly computed ones are
   recomputed as well, as long as they are not specified in the
   ’reuseEvents’ array. */
var b = createBranch(”big bang”);
b.partialReplay({ recompute: recomputeCommands, reuse: reuseEvents });
```

In the code above *all* `fetchCurrencyRate()` commands and the direct and indirect causally related events are reprocessed. But it would also have been possible to reprocess just a single specific `fetchCurrencyRate()` and its causally related events. This can be achieved through the usage of unique (or appropriately fine-grained) tags within the commands. These tags can then be filtered for in a later replay.

### Alternate Command Behavior

We can adapt the above code to evaluate how different currency exchange rates would have affected the state. To achieve this, we exchange the `fetchCurrencyRate()` command in a branch. Instead a function which returns hypothetical exchange rates from a fixed list is used. Then a replay of the relevant commands is conducted. Next, the resulting state can be accessed for further analysis:

```
var experimentalRatesFn = function(request, state) {
  /* use this fixed currency rate for the new processing */
  state.currencyRates = { eur2usd: 1.13 };

  return {
    success: true,
    newState: state,
    read: []
  };
}
```

```
var someCommand = function(request, state) {
  b.createBranch("big bang");
  b.changeCommandFunction("fetchCurrencyRate",
                          "experimentalRatesFn");

  b.partialReplay({
    recompute: ["fetch-currency"],
    reuse: ["confirmation-mail"]
  });

  var experimentalOrders = b.getState().orders;

  // further processing
  // ...

  return { success: true, newState: ..., read: ... };
}
```

### Observe Alternate States

We implemented a feature which allows for displaying how the orders would have been different, if customers would not have removed any products from their cart. For this, a branch is created and all events tagged with `product-removed-from-cart` are deleted from it. Next, a replay of commands tagged with `place-order` is conducted.

```
commands.calculatePossibleOrders = function(request, state) {
  var b = retroactive.createBranch("big bang");

  /* supplying true would remove all causally dependent
     "place-order" events as well. */
  b.deleteEvents("product-removed-from-cart", false);

  /* only these commands (and the ones which read state modified
     by these ones) are recomputed, for other commands the events
     are reused */
  var recomputeCommands = [ "place-order" ];

  var b = createBranch("big bang");
  var reuseEvents = ["confirmation-mail"];
  b.partialReplay({
    recompute: recomputeCommands,
    reuse: reuseEvents
  });

  var branchState = b.getState();
  state.whatifOrders = branchState.orders;

  return {
    newState: state,
    success: true,
    read: []
  };
}
```

The result from this analysis is saved to the state property `state.whatifOrders` and subsequently persisted to the timeline as an event, by the runtime engine. The query model(s) eventually receive the events as state updates. Then the results are visible in the query model. They are exposed to the clients through a query:

```
queries.getPossibleOrders = function(request, state) {
  return state.whatifOrders;
}
```

### Retroaction for Optimization

It is possible to use the application's history for optimization or adaption of the application's current behavior. This is the use case which we described in Chapter 3 as history-aware and self-improving algorithms. Concerning the online shop, this section described the implementation of a history-aware discount algorithm. But it is also possible to use the system's history for evaluating experimental – and possibly better performing – algorithms. This can be achieved by replaying commands from the timeline with a different command processing implementation. The resulting state can then be used for comparison against the current state of the timeline – and thus against the current discount algorithm. Next, we can adapt the future application behavior based on these insights.

To achieve this, we create a command which creates a branch, exchanges the `placeOrder()` command implementation, conducts a replay, and compares the branch state against the application state. Based on this result, the current command function can then be exchanged.

```
commands.tags.placeOrder = ["place-order"];
commands.experimentalFn = function(request, state) { ... }

commands.optimizeAlgorithm = function(request, state) {
  var b = createBranch("beginning of timeline");
  b.changeCommandFunction("placeOrder", "experimentalFn");
  b.partialReplay({ recompute: [ "place-order" ] });

  /* compare branch state against current state */
  var branchState = b.getState();
  if (...) {
    /* exchange command function in ongoing timeline */
    changeCommandFunction("commands.placeOrder",
                          "commands.experimentalFn");
  }

  return ...;
}
```

In order to compare different processing implementations against each other, it has to be clear that the differences after a replay have occurred due to the different command processing implementation. Therefore a deterministic command replay needs to be ensured. If commands are processed in a different order, the differences might as well be due to this indeterminism. Hidden causalities need to be taken into account as well, otherwise one could mistakenly attribute differences in the outcome to the different command processing, although hidden causalities or side effects could be responsible as well.

#### Iterate Towards Target State

A further use case is to define a target state and modify the past until we reach it. We can determine if we get from a specified start state to a defined target state by applying subsequent, iterative modifications. This feature is especially interesting for problems where modifications have to be "played through" and cannot be trivially calculated. For example, the online shop could have a complex algorithm, which calculates price increases each time an order is placed. Lets say this calculation is complex. It depends on the state of the system at the time when an order is placed and takes the number of similar recent product orders into account. If we want to find out how much more orders would have had to be placed for the current price of a product to reach a specified state, we can retroactively insert orders until we reach our target state.

But this strategy is not limited to insertion or removal operations of commands and events. It can also be applied to the command processing implementation, to determine how the logic needs to be modified to reach a certain state. This process then yields the necessary modifications. In general, this can be beneficial for a lot more cases, where the behavior of the application over time needs to be considered or is time-dependent. We imagine that this feature allows for novel debugging schemes and bug fixes.

### 5.3.7 Prototype Limitations

A limitation of our prototype is that if one command in the series fails, the series will not be executed further. This limits the expressiveness, since rollbacks of commands in the series are not possible. Additionally, it is not possible to issue a hierarchical series, where parts of the series can be executed independent of each other.

The annotation, which state properties were used for the computation of a state change, is cumbersome and could probably be done by an underlying runtime engine. We are confident that this could be achieved through source code analysis, by observing which state properties are accessed and modified. Another option is to use explicit getter and setter methods when accessing properties of the state object. Such methods can record the access of state properties as well.

Last, we have by far not exhausted the possibilities for retroactive operations, which the API could expose. For example, forward deleting all events from a certain branching point could be helpful, if the objective is to append simulated events from a certain point on. This is not possible with our described prototype, but could certainly be implemented. We have restrained our description to a basic set of operations to keep the concept simple and comprehensible.

## 5.4 Summary

This chapter demonstrated the practical applicability of the concepts described in Chapter 4. In Section 4.4.2, we described two contrasting architectural modifications for event-sourced systems following a CQRS style of architecture, as means to utilize retroaction: a unified architecture and a plug-in architecture. This chapter outlined a programming model for the unified architecture. A prototypical implementation using the scenario of an online shop was described. We demonstrated how issues of retroactive systems can be addressed (e.g. by constraining possible modifications) and which new problems emerge from the application of our ideas (e.g. retroaction-aware programming). Furthermore, the central role which the delta encoding of events poses to retroaction has been highlighted. Our programming model provides the ability to access and manipulate the application's state history in a single environment. This enables a variety of possibilities, such as history-aware algorithms and novel debugging schemes. Developers can use the application's data structures, functions, and libraries in the retroactive code. They can conduct analyses of the application's history, explore alternate branches, or create history-aware algorithms.

# Chapter 6

# Summary of Architectural Considerations

Integrating retroaction in event-sourced systems introduces a set of capabilities which are not feasible in traditional event-sourced architectures. A comparison of our conceptual considerations to existing systems is difficult, since the capabilities which we propose for event-sourced systems cannot be found in this combination in the event-sourced field, nor in related work. To the best of our knowledge, there exists no work which examines retroactive computing in event-sourced systems. Thus, *it is not possible to evaluate our conceptual proposals and ideas against a comparable system in the event sourcing domain.*

If we turn to related work, there are some domains which have similar ideas. But as illustrated in Chapter 3, most of these domains utilize the recorded history of a program in a different manner than we do: decoupled from the application (debuggers), limited to passive retrospection (history-aware languages and algorithms), or on a meta level (VCSes). Oftentimes the recorded history of an application is only utilized for post hoc analysis in separate tools, after the execution has finished. In Chapter 3, we described related works and efforts to utilize retroaction in data structures and aspect oriented programming languages. These approaches have parallels to our work, but do not consider problems of temporal and causal inconsistencies, necessary restrictions, or side effects. The problem of how to handle side effects in replays is mentioned in the related work concerning reotractive aspects as well, though there is no clear solution, whereas we have provided an in-depth examination of side effects in replays and suggested ways of recording and controlling them. If side effects are outsourced into separate, individual commands, they can be partially reused or reinvoked. As we view it, these systems do not consider the challenges which arise from modifying and interacting with the application's history in a single environment. Consistency issues, causality violations, branching, and the control of side effects in replays, are some of the challenges which we have attempted to solve in the two previous chapters.

In our approach, applications can examine their state history retrospectively and modify this history as a mean to explore alternative states. Event sourcing with CQRS is a perfect match for this, since it inherently captures state changes as commands and events. In this first of part of the thesis, we identified the major challenge of retroaction in event-sourced systems. We discussed temporal and causal inconsistencies, potential solutions and necessary restrictions. Furthermore, we provided an extensive overview on limitations and constraints of retroaction in event-sourced systems. Next, we described two appropriate architectural modifications to event-sourced systems following a CQRS style of architecture. We demonstrated the applicability of both architectures by implementing them as prototypes. For the unified architecture, we described an appropriate programming model and its implementation as a prototype. In Chapter 4, we illustrated that the usage of retroactive computing is heavily dependent on the application domain and its domain-specific constraints. Some domains cannot take advantage of its full potential due to strict constraints caused by side effects, real-world coupling, or hidden causalities. Other domains on the other hand benefit heavily of retroactive aspects, as it allows for an entirely new perspective on application state. As can be seen with the other retroactive constraints as well, it is heavily dependent on the domain model how much can be made of retroaction and how high the informative value of retroactive modifications can be. If retroaction is taken into account from the start when building a system, the informative value of retroactive changes can be maximized.

# Chapter 7

# The Chronograph Platform

The foregoing chapters of this thesis described how retroactive capabilities with an ES+CQRS style of architecture can be utilized. In this chapter, we examine if and how the results from the previous chapters can be transferred to an event-sourced system with a different architectural style. The Chronograph research platform is the subject of this examination.

## 7.1 Background

Chronograph [51] is an event-sourced, distributed platform with the underlying structure of a graph. The platform thus has potential retroactive capabilities, which have not been considered so far. Events are only ever appended, but no retroaction is applied. The programming model is vertex-centric, each vertex in the graph possesses an individually specified behavior – the *behavior function.* Vertices communicate via asynchronous message passing with their neighbors, can spawn new vertices, and create or remove edges. A message addressed to a vertex triggers the processing of this message within the vertex. Such a computation can result in changes to the vertex state, modifications to the vertex edges, the spawning of new vertices, or outgoing messages. State changes are persisted as events into an event log which belongs to the vertex. Thus, event sourcing is the mean to persist vertex state. The behavior function is part of the vertex state and is also persisted with the event. It can be exchanged dynamically by the vertex. Formally, a vertex at a time $t$ can be described as reacting to an incoming message $m$ by invoking its current behavior function $f_t$. The function returns the updated state and a (possibly different) behavior function:

$$(S_{t+1}, f_{t+1}) = f_t(m, S_t)$$

A pseudo code implementation of this formal description can be written like this:

```
vertices["foo"].behaviorFn = function(state, msg) {
  // local computations
  state.foo = state.bar + 1;

  return [state, vertices["foo"].behaviorFn];
}
```

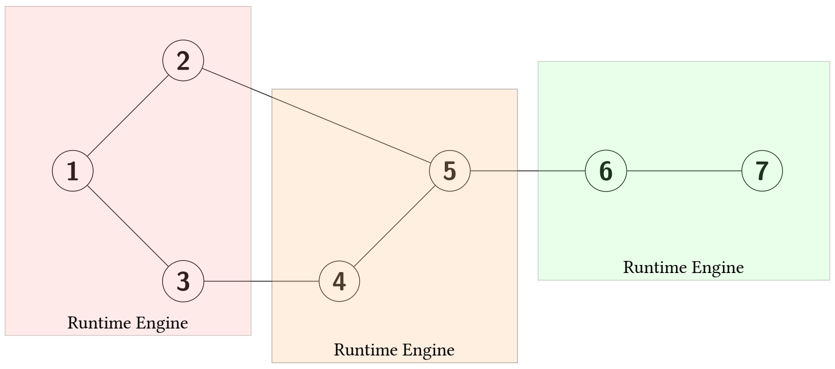


Figure 7.1: In Chronograph, vertices can be spread over multiple workers, which do not necessarily have to run on the same physical machine.

The behavior function of this vertex "`foo`" is invoked once an incoming message is available. The prior vertex state is supplied to the behavior function as an argument. The function then processes the message and returns the updated state object, as well as the subsequent behavior function.

In Chronograph, a runtime engine is responsible for providing underlying services (message passing, persistence, calculating events, etc.). Multiple instances (i.e. workers) can be connected, in order to distribute a graph over multiple machines or processes (Figure 7.1).

## 7.2 Differences to Prior Architectures

The event-sourced architectures, which we described in Chapters 4 and 5 have some differences to the Chronograph architecture. Before we examine the application of retroaction in Chronograph further, we examine these differences first.

### 7.2.1 Local vs. Global View

In Chronograph each vertex possesses a local view. It can communicate with neighbors, modify its edges, and spawn new vertices, but its computation takes only the local vertex state and arriving messages into consideration. This is basically the scenario we viewed in Chapter 4 and 5, where we considered a system which receives commands from an outside world. The IoT scenario, which we used to illustrate our conceptual considerations in Chapter 4, is a use case which suits well for Chronograph. Independent IoT devices can be modelled as vertices in a graph. The devices are connected via edges and communicate via asynchronous message passing. Devices are independent from each other; each one possesses an individual behavior function and its own timeline. Though, each device is loosely coupled to nearby devices and there might be causal dependencies among devices. In this IoT scenario, the local view refers to the view of one device on itself.

But in Chronograph a global view exists as well. Vertices are interconnected via edges and form a graph topology. This topology restrains the communication possibilities, since only neighboring vertices can communicate with each other. If retroaction is applied to the graph, new challenges emerge: There is no global timeline, logic, or event log – each vertex possesses its own. The global view only emerges when the graph is examined in its entirety. For this, multiple event logs need to be taken into account and the creation of a consistent graph snapshot is necessary. The issue of creating snapshots in distributed event-sourced has been addressed by Habiger [52] in 2015. In our IoT scenario, a global view could be calculated by a controller device which keeps track of all devices.

### 7.2.2 CQRS in Chronograph

Chronograph possesses a segregation of commands and queries as well. But the implementation differs from our prior architectures. Commands are sent by vertices to vertices, but queries are issued from an underlying platform. Queries take place outside of the graph, on a different layer than commands. Behavior functions are written by a programmer to specify the behavior of vertices. Within a behavior function, incoming messages are processed, state may be mutated, and new messages sent. Queries on the other hand are issued from the runtime engine (e.g. by an administrator) and refer to the state of the entire graph. There is usually a snapshot mechanism, which yields a consistent state of the graph, involved.

Messages in Chronograph can be considered what we previously described as commands. Here, messages which are passed between vertices trigger a computation within a vertex, and thus possibly an internal state change. To summarize: Queries and commands are also viewed as separate operations, but a query happens on the snapshot of a graph and is decoupled from the application itself. Commands, on the other hand, are sent as messages between vertices.

## 7.3 Retroaction in Chronograph

This section examines how retroaction can be achieved in the platform. For this, we apply the ideas introduced in Chapters 4 and 5, in order to utilize temporal aspects of the platform. There are a number of use case which motivate retroaction in Chronograph.

**Retroactive Insights** Past states of the graph can be explored. This can be used for analysis purposes and the results can be utilized to e.g. change the future behavior of the system.

**Compare Behavior Functions** It can be examined how different behavior functions would have affected the system. This can be achieved by replaying the messages from the graph's history, with different behavior functions for vertices.

**Messages** Messages can be retroactively inserted or deleted, to examine how they would have affected the graph's state.

**Topology Changes** Topology changes can be applied retroactively to examine the impact of a different graph topology.

An example of a retroactive application in the IoT scenario, is to retroactively explore how different behavior functions in individual devices would have affected the system state.

### 7.3.1 Behavior Function vs. Runtime Engine

In Chronograph, access to retroaction can be exposed either (1) in the behavior function or (2) in the underlying runtime engine. In case of (1), this leads to a similar programming model as in Chapter 5.

### 7.3.2 Local vs. Global

Causalities in Chronograph occur, when (1) local state is read or mutated and (2) when messages are sent or received. When messages are exchanged with neighboring vertices, they trigger computations, which in turn might modify state and dispatch new messages. In contrast to the systems in Chapters 4 and 5, this implies that causalities occur to a larger degree outside the local view of a vertex. Thus, to fully utilize the expressiveness of retroaction, the entire graph needs to be taken into account. In replays, hidden causalities can otherwise occur in neighbors of a vertex: Since the vertices outside of a vertex would not be newly computed, there is no coupling with them – messages sent to them would not be processed and thus not influence the replay. If retroactive changes are applied in only one vertex this can weaken the informative value. Side effects can be a reason for further hidden causalities.

It is actually possible to apply a lot of the ideas from the previous chapters to Chronograph – if we just examine the local view in a vertex we can apply our insights from Chapters 4 and 5. Though, hidden causalities in the graph outside of a vertex can have a limiting effect. Through the interaction of the graph with the outside world (through side effects), additional hidden causalities might exist. Thus, we cannot limit retroactive capabilities to a local vertex, the nature of the graph rather has to be taken into account when utilizing retroaction in Chronograph.

Besides the primitives of retroaction which we described in the programming model from Chapter 5, Chronograph has an additional retroactive primitive which allows to retroactively edit the topology. Transferred to the IoT scenario, this allows to explore how a different topology would have influenced the system. For example, individual devices might have chosen for different actions if neighboring devices would have been available.

## 7.4 Proposals to Implement Retroaction

Some of the concepts, which we applied for implementing retroaction in event-sourced systems, can be applied to Chronograph as well. In Chapter 4, we argued that causality violations occur when a timeline modifies its own past. Furthermore, directly editing the past contradicts the append-only nature of event-sourced systems and breaks with its benefits for e.g. scalability or traceability. Thus, we proposed to distinguish between a *main timeline* and its *branches*. We restricted direct editing to branches and retained the append-only behavior for the main timeline. The same reasons hold true for Chronograph and we propose to apply the differentiation of a main timeline with branches in Chronograph as well.

### 7.4.1 Side Effects

As described in Section 4.2.5, side effects pose a challenge to retroaction. They can have a negative effect in replays, since they can be a source for hidden causalities. This can be the reason that the system behaves unforeseeable differently in a replay. Side effects are also tricky in terms of control in replays. It has to be possible to (1) suppress their invocation, (2) reuse the result from the original invocation, and (3) reinvoke them and use the new result for the further processing. In Section 4.2.5, we proposed the concept of partial replays to impose control over side effects. To briefly recapitulate on the core insights: We excavated side effects from within application commands, instead they were placed in individual commands with few application logic. This concept allows for handling side effects with system primitives, they were invoked as commands and their results recorded as events. We also highlighted that through this splitting of large commands into a series of commands and side effect afflicted commands, three constraints follow:

1. Few logic is executed when the series is issued, only the success and failure of commands is checked. Otherwise, this logic layer would not be persisted to the timeline and would not be taken into account in a command replay.

2. Each command in this series needs to be persisted to the timeline, independent of its success or failure. This way a later replay can re-execute this series with a modified processing logic.

3. The commands in this series need to be invoked in a serializable order. A non-serialized replay may break the deterministic cause of events.

This concept can be transferred to Chronograph, by excavating side effect operations from the behavior function into separate vertices. These operations can then be dispatched via asynchronous message passing. Vertices conduct the side effect afflicted operations and return the result via asynchronous messages. This way the invocation message and the resulting data are persisted in the timeline. Thus, in a replay it is possible to control if the result from the original invocation should be reused (i.e. the persisted message which returned), or if the side effect operation should be executed again (i.e. the message sent to the vertex again).

But it is not sufficient to solely place side effect operations within separate vertices – these vertices also need a special role. They cannot be treated like normal Chronograph vertices, since a vertex which performs I/O operations cannot be restored to an arbitrary state. A vertex could e.g. perform the function of processing data from the TCP connection of a live stream; restoring an arbitrary state of this TCP connection is generally not possible. Thus the usage of *I/O vertices* becomes necessary. In terms of communication via asynchronous messages, spawning them, and programming them, these I/O vertices act like normal vertices. But they do not return or persist state. The type of a vertex – normal vertex or I/O vertex – can be supplied when it is initially spawned. The following listing depicts an example how this could be implemented. In this example, the behavior function does not utilize I/O vertices, instead "normal" processing logic in the behavior function is blended with side effect afflicted operations. A stock price is fetched from a web service and depending on the result, the state object is modified and a mail message may be sent.

```
vertices[”foo”].behaviorFn = function(state, msg) {
  var stockPrice = http.get(”https://.../price”);
  if (stockPrice < 500) {
    state.buyingRecommendation = true;
    mail.send(”alice@foo.com”, ”Buying recommendation!”);
  }

  return [state, vertices[”foo”].behaviorFn];
}
```

The problem here is that in a replay neither reusage, nor reinvoking, nor skipping of the individual side effects can be controlled. But this behavior function can be rewritten to utilize I/O vertices in this way. In the following listing two special vertices `stockPriceIO` and `mailIO` have been created. The side effect afflicted operations are moved to these vertices. The code changes to the original function are marked in red. The functionalities can then be triggered by sending messages to these vertices. Results are returned to the sender via the native asynchronous message passing primitives of Chronograph.

```
vertices[”foo”].behaviorFn = function(state, msg) {
   sendMsg(”stockPriceIO”, ”get-price”);
   if (msg.content === ”price-fetched” && msg.price < 500) {
     state.buyingRecommendation = true;
     sendMsg(”mailIO”, ”send-alert-mail”);
   }
   return [state, vertices[”foo”].behaviorFn];
}

vertices[”stockPriceIO”].behaviorFn = function(msg) {
   if (msg.content === ”get-price”) {
     var data = http.get(”https://.../get-price”);
     sendMsg(msg.sender, data);

     return [ vertices[”stockPriceIO”].behaviorFn ];
   }
}
```

In a replay, the `stockPriceIO` function could be exchanged to e.g. simulate different I/O by returning simulated stock prices from a fixed list or fetch them newly from a different server. In the above example, the function which controls the I/O vertex logic was described as a behavior function within the system. Since I/O vertices are a special type of vertex, the I/O vertex could also be provided by the underlying runtime engine. This may allow for more efficient processing or the coupling with different programming environments (or components).

### 7.4.2 Retroactive Modification

As a result of a retroactive modification, the state at a point in the vertex timeline may have changed. In Section 4.2.3 we described possibilities to ensure the consistency of the subsequent timeline. Among them were the removal or recomputation of causally dependent events. We proposed to annotate the system information which was used to compute each event for this. In our prototype, this information was defined by the reading and writing operations which accessed properties of the system state object. This annotation was done manually, by the developer.

This concept can be transferred to Chronograph: We propose for the behavior function to return the properties of the vertex state object which were utilized during an invocation of the behavior function. Though, we propose an improvement of the concept from our prototype: the usage of getter and setter methods when accessing properties of the vertex state object. These methods are provided by the runtime engine and allow to automatically capture which state properties were accessed and modified. Through this concept, the developer does not have to annotate this information by hand. Listing 7.1 illustrates how this concept can be implemented. A further improvement over the usage of getter and setter methods would be source code analysis by the underlying runtime engine, to automatically annotate this information. Developers would then not have to use setter or getter methods, but could rather use normal language syntax.

But this concept is only sufficient to track causal dependencies of a vertex in itself. This was sufficient for our prior architectures, in which only a single event log existed. In these prior architectures, commands could not invoke other commands. Commands were issued by an application and it was not clear if newly issued commands had a causal relationship to previous commands. As a consequence, causal dependencies among commands could not be traced, although they may have existed. In Chronograph, the underlying model is different. Here, vertices can send messages to neighbors. These messages trigger a computation in the receiver and result in new events being appended to individual event logs. Thus, the computation in vertices has a causal relationship to the computation of the vertex which sent the message. In case of retroaction, retroactive modifications can result in a message no longer being sent. This has consequences for the consistency of the timeline, since the neighboring vertex then would never have received a message, and thus never executed its computation. This is why direct – and recursively indirect – causal dependencies in other vertices need to be taken into account when retroactive changes are applied (Section 4.2.4). In terms of causal dependencies among messages, the concept of vector clocks could be applied. Vector clocks can be used to trace causalities among messages in a distributed system [53]. To ensure a consistent timeline, the computations in vertices which have received direct messages and sent subsequent indirect messages, need to be recursively removed (or replayed) as well.

Listing 7.1: Getter and setter methods can be used to capture causalities among computations. If a retroactive modification is applied, dependent computations can be recursively handled. In the case of e.g. a message which is retroactively removed, all consequences can be removed from the subsequent timeline as well.

```
vertices["foo"].behaviorFn = function(state, msg) {
  var f = state.get("foo"); /* state.foo */
  state.set("bar", f++);    /* state.bar */

  return [state, vertices["foo"].behaviorFn];
}
```

### 7.4.3 Tagging Events and Commands

In Section 5.2.3, we described the concept of tagging commands and events. We propose to apply this concept in Chronograph as well. Enabling the tagging of messages *and* events makes sense, since they both serve a different purpose: A message is always issued by the vertex which sends the message; the event describes how the vertex who received the message reacts to it. This resulting event depends on the behavior function of the receiver. Therefore it makes sense to annotate the resulting event with tags *in the behavior function*. Annotating messages enables us to filter for them, independent of how they are processed. For messages, tagging allows to flexibly exclude message categories (or individual messages) in replays. This can be used to exclude e.g. messages to I/O vertices, in order to prevent side effects from being reinvoked.

Concerning events, tagging allows for an additional use case: creating references to certain points in the timeline. These references can be used as timeline pointers in retroactive operations (e.g. as branch or insert marks). Concerning the implementation of tagging, we can utilize the return values of the behavior function to return tags as well. The following code example illustrates how the concept of tagging in the behavior function can work:

```
vertices[”foo”].behaviorFn = function(state, msg) {
  var eventTags = [ ”branching-point1”, ”important-data-processed” ];

  var messageTags = [ ”some-message-tag” ];
  msg.addTags(messageTags);

  return [state, vertices[”foo”].behaviorFn, eventTags];
}
```

### 7.4.4 Capturing State Changes in Events

In Chronograph, events are implicit changes to a state object. The difference before and after the modification is calculated by the underlying platform. The persisted event captures these state changes and can be used to rebuild state. But the Chronograph specification does not yet define how these deltas are encoded.

In the context of retroaction, the form how state changes are encoded is important to consider, since this has a large impact on the expressiveness and limitations of retroaction. In Section 5.2.1, we described this impact in detail. For the reasons described there, the form of JSON Patches suits better for Chronograph. Here, the patch contains the operations which were applied – opposed to the mere recording of the result. This allows for retroactive changes to *not* be annihilated by subsequent events. But the issue is not solved completely. JSON Patches record the operation which has been applied to an object only in some cases; appending an element to an array, for example. But in the case of direct value assignments or the direct modification of a value at a certain array index, the JSON Patch still annihilates retroactive modifications. This is not necessarily an issue, but it can have a limiting effect on the insertion of retroactive events and commands.

Individual instructions within commands can conflict with prior retroactive modifications. For example, in a behavior function a programmer could always decrement the value a vertex receives from a certain vertex, relying on some knowledge of the context. Through retroactive changes, this context may no longer hold and the operation thus suddenly have a different semantics. In Section 5.2.4, we provided a more detailed examination of this issue and proposed retroaction-aware programming to provide more semantics to instructions. This can be done by providing underlying functions to the developer, which resolve the ambiguity of instructions (`getLast()`

instead of `object[99]`). This way, conflicts of retroactive changes with later operations can be mitigated.

The delta encoding in an event can be used to capture the intention of a state change as well. In Section 5.2.4 we mentioned the example of encoding append operations to an array in the event, opposed to direct value assignments of array indices.

## 7.4.5 Integrating Results from Branches

There are a number of possibilities, how results from computations on branches can be integrated back into the main timeline.

**System Primitives** One possibility is an object-oriented approach, in which branches are viewed as objects with a getter method for the current state object of a vertex. This is the concept that we applied for the programming model in Chapter 5. It can be implemented as illustrated in the following listing. In this example, the results from branches are persisted to the timeline as part of the state.

```
vertices["foo"].behaviorFn = function(state, msg) {
   var timelinePointer = "timeline-tag0";
   var branchVertex = retroactive.branchVertex(timelinePointer)

   var excludeMessages = ["alert-mail"];
   var recomputeMessages = ["fetch-price", "alert-mail"];
   branchVertex.partialReplay(fromReference, excludeMessages);

   /* get state object of a certain vertex */
   var s = branchVertex.getState();

   return [state, vertices["foo"].behaviorFn];
}
```

**Timeline↔Branch Portal** Another possibility is to implement a "portal" between a branch and the timeline. Portals could be considered "magical" doorways between branch and timeline universes. Messages could then be exchanged between timeline and branch through such a portal. The following listing depicts how this can work if a branch of an entire graph is created.

```
/* on branch */
vertices["foo"].behaviorFn = function(state, msg) {
   /* send a message to the "main" graph. the portal is
      addressed as the vertex 'timelinePortal'. */
   sendMsg("timelinePortal", { to: "vertexAlice", msg: ... });
}
```

This concept though would require the introduction of a further "portal vertex" type (besides normal and I/O vertices). Furthermore, the direction in which messages can pass through the portal can be restricted in order to prevent domain-specific issues and blending of computations on branches and the timeline. When aiming for a strictly deterministic replay, communication among universes (i.e. the timeline and its branches) is discouraged. Otherwise a coupling of vertices in both universes can lead to non-deterministic behavior in replays.

### 7.4.6 Exchanging the Behavior Function

The behavior function in a vertex can be exchanged after a behavior function has been invoked. This is achieved by returning a pointer to a different behavior function. This change in the behavior function is persisted with the event. The next time the vertex receives a message, the new behavior function is invoked. The following listing illustrates this.

```
/* newBehaviorFn is a new behavior function */
var newBehaviorFn = function(state, msg) {
  // ...
  return [state, this];
}

/* the current behavior function is invoked when the vertex
   receives a message */
vertices["foo"].behaviorFn = function(state, msg) {
  /* newBehaviorFn is invoked, the next time the vertex
     receives a message */
  return [state, newBehaviorFn];
}
```

This versatile feature is considered to provide a flexible programming model in which vertices can modify their own behavior. But it poses challenges, when the behavior function of a vertex needs to be retroactively exchanged. This is due to the characteristic that the behavior function belongs to the state of a vertex and changing it can only be done by persisting an event with the modified behavior function. This is sufficient when an event replay is conducted and state is restored solely based on events. For command replays though, it is not. If one wants to replay *all commands* of a vertex with a modified behavior function, the already persisted events are not taken into account and thus one cannot insert an event to denote a change in the behavior function. Retroactively exchanging the behavior function for a certain vertex in a command replay cannot be done using solely Chronograph system primitives. This is in contrast to the programming model from Chapter 5, where the command processing implementation can be replaced without a need for new primitives. For Chronograph, there has to be some mean to describe that from this very point in the timeline on, the vertex should operate with a different behavior function. A possibility is to introduce a special type of *meta message*, which advises a vertex to change its behavior function to a specified function. Such a message can then be retroactively injected and will be taken into account in a command replay.

### 7.4.7 Message Replays

Our prior architectures possessed only a single event log (or timeline), to which events were appended in the sequence in which the commands had been invoked. For replays, we branched the timeline and utilized this order by re-executing commands in a causally equivalent order. This concept can be applied to an isolated vertex in Chronograph as well. By recomputing messages, it is possible to examine the influence of e.g. a different behavior function or different messages. In the case of a single vertex, the computation is limited to information consumption, though. Since the vertices outside of the vertex are not newly computed, there is no coupling with them – messages sent to them would not be processed and thus not influence the replay. Thus, hidden causalities in the neighbors are not taken into account.

This issue could potentially be addressed by replaying a subgraph of neighboring vertices or even the entire graph. But this leads to a number of challenges. A global message replay raises the issue of how newly yielded messages and events should be processed. For example, a vertex in a replay might reprocess a message and issue a different message than in the original timeline to a neighbor. It is unclear, how the originally sent message should be handled. It could either be sent again or not. If the original message is sent again, the new state of the vertex is still persisted as an event and an inconsistent graph may result. If it is not sent again and instead the newly computed messages are sent, the graph may still be inconsistent since its state resulted from the old message. This issue of blended original and new events (and messages) can lead to confusion if not clearly specified. There is a lot of room for possible decisions here. However, a global message replay raises a number of questions which are heavily dependent on the actual application. From our point of view, a use case can also be to not conduct global message replays. Instead, a branch of the graph at a certain snapshot could be created and the subsequent timeline after the branching point removed. The application could then continue its processing from there on or append simulated messages.

## 7.5 Limitations of Retroaction

The limitations of retroaction (as described in Section 4.3) apply to Chronograph as well. Though, the degree to which they have a limiting effect is influenced heavily by the actual application and the context in which retroaction is applied. This section briefly recapitulates limitations of retroaction.

**Hidden Causalities** In Section 4.3.1 we examined hidden causalities. In an ideal case for retraction, there are no hidden causalities in an event-sourced systems. Thus, when a retroactive change is made to the timeline and a replay is conducted, the outcome is exactly as it would have been if the retroactive change had originally been in place. But through side effects or real-world coupling, hidden causalities can occur outside of Chronograph, making it impossible to track them. These hidden causalities impact the informative value which can be derived from the results of retroactive changes. If only a subgraph is examined and computations in the rest of the graph are not reprocessed, hidden causalities in the rest of the graph might also not be taken into account.

**Causality Violations** Causality violations can occur through paradoxes (such as causal loops). We described this issue in detail in Section 4.3.2.

**Message Semantics** The semantics of messages may annihilate retroactively applied modifications (see Section 4.3.3). This is not necessarily a limitation, but needs to be taken into account when applying retroaction.

**Determinism in Replays** The causal equivalence of replays can be another limiting factor for retroaction, as it affects the outcome of replays (Sections 4.3.4 and 7.4.7). Indeterminism can weaken the comparability of results from computations on branches.

**Decoupled Retroaction** Retroaction in large graphs may come with high costs of performance, since it requires the creation of snapshots and replay mechanisms. If the system advances, whilst retroaction is applied in a decoupled component, the results may be longer longer be valid once they are available to the system (Section 4.3.5). Thus it can be important for retroaction to be efficiently computable.

**Type of Diffs** The delta encoding of events can have a limiting impact on retroaction as well (see Sections 5.2.1 and 7.4.4). In Chronograph, it poses an essential role to retroaction and is connected to issues of retroaction-aware programming.

## 7.6 Further Research Areas

Some challenging areas with room for further interesting research emerged in this chapter. These challenges are out of the scope for this thesis, but we list them here as reference for future works in the project.

**Annotation of Causal Dependencies** In our proposals, we used getter and setter methods when accessing the state object. This allows the runtime engine to track causalities among events. The question arises if causalities between events can be annotated automatically by an underlying runtime engine. We are confident that this can be achieved through source code analysis and by examining sent and received messages, but this issue needs to be investigated further.

**Serializable Graph Replay** The issue of a serialized, causally equivalent message replay of the entire graph arose. If a replay is non-deterministic, the order of processing can otherwise result in entirely different states. Chronograph has the underlying model of distributed, asynchronous processing and a multitude of vertices with independent event logs. In order to realize a serialized global replay, all event logs need to be combined. This is no trivial issue and further research is required here.

## 7.7 Summary

This chapter examined how the conceptual considerations from Chapter 4 and 5 can be applied to the Chronograph platform. As illustrated in these earlier chapters, applying retroaction in event-sourced systems can get complex fast. Thus, it is important to clearly define the context in which it is applied and the objectives which should be reached.

In this chapter, we illustrated that there is a broad spectrum of how to apply retroaction in Chronograph and individual decisions depend heavily on the actual application. We described the introduction of a special I/O vertex type, causal relationship annotations via tags, retroaction-aware programming, and the impact of the delta encoding. Moreover, we proposed platform-specific solutions, such as I/O vertices or portals. What can be transferred to Chronograph as well, are the limitations of retroaction. They influence the expressiveness and informative value of retroaction heavily. If taken into account when creating a system their limitations can be reduced.

# Chapter 8

# Results and Conclusion

This thesis provides an extensive examination of the concept of retroactive computing in event-sourced systems. We identified key challenges, proposed solutions, and demonstrated their applicability. This chapter provides our concluding remarks.

## 8.1 Summary

Event sourcing is a perfect match for retroactive computing, since it inherently captures state changes as modifications. Thus, prior states can be restored or retroactively modified. The Chapters 2 and 3 described the field, related work, and delimitations of existing work to our objectives. In Chapter 4 we identified major issues of retroaction, such as temporal and causal inconsistencies, or side effects. We illustrated that restraining direct editing operations to branches of a timeline can prevent causality issues. Furthermore, it is possible to retain traceability and append-only behavior in event-sourced systems if results from retroaction are integrated back into the system as events.

Considering commands in replay scenarios, we discussed several possibilities of imposing control over potential side effects, and concluded by introducing partial replays, which require the tracking of causalities among events. The command and event primitives of event sourcing with CQRS are an ideal match for imposing control over side effects. Through events, side effects can be recorded and reused. Through the sourcing of commands, side effects can be reinvoked. If side effects are outsourced into separate, individual commands, they can be partially reused or reinvoked.

Next, we discussed a number of ideas for keeping a consistent timeline after a retroactive modification. For this, we transferred ideas from time travel theory to retroaction in event-sourced systems. Corresponding to parallel universe theories, we proposed the prohibition of editing the system's own timeline, while allowing direct retroactive modifications only for branches. Analogous to the self-consistency principle, we proposed validation conditions and the replay (or removal) of causally related events. Furthermore, this chapter contributed an overview on constraints and limitations of retroaction in event-sourced systems. We highlighted the central role of hidden causalities through real-world coupling or through side effects. Further limitations, which we identified, were the influence of causality violations, causally equivalent replays, and the semantics of commands which may annihilate retroactive modifications. The performance of retroactive computations can be a further limiting factor. Our conceptual considerations were concluded with the suggestion of architectural modifications (e.g. resolving the command/query segregation for retroaction) and the demonstration how different architectures can be used for enabling retroactive capabilities of event-sourced applications. We argued that it is heavily dependent on the actual application domain how much can be made of retroaction and how high the informative value of retroactive modifications can be. If retroaction is taken into account from the start when building a system, the informative value of retroactive changes can be maximized and limitations reduced.

Chapter 5 demonstrated the practical applicability of the concepts described in Chapter 4. We outlined a programming model for the unified architecture. Moreover, a prototypical implementation was described using the scenario of an online shop. We demonstrated how issues of retroactive systems can be addressed in a programming model (e.g. by constraining possible modifications) and which new challenges emerge from the application of our ideas (e.g. retroaction-aware programming). Furthermore, the central role of the delta encoding of events was highlighted for its effect on retroaction. Our programming model provides the ability to access and manipulate the application's state history in a single environment. This enables a variety of possibilities, such as history-aware algorithms and novel debugging schemes. Developers can use the application's data structures, functions, and libraries in the retroactive code. They can conduct analyses of the application's history or explore alternate branches.

Integrating retroaction in event-sourced systems introduces a set of capabilities which are not feasible in traditional event-sourced architectures. Chapter 6 illustrated that a comparison of our conceptual and architectural considerations to existing systems is difficult, since the capabilities which we propose for event-sourced systems cannot be found in this combination in the event-sourced field, nor in related work.

In Chapter 7, we applied the ideas described in Chapters 4 and 5 to the Chronograph research platform. Many of the described concepts can be applied there as well: timeline pointers via tagging, the delta encoding of events, retroaction-aware programming, or tracking causalities, for example. Moreover, we proposed platform-specific solutions, such as I/O vertices or portals. An additional element that can be transferred to Chronograph consists of the limitations of retroaction. These limitations pose a significant restriction to the expressiveness and informative value of retroaction. They can be reduced when taken into account while creating a system.

## 8.2 Outlook and Future Work

Retroactive computing allows for self-improving, self-modifying, self-learning, and self-referential algorithms in event-sourced systems. Applications can adapt their future behavior by analyzing their own past. An area where this might have interesting applications is the usage of retroactive computing in event-sourced systems for machine learning, artificial intelligence, or data mining. The application's history could be used as input to optimize parameters of an algorithm. In a further step, updated or modified versions of algorithms could be evaluated and compared against each other by replaying commands from the timeline in experimental branches.

Future work on the topic should further analyze which domains could benefit of retroactive computing. For this, more event-sourced architectures which enable retroactive capabilities need to be taken into consideration, apart from the two architectures that we examined.

## 8.3 Conclusion

To the best of our knowledge, we provide the first detailed examination of the concept of retroactive computing in event-sourced systems. The utilization of retroaction in event-sourced systems enables a set of features that cannot be accomplished with traditional application architectures. Applications cannot only analyze their own history retrospectively, they can also actively modify their past in order to explore alternative states. We illustrated that the usage of retroactive computing is heavily dependent on the application domain and its domain-specific constraints. Some domains cannot take advantage of its full potential due to strict constraints caused by side effects, real-world coupling, or hidden causalities. Other domains on the other hand benefit heavily of retroactive aspects, as it allows for an entirely new perspective on application state.

# References

[1] Martin Fowler. *Event Sourcing*. Dec. 12, 2005. URL: http://martinfowler.com/eaaDev/EventSourcing.html.

[2] Gregor Hohpe. *Programming Without a Call Stack – Event-driven Architectures*. Technical Report. eaipatterns.com, 2006.

[3] Benjamin Erb and Frank Kargl. "Combining Discrete Event Simulations and Event Sourcing". In: ICST, Aug. 2014. DOI: 10.4108/icst.simutools.2014.254624.

[4] Martin Fowler. *Patterns of Enterprise Application Architecture*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2002. ISBN: 0321127420.

[5] Martin Fowler. *Retroactive Event*. Dec. 12, 2005. URL: http://martinfowler.com/eaaDev/RetroactiveEvent.html.

[6] Jérémie Chassaing. *Event Sourcing vs Command Sourcing*. July 2013. URL: http://thinkbeforecoding.com/post/2013/07/28/Event-Sourcing-vs-Command-Sourcing.

[7] Johannes Seitz Benjamin Reitzammer. *Event Sourcing in Practice*. Dec. 2013. URL: http://ookami86.github.io/event-sourcing-in-practice.

[8] Greg Young. *CQRS, Task Based UIs, Event Sourcing agh!* Feb. 2010. URL: http://codebetter.com/gregyoung/2010/02/16/cqrs-task-based-uis-event-sourcing-agh.

[9] Greg Young. *CQRS Documents*. Oct. 2013. URL: https://cqrs.files.wordpress.com/2010/11/cqrs_documents.pdf.

[10] D. Betts, J. Dominguez, G. Melnik, F. Simonazzi, M. Subramanian, and G. Young. *Exploring CQRS and Event Sourcing: A Journey Into High Scalability, Availability and Maintainability with Windows Azure*. Microsoft Developer Guidance, 2013. ISBN: 9781621140160.

[11] Martin Krasser. *Introduction to Akka Persistence*. 2013. URL: http://krasserm.blogspot.de/2013/12/introduction-to-akka-persistence.html.

[12] Martin Krasser. *Event Sourcing and CQRS with Akka Persistence and Eventuate*. Presented at the Reactive Systems Hamburg, 2015. URL: https://www.youtube.com/watch?v=vFVry457XLk.

[13] U.S. Securities and Exchange Commission. *Electronic Storage of Broker-Dealer Records*. 2003. URL: http://www.sec.gov/rules/interp/34-47806.htm.

[14] J. Chris Anderson, Jan Lehnardt, and Noah Slater. *CouchDB: The Definitive Guide*. 1st. O'Reilly Media, Inc., 2010.

[15] Event Store Documentation. *Event Sourcing Basics.* 2015. URL: http://docs.geteventstore.com/introduction/event-sourcing-basics.

[16] Martin Fowler. *The LMAX Architecture.* 2011. URL: http://martinfowler.com/articles/lmax.html.

[17] Bertrand Meyer. *Object oriented Software Construction.* 1st. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1988. ISBN: 0136290493.

[18] Armando Fox and Eric Brewer. "Harvest, Yield, and Scalable Tolerant Systems". In: *Hot Topics in Operating Systems, 1999. Proceedings of the Seventh Workshop on.* IEEE. 1999, pp. 174–178.

[19] Jonas Bonér, Dave Farley, Roland Kuhn, and Martin Thompson. *The Reactive Manifesto.* 2013. URL: http://www.reactivemanifesto.org.

[20] Lukas Hauser. "Distributed Architecture using Event Sourcing & Command Query Responsibility Segregation". Bachelor's Thesis. Institute of Distributed Systems, Ulm University, 2014.

[21] Eric Evans. *Domain-Driven Design – Tackling Complexity in the Heart of Software.* Addison-Wesley, Sept. 2003. ISBN: 0321125215.

[22] Greg Young. *CQRS and Event Sourcing.* Presented at the Code on the Beach Conference, Florida, 2014. URL: https://www.youtube.com/watch?v=JHGkaShoyNs.

[23] Arun Kejariwal, Alexandru Nicolau, and Constantine D. Polychronopoulos. "History-aware Self-Scheduling". In: *Proceedings of the 2006 International Conference on Parallel Processing.* ICPP '06. Washington, DC, USA: IEEE Computer Society, 2006, pp. 185–192. ISBN: 0769526365. DOI: 10.1109/ICPP.2006.49.

[24] Hai Nguyen, Gonzalo Zarza, Daniel Franco, and Emilio Luque. "History-Aware Adaptive Routing Algorithm For Energy Saving in Interconnection Networks". In: *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA).* The Steering Committee of The World Congress in Computer Science, Computer Engineering and Applied Computing (WorldComp). 2013, p. 450.

[25] Todd A Proebsting and Benjamin G Zorn. *Tangible Program Histories.* Technical Report MSR-TR-2000-54. Microsoft Research, May 2000.

[26] C.W. Fraser, T.A. Proebsting, and B.G. Zorn. *Program History in a Computer Programming Language.* US Patent 7,111,283. Sept. 2006.

[27] Nir Ailon, Bernard Chazelle, Kenneth L Clarkson, Ding Liu, Wolfgang Mulzer, and C Seshadhri. "Self-improving Algorithms". In: *SIAM Journal on Computing* 40.2 (2011), pp. 350–375.

[28] Jürgen Schmidhuber. "Gödel Machines: Fully Self-referential Optimal Universal Self-improvers". In: *Artificial general intelligence.* Springer, 2007, pp. 199–226.

[29] Guillaume Pothier, Éric Tanter, and José Piquer. "Scalable Omniscient Debugging". In: *ACM SIGPLAN Notices.* Vol. 42. 10. ACM. 2007, pp. 535–552.

[30] Adrian Lienhard, Tudor Gîrba, and Oscar Nierstrasz. "Practical Object-Oriented Back-in-Time Debugging". In: *Lecture Notes in Computer Science* 5142 (2008), pp. 592–615.

[31] Debugging with GDB. *Reverse Execution.* 2016. URL: https://sourceware.org/gdb/current/onlinedocs/gdb/Reverse-Execution.html.

[32] GDB Documentation. *Process Record Tutorial.* 2009. URL: http://www.sourceware.org/gdb/wiki/ProcessRecord/Tutorial.

[33] Patrick Sabin. "Implementing a Reversible Debugger for Python". Diploma Thesis. Faculty of Informatics, TU Wien, 2010.

[34] Bret Victor. *Inventing on Principle.* Presented at the Canadian University Software Engineering Conference (CUSEC), 2012. URL: https://vimeo.com/36579366.

[35] Robin Salkeld, Wenhao Xu, Brendan Cully, Geoffrey Lefebvre, Andrew Warfield, and Gregor Kiczales. "Retroactive Aspects: Programming in the Past". In: *Proceedings of the Ninth International Workshop on Dynamic Analysis.* WODA '11. Toronto, Ontario, Canada: ACM, 2011, pp. 29–34. ISBN: 9781450308113. DOI: 10.1145/2002951.2002960.

[36] Robin Salkeld and Ronald Garcia. "Essential Retroactive Weaving". In: *Companion Proceedings of the 14th International Conference on Modularity.* MODULARITY Companion 2015. Fort Collins, CO, USA: ACM, 2015, pp. 52–57. ISBN: 9781450332835. DOI: 10.1145/2735386.2736751.

[37] Erik D. Demaine, John Iacono, and Stefan Langerman. "Retroactive Data Structures". In: *ACM Trans. Algorithms* 3.2 (May 2007). ISSN: 1549-6325. DOI: 10.1145/1240233.1240236.

[38] Nicholas J.J. Smith. "Time Travel". In: *The Stanford Encyclopedia of Philosophy.* Ed. by Edward N. Zalta. Spring 2016. 2016. URL: http://plato.stanford.edu/archives/spr2016/entries/time-travel.

[39] M. Rea. *Metaphysics: The Basics.* The Basics. Taylor & Francis, 2014. ISBN: 9781317756057.

[40] V. Frolov and I. Novikov. *Black Hole Physics: Basic Concepts and New Developments (Fundamental Theories of Physics).* 1st ed. Vol. 96. Fundamental Theories of Physics. Dordrecht: Kluwer Academic Publishers, Nov. 1998. ISBN: 0792351460.

[41] Stewart Robinson. *Simulation: The Practice of Model Development and Use.* Wiley, 2004. ISBN: 9780470092781.

[42] Raghu Ramakrishnan and Johannes Gehrke. *Database Management Systems.* 3rd ed. New York, NY, USA: McGraw-Hill, Inc., 2003. ISBN: 0072465638, 9780072465631.

[43] Alberto Apostolico, Giuseppe F. Italiano, Giorgio Gambosi, and Maurizio Talamo. "The Set Union Problem With Unlimited Backtracking". In: *SIAM J. Comput.* 23 (1994), pp. 50–70.

[44] Melanie Swan. *Blockchain: Blueprint for a New Economy.* " O'Reilly Media, Inc.", 2015.

[45] Simon Peyton Jones. "Tackling the Awkward Squad: monadic input/output, concurrency, exceptions, and foreign-language calls in Haskell". In: *Engineering Theories of Software Construction.* Press, 2001, pp. 47–96.

[46] Paul C. Bryan and Mark Nottingham. *JavaScript Object Notation (JSON) Patch.* IETF RFC 6902. 2013.

[47] *The JavaScript Object Notation (JSON) Data Interchange Format.* IETF RFC 7159. 2014.

[48] Roy Thomas Fielding. "Architectural Styles and the Design of Network-based Software Architectures". PhD Dissertation. University of California, Irvine, 2000.

[49] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. *RFC 2616, Hypertext Transfer Protocol – HTTP/1.1.* 1999.

[50] Roy T. Fielding and Julian F. Reschke. *Hypertext Transfer Protocol (HTTP/1.1): Conditional Requests.* IETF RFC 7232. 2014.

[51] Benjamin Erb and Frank Kargl. "A Conceptual Model for Event-sourced Graph Computing". In: *Proceedings of the 9th ACM International Conference on Distributed Event-Based Systems.* DEBS '15. Oslo, Norway: ACM, 2015, pp. 352–355. ISBN: 9781450332866. DOI: 10.1145/2675743.2776773.

[52] Gerhard Habiger. "Distributed Versioning and Snapshot Mechanisms on Event-Sourced Graphs". Master's Thesis. Institute of Distributed Systems, Ulm University, Oct. 2015.

[53] Reinhard Schwarz and Friedemann Mattern. "Detecting Causal Relationships in Distributed Computations: In Search of the Holy Grail". In: *Distributed Computing* 7.3 (1994), pp. 149–174.

# Colophon

This thesis was typeset using the XƎTEX engine. The microtype package is responsible for micro-typographic adjustments concerning the color of the text, line breakages, and the addition of character protrusion. TikZ has been used for the illustrations. The font used for the text and headings is Philipp Poll's Linux Libertine. Some of the beautiful open type features which the typeface provides – such as old-style figures and swashes – have been activated. The font used for the monospaced code listings is Paul Hunt's `Source Code Pro`, which he developed for Adobe. Both fonts are provided under free licenses.

The quote used on page vii is taken from Grimm's Household Tales by Jacob and Wilhelm Grimm. Their collection of fairy tales was first published in 1812 and translated from German to English in 1884 by Margaret Hunt[1]. The text is part of the public domain. It is typeset in the Linux Libertine font as well, though with historical ligatures.

The title page typeface is Knuth's Computer Modern Sans Serif. The illustration on the title page is titled »*Hänsel and Gretel*«; it was created by Alexander Zick (1845-1907) and is in the public domain[2].

The choice of the title illustration and the subsequent quote was motivated by parallels in event-sourced systems and this certain fairytale. Pigeons ate their breadcrumb trail and thus Hänsel and Grethel were unable to find their way back home. The objective of an event-sourced system is to never get to a point where it is not possible to reconstruct how the system got there. This thesis shows that there is much more to an event-sourced system. Not only can we rebuild how we got somewhere, we can also explore which other houses we could have reached taking different trails through the forest. How the story could have gone differently, if Hänsel and Gretel had had an event-sourced system with retroactive capabilities...

[1]https://en.wikisource.org/wiki/Page:Grimm%27s_Fairy_Tales.djvu/355
[2]https://commons.wikimedia.org/wiki/File:Hänsel_und_Gretel.jpg