NATIONAL AND KAPODISTRIAN UNIVERSITY OF ATHENS

SCHOOL OF SCIENCE
DEPARTMENT OF INFORMATICS AND TELECOMMUNICATION

BSc THESIS

Modelling business workflows using version control primitives atop DOLAR

Spyros D. Trifonidis

Supervisors: A. Delis, Professor
K. Saidis, Dr.

ATHENS
JANUARY 2024

ΕΘΝΙΚΟ ΚΑΙ ΚΑΠΟΔΙΣΤΡΙΑΚΟ ΠΑΝΕΠΙΣΤΗΜΙΟ ΑΘΗΝΩΝ

ΣΧΟΛΗ ΘΕΤΙΚΩΝ ΕΠΙΣΤΗΜΩΝ
ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΤΗΛΕΠΙΚΟΙΝΩΝΙΩΝ

ΠΤΥΧΙΑΚΗ ΕΡΓΑΣΙΑ

Μοντελοποίηση διαδικασιών χρησιμοποιώντας στοιχεία ελέγχου εκδόσεων με το DOLAR

Σπύρος Δ. Τρυφωνίδης

Επιβλέποντες: Α. Δελής, Καθηγητής
Κ. Σαΐδης, Δρ.

ΑΘΗΝΑ
ΙΑΝΟΥΆΡΙΟΣ 2024

BSc THESIS

Modelling business workflows using version control primitives atop DOLAR

Spyros D. Trifonidis
S.N.: 1115201600175

SUPERVISORS: A. Delis, Professor
K. Saidis, Professor

ΠΤΥΧΙΑΚΗ ΕΡΓΑΣΙΑ

Μοντελοποίηση διαδικασιών χρησιμοποιώντας στοιχεία ελέγχου εκδόσεων με το DOLAR

Σπύρος Δ. Τρυφωνίδης
Α.Μ.: 1115201600175

ΕΠΙΒΛΕΠΟΝΤΕΣ: Α. Δελής, Καθηγητής
Κ. Σαϊδης, Καθηγητής

ABSTRACT

Version control systems (VCS) are essential tools for tracking changes in software source code. However, their utility extends beyond the traditional boundaries of software engineering. Recognizing the benefits of version control for a broader spectrum of applications, this thesis presents the design and implementation of a framework that allows business application developers to integrate version control capabilities into the implementation of business workflows with minimal overhead. The framework capitalizes on the DOLAR framework's functionalities to provide a data store agnostic, version controlled object database, offering an efficient way to manage changes in the workflow driver datasets.

Through the proposed implementation, this thesis addresses both the challenge of introducing version control into new applications and the complexity of integrating it into existing legacy systems. The underlying structure of our approach comprises adaptations of VCS principles, using constructs such as repositories, commits, and branches to handle changes in application data. The evaluation of our approach includes detailed real-world scenarios, demonstrating how it could elegantly support processes such as academic thesis submissions and generic form submissions within business applications.

In assesing the practical implementation of our approach, the thesis showcases how such a version control system can serve to reduce complexity, enhance transparency in changes, and facilitate smoother workflows. While acknowledging the trade-offs made in terms of increased storage overhead and impact on search systems, the thesis concludes by emphasizing the framework's contribution to application development and its potential for future enhancements, such as conflict detection and resolution, extended user interface options, and richer VCS operations.

SUBJECT AREA: Version Control Systems
KEYWORDS: version control systems, data modelling

ΠΕΡΙΛΗΨΗ

Τα συστήματα ελέγχου εκδόσεων (ΣΕΕ) ειναι απαραίτητα εργαλεία για την καταγραφή αλλαγών στον πηγαίο κώδικα έργων λογισμικού. Παρ’ όλα αυτα, η χρησιμοτητά τους εκτείνετε εκτός της συνιθυσμένης χρήσης τους στην παραγωγή λογισμικού. Αναγνωρίζοντας τα προτερήματα του ελέγχου εκδόσεων σε μια πιο ευρεία γκάμα απο εφαρμογές, αυτή η εργασία παρουσιάζει την σχεδίαση και υλοποίηση ενός εργαλείου το οποίο επιτρέπει σε προγραμματιστές να ενσωματώσουν δυνατότητες ελέγχου εκδόσεων στο λογισμικό τους με μικρό κόστος. Το εργαλείο εκμεταλεύεται τις δυνατότητες του DOLAR για να προσφέρει μια αγνωστική ως προς την τεχνολογία αποθήκευσης, βάση αντικειμένων με δυνατότητες ελέγχου εκδόσεων, το οποίο επιτρέπει εύκολες αλλαγές στο μοντέλο δεδομένων.

Μέσα απο την υλοποίηση της, αυτή η εργασία αντιμετωπίζει και την δυσκολία της εισαγωγής δυνατοτήτων ελέγχου εκδόσεων σε νέες εφαρμογές καθώς και την περιπλοκότητα της εισαγωγής τους σε ήδη υπάρχοντα προγράμματα. Η αρχιτεκτονική της προσέγγισης αποτελείτε απο υλοποιήσεις δομικών στοιχείων ενώς συστήματος ελέγχου εκδόσεων όπως αποθετήρια, καταγραφές, και διακλαδώσεις για να χειριστεί τις αλλαγές στα δεδομένα της εφαρμογής. Η αποτίμηση της προσέγγισης συμπεριλαμβάνει πραγματικά σενάρια όπως την εναπόθεση ακαδημαϊκών εργασιών και γενικά την υποβολή φορμών στα πλαίσια μια εφαρμογής.

Χρησιμοποιώντας την υλοποίηση της προσέγγισης, η εργασία αναδεικνύει πως ενα τέτοιο σύστημα ελέγχου εκδόσεων μπορεί να μειώσει την περιπλοκότητα της υλοποίησης, να ενισχύσει την διαφάνεια των αλλαγών σε δεδομένα, και να διευκολύνει την διαδικασία των εργασιών που προσφέρει το σύστημα. Ενω αναγνωρίζει τους συμβιβασμούς που έγιναν ως προς την επαυξημένη ανάγκη για αποθηκευτικούς πόρους και την περιπλοκή των συστημάτων αναζήτησης, η εργασία καταλήγει τονίζωντας την συνδρομή του εργαλείου στην ανάπτυξη εφαρμογών και την δυνατότητα για μελλοντικές βελτιώσεις, όπως την ανίχνευση λύση συγκρούσεων, περισσότερες διεπαφές για το χρήστη, και πιο εμπλουτισμένες λειτουργίες ενός ΣΕΕ.

ΘΕΜΑΤΙΚΗ ΠΕΡΙΟΧΗ: Συστήματα Ελέγχου Εκδόσεων
ΛΕΞΕΙΣ ΚΛΕΙΔΙΑ: συστήματα ελέγχου εκδόσεων, μοντελοποίηση δεδομένων

CONTENTS

1. INTRODUCTION

Version control systems are ubiquitous in modern software development. From software engineering classes in universities to globally distributed software engineering teams, version control systems help engineers work with and reason about the changes in their program's source code across its life cycle. Their building blocks, like commits and branches, are simple and allow engineers to work on coherent sets of changes without interfering with each other. Moreover, they serve as an archive for each and every change the source of a program has gone through. Thus, they enable engineers to inspect the history of changes of their source code and the reasoning behind them. The history and fundamentals of modern version control systems are covered in Chapter 4.

Software engineering is not the only domain that has benefited from version control systems. Authoring software like word processors and content management systems often employ version control in the form of revision control, by keeping a linear history of past versions of a document. Likewise, many business applications could benefit by a version control system similar to those used by software engineers. We believe that applications with submission and review cycles or applications that deal with complex business object hierarchies are prime candidates for the usage of version control in their data models. More details about such applications are discussed in Chapter 3. In this context, application developers need to take the need for a version control system into consideration from the start of the development process. Furthermore, implementing version control in a legacy system is often downright impossible without rewriting large portions of the system [1]. Thus, the costs associated with building version control capabilities in business applications is often prohibitive.

This thesis presents the implementation of a version control system layer that can be used by downstream applications, like applications that implement business workflows. Specifically, this thesis is implemented as an extension to the DOLAR project [2]. DOLAR abstracts the details of storing business objects and provides a uniform interface for interacting with them. The primitives DOLAR provides will be presented in Chapter 5. Furthermore, the way this thesis leverages DOLAR will be analysed in Chapter 6. Finally, Chapter 7 discusses the perceived benefits of the approach in the context of the real world scenarios outlined before.

2. RELATED WORK

There have been various efforts aimed at bringing VCS features to a more diverse set of applications. The most similar one to the implementation of this thesis is *CoreObject* [3], which also implements a version controlled object database. The main differences with *CoreObject* are the backing databases and target applications. *CoreObject* uses SQLite whereas the implementation of this thesis is storage agnostic. Additionally, *CoreObject* is aimed at native desktop applications while the implementation of this thesis is geared towards web applications.

Another relevant project is the *XTDB* database [4]. XTDB is a bi-temporal database system meaning that data is stored across two axes of time. These axes of time are the *valid time*, which is the application defined time during which the object was valid and *transaction time* which is the time the object was inserted into the system. XTDB models objects as immutable records. Thus, users can query records across every state they have ever been in. This allows users to implement features like a version control system with relative ease. Similarly, bi-temporal extensions exist for traditional relational database management systems (RDBMS). One such extension is the *temporal_tables* [5] extension for the PostgreSQL RDBMS. While bi-temporal databases are very useful they operate on a lower abstraction layer than this thesis which provides a dedicated version controlled object database. Thus, such systems have no notion of commits, branches or merges and their implementation depends on the application developer should they be needed.

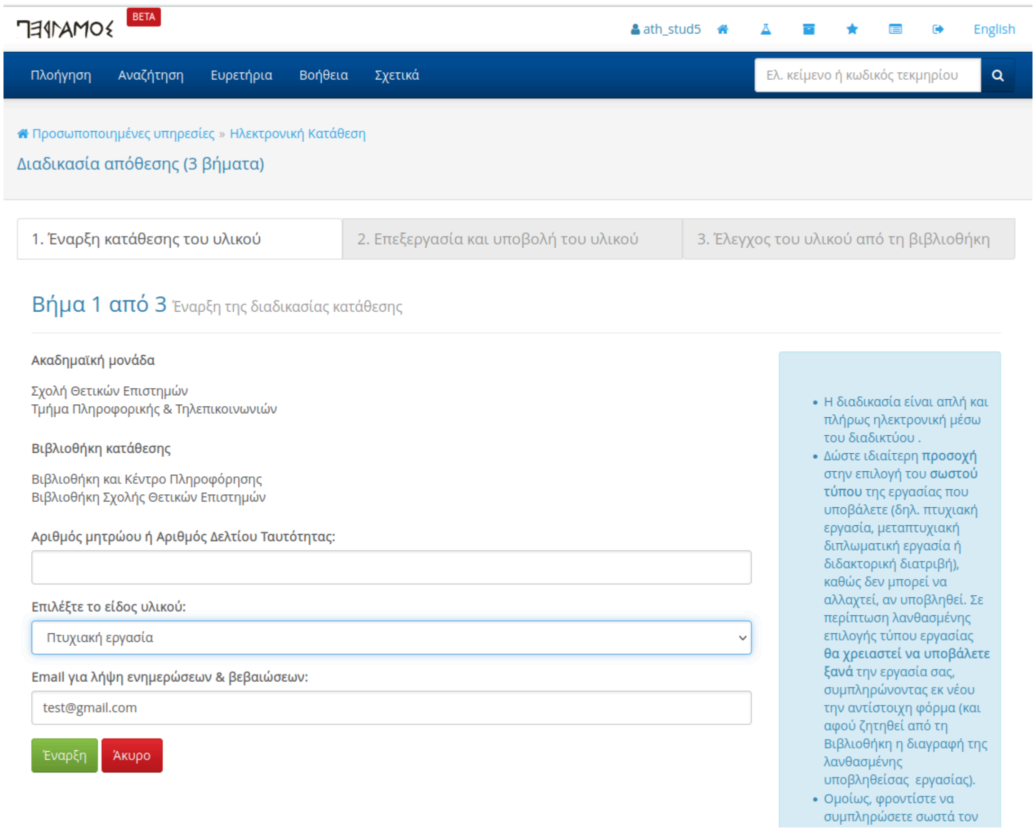
Other systems used by the industry support some simple forms of object versioning. The *Simple Storage Service* or *S3* provided by *Amazon Web Services* supports keeping track of each version of an object stored in it [6] as does the similar *Cloud Storage* service by *Google Cloud Platform* [7]. Alternatively simple versioning can be achieved in traditional RDBMS using extensions such as the *table_version* [8] extension for PostgreSQL which tracks the history of rows in a table. Such systems are simple in their operation and effective if applications only need history tracking and simple restoration of older versions but like bi-temporal databases lack any higher level VCS features.

3. MOTIVATION

Implementing version control capabilities in business software is expensive. This cost stems from the complexity of implementing the data model of a VCS on top of an already sophisticated business domain data model. Hence, equipping software engineers with tools that manage that complexity can enable the creation of business applications featuring VCS capabilities. Moreover, tools that allow the retroactive introduction of VCS capabilities in existing software can increase the usability of legacy software without large scale rewrites. To this end, this thesis aims to implement such a tool, namely a version controlled object database, by leveraging the DOLAR project. We next briefly present Pergamos, the digital repository of the University of Athens, as a use case for the realization of our proposal.

3.1. Pergamos thesis submission workflow

Business applications can often benefit greatly from the use of version control features. Consider the scenario of a thesis submission to the library of a university for review and publishing. Let's take for example Pergamos [9], the digital library of the University of Athens, that manages theses written by students at the university. Firstly, the student submits their draft to Pergamos for review (Figure 1).


BETA
ath_stud5
English
Πλοήγηση
Αναζήτηση
Ευρετήρια
Βοήθεια
Σχετικά
Ελ. κείμενο ή κωδικός τεκμηρίου
Προσωποποιημένες υπηρεσίες » Ηλεκτρονική Κατάθεση

Διαδικασία απόθεσης (3 βήματα)

1. Έναρξη κατάθεσης του υλικού | 2. Επεξεργασία και υποβολή του υλικού | 3. Έλεγχος του υλικού από τη βιβλιοθήκη

Βήμα 1 από 3 Έναρξη της διαδικασίας κατάθεσης

Ακαδημαϊκή μονάδα

Σχολή Θετικών Επιστημών
Τμήμα Πληροφορικής & Τηλεπικοινωνιών

Βιβλιοθήκη κατάθεσης

Βιβλιοθήκη και Κέντρο Πληροφόρησης
Βιβλιοθήκη Σχολής Θετικών Επιστημών

Αριθμός μητρώου ή Αριθμός Δελτίου Ταυτότητας:

Επιλέξτε το είδος υλικού: Πτυχιακή εργασία

Email για λήψη ενημερώσεων & βεβαιώσεων: test@gmail.com

- Η διαδικασία είναι απλή και πλήρως ηλεκτρονική μέσω του διαδικτύου .
- Δώστε ιδιαίτερη **προσοχή** στην επιλογή του **σωστού τύπου** της εργασίας που υποβάλετε (δηλ. πτυχιακή εργασία, μεταπτυχιακή διπλωματική εργασία ή διδακτορική διατριβή), καθώς δεν μπορεί να αλλαχτεί, αν υποβληθεί. Σε περίπτωση λανθασμένης επιλογής τύπου εργασίας **θα χρειαστεί να υποβάλετε ξανά** την εργασία σας, συμπληρώνοντας εκ νέου την αντίστοιχη φόρμα (και αφού ζητηθεί από τη Βιβλιοθήκη η διαγραφή της λανθασμένης υποβληθείσας εργασίας).
- Ομοίως, φροντίστε να συμπληρώσετε σωστά τον

Figure 1: The first step of the Pergamos thesis submission interface

In turn, Pergamos stores the draft and notifies the relevant reviewer. Next, the reviewer reviews the draft submission and decides to accept or reject it with comments for why it was rejected. Upon rejection, Pergamos notifies the submitter with the comments the reviewer made. Then, the submitter fixes their submission and resubmits it to Pergamos which notifies the reviewer again. Finally, after the submitter has fixed all the comments made by the reviewer and the reviewer has accepted the submission the thesis is recorded in the Pergamos database. This processes is described graphically in Figure 2. Once published, the thesis is accessible publicly.

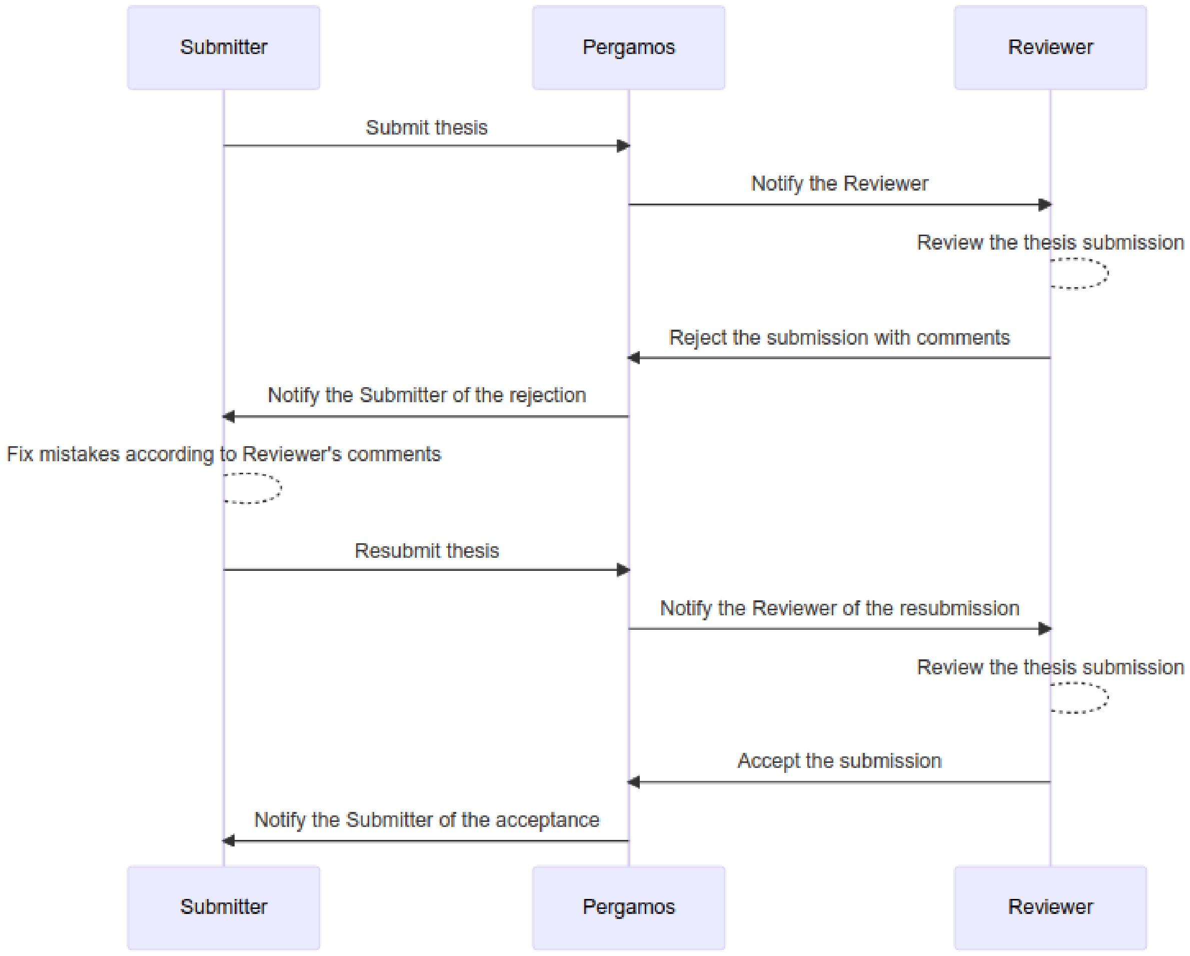

Submitter
Pergamos
Reviewer
Submit thesis
Notify the Reviewer
Review the thesis submission
Reject the submission with comments
Notify the Submitter of the rejection
Fix mistakes according to Reviewer's comments
Resubmit thesis
Notify the Reviewer of the resubmission
Review the thesis submission
Accept the submission
Notify the Submitter of the acceptance
Submitter
Pergamos
Reviewer

Figure 2: A possible submission process to the Pergamos system

This is a prime example of a process that can be elegantly expressed in a series of atomic steps. Additionally, depending on the way the developers have modeled this process, it can prove hard to take into account all the possible states a thesis can be in. A simple implementation using a `Thesis` object with a field called `state` can be in several states depending on the step in the process described before (Figure 3).

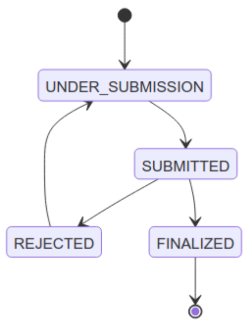
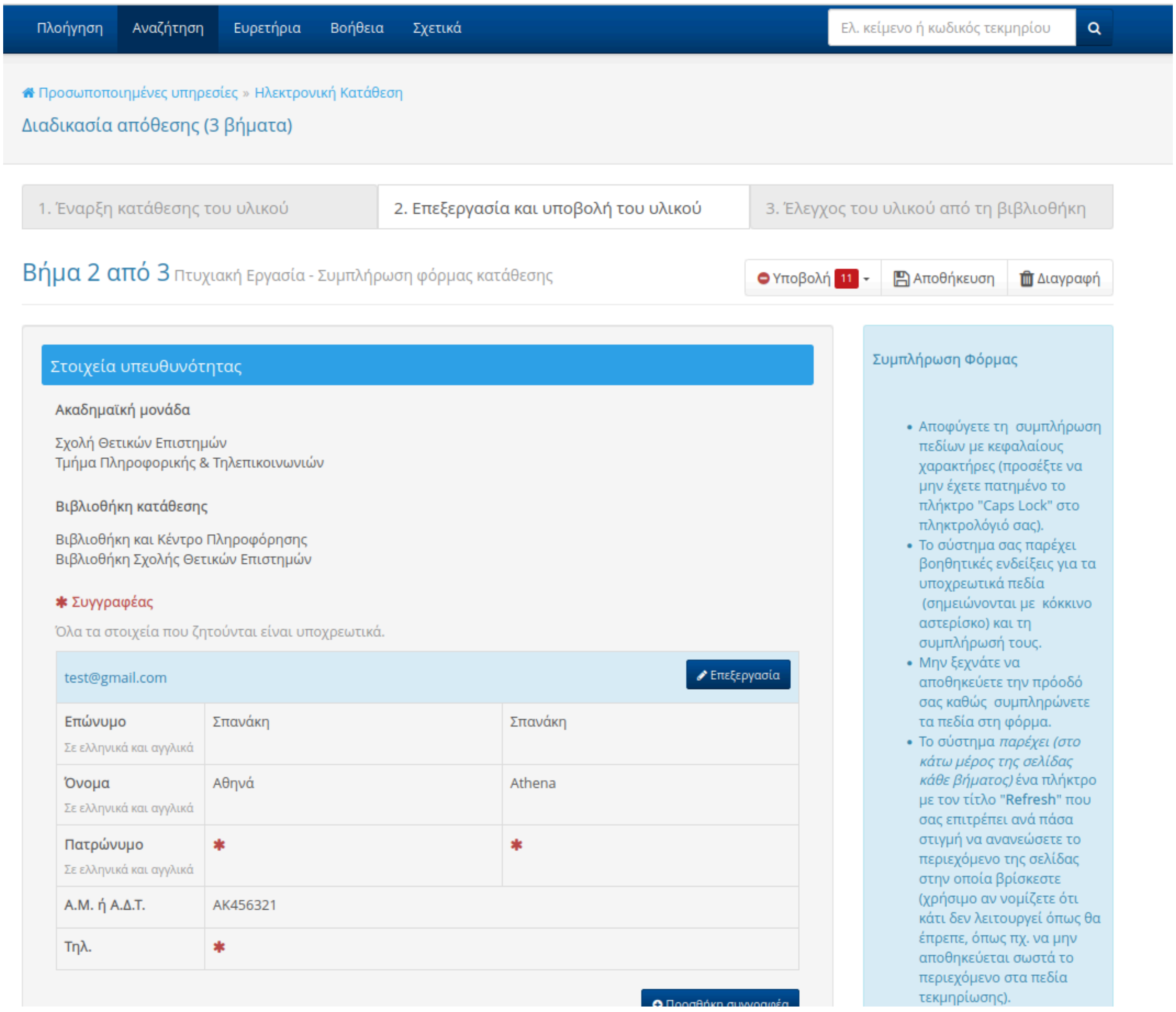


Figure 3: The state diagram of a thesis in the Pergamos system

The object would start in the `UNDER_SUBMISSION` state until the submitter submitted it for review when it would transition to a `SUBMITTED` state. Next, the reviewer would review it and either reject it transitioning it back to the `UNDER_SUBMISSION` state or approve it and transition it to the final `FINALIZED` state. This would mean that every query involving `Thesis` objects would need to take the `state` field into account in order to avoid including unwanted objects in the result. Thus, the application developer needs to always be aware of all the possible states a `Thesis` object can be in when working on a feature involving them. Moreover, having a clear historical view of the procedure can prove useful for analytic or auditing purposes. For example, statistics about rejections and their reasons could prove useful in documenting common pitfalls in the submission process thus making it smoother for future submitters.



Προσωποποιημένες υπηρεσίες » Ηλεκτρονική Κατάθεση

Διαδικασία απόθεσης (3 βήματα)

1. Έναρξη κατάθεσης του υλικού | 2. Επεξεργασία και υποβολή του υλικού | 3. Έλεγχος του υλικού από τη βιβλιοθήκη

Βήμα 2 από 3 Πτυχιακή Εργασία - Συμπλήρωση φόρμας κατάθεσης

Στοιχεία υπευθυνότητας

Ακαδημαϊκή μονάδα

Σχολή Θετικών Επιστημών
Τμήμα Πληροφορικής & Τηλεπικοινωνιών

Βιβλιοθήκη κατάθεσης

Βιβλιοθήκη και Κέντρο Πληροφόρησης
Βιβλιοθήκη Σχολής Θετικών Επιστημών

✱ Συγγραφέας

Όλα τα στοιχεία που ζητούνται είναι υποχρεωτικά.

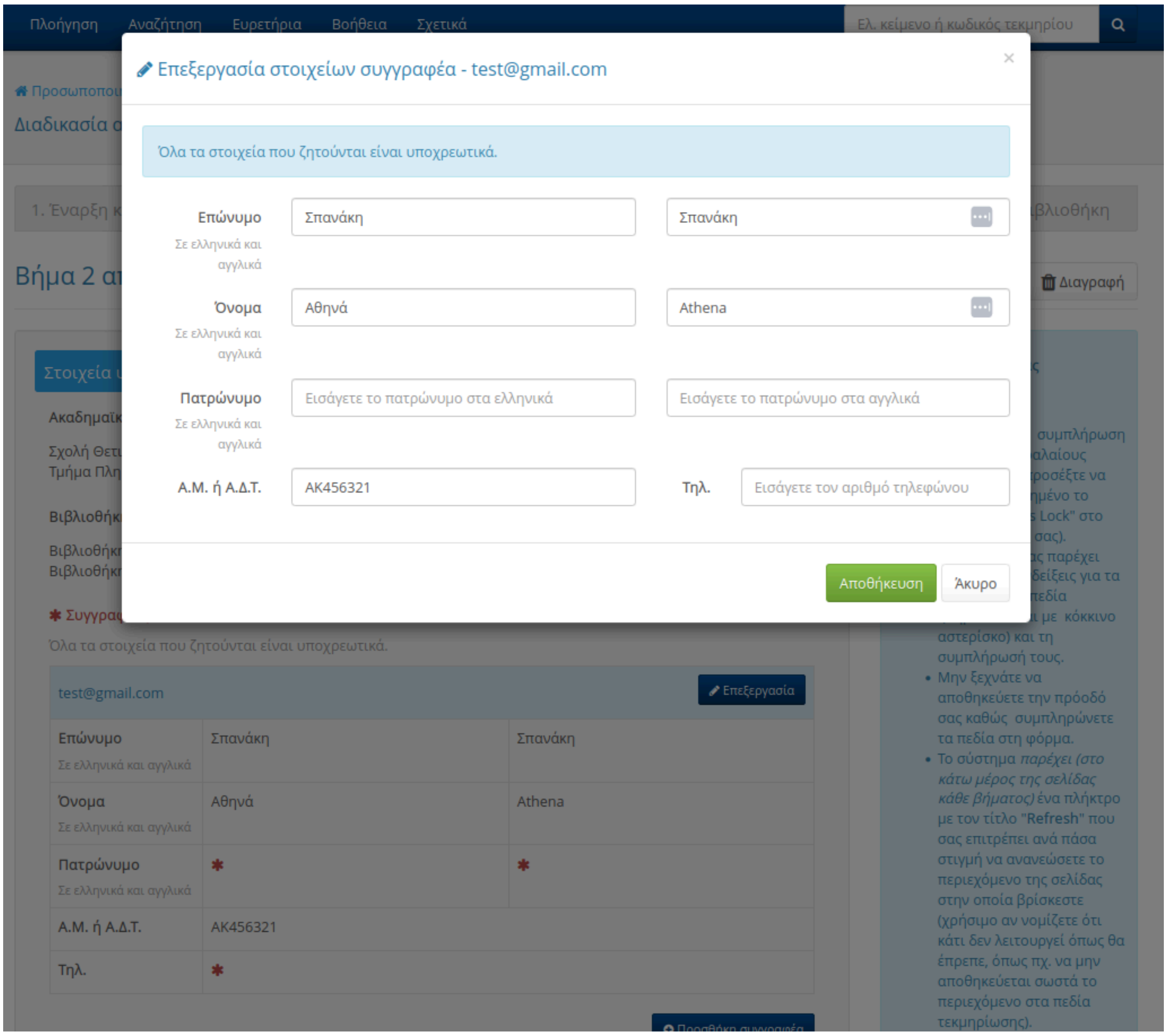
test@gmail.com

Επώνυμο Σε ελληνικά και αγγλικά	Σπανάκη	Σπανάκη
Όνομα Σε ελληνικά και αγγλικά	Αθηνά	Athena
Πατρώνυμο Σε ελληνικά και αγγλικά	✱	✱
Α.Μ. ή Α.Δ.Τ.	ΑΚ456321	
Τηλ.	✱	

Συμπλήρωση Φόρμας

- Αποφύγετε τη συμπλήρωση πεδίων με κεφαλαίους χαρακτήρες (προσέξτε να μην έχετε πατημένο το πλήκτρο "Caps Lock" στο πληκτρολόγιό σας).
- Το σύστημα σας παρέχει βοηθητικές ενδείξεις για τα υποχρεωτικά πεδία (σημειώνονται με κόκκινο αστερίσκο) και τη συμπλήρωσή τους.
- Μην ξεχνάτε να αποθηκεύετε την πρόοδό σας καθώς συμπληρώνετε τα πεδία στη φόρμα.
- Το σύστημα *παρέχει (στο κάτω μέρος της σελίδας κάθε βήματος)* ένα πλήκτρο με τον τίτλο "**Refresh**" που σας επιτρέπει ανά πάσα στιγμή να ανανεώσετε το περιεχόμενο της σελίδας στην οποία βρίσκεστε (χρήσιμο αν νομίζετε ότι κάτι δεν λειτουργεί όπως θα έπρεπε, όπως πχ. να μην αποθηκεύεται σωστά το περιεχόμενο στα πεδία τεκμηρίωσης).

Figure 4: The second step of the Pergamos thesis submission interface



Επεξεργασία στοιχείων συγγραφέα - test@gmail.com

Όλα τα στοιχεία που ζητούνται είναι υποχρεωτικά.

Επώνυμο Σε ελληνικά και αγγλικά	Σπανάκη	Σπανάκη
Όνομα Σε ελληνικά και αγγλικά	Αθηνά	Athena
Πατρώνυμο Σε ελληνικά και αγγλικά	Εισάγετε το πατρώνυμο στα ελληνικά	Εισάγετε το πατρώνυμο στα αγγλικά
Α.Μ. ή Α.Δ.Τ.	ΑΚ456321	Τηλ. Εισάγετε τον αριθμό τηλεφώνου

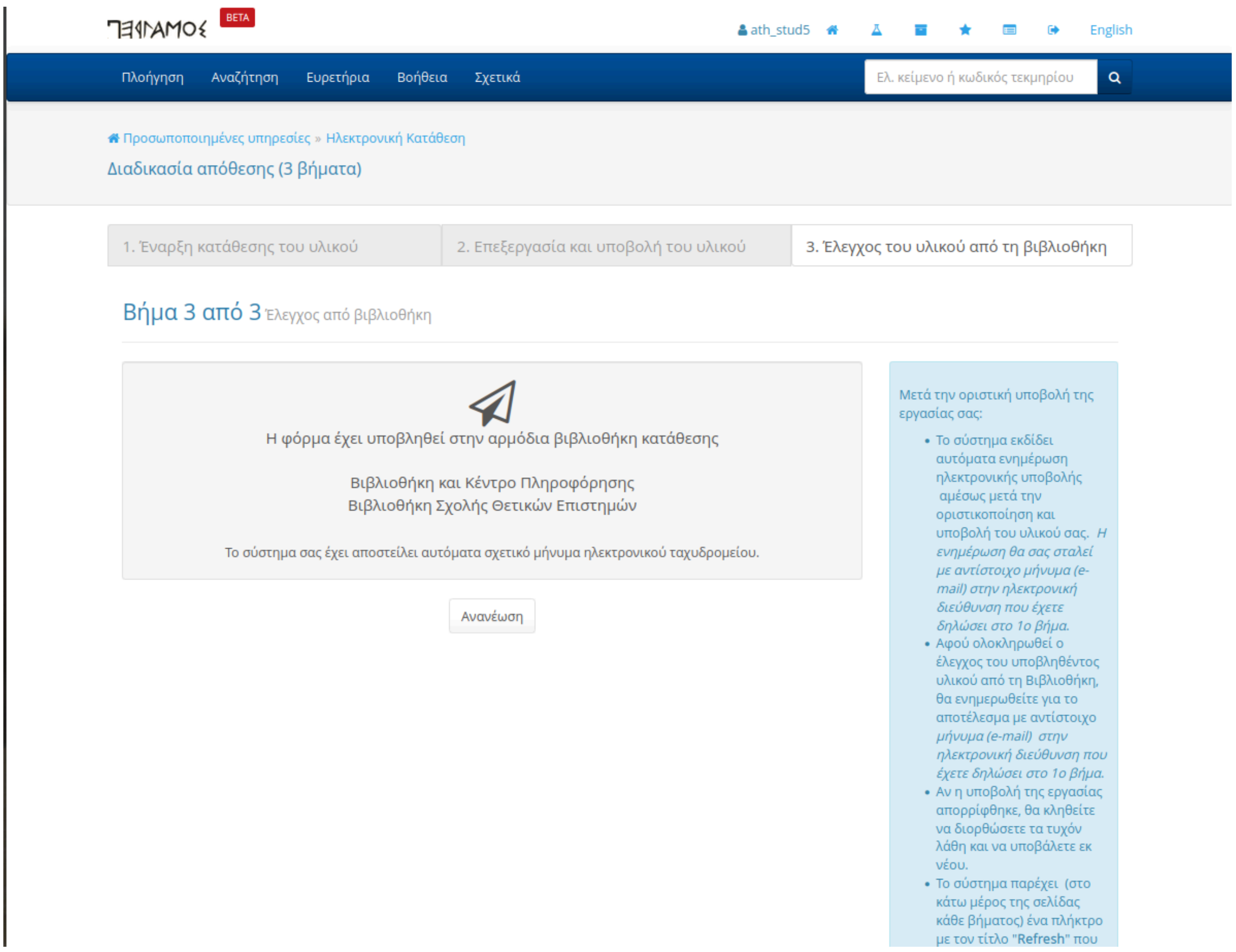
Όλα τα στοιχεία που ζητούνται είναι υποχρεωτικά.

test@gmail.com

Επώνυμο Σε ελληνικά και αγγλικά	Σπανάκη	Σπανάκη
Όνομα Σε ελληνικά και αγγλικά	Αθηνά	Athena
Πατρώνυμο Σε ελληνικά και αγγλικά	*	*
Α.Μ. ή Α.Δ.Τ.	ΑΚ456321	
Τηλ.	*	

- Μην ξεχνάτε να αποθηκεύετε την πρόοδό σας καθώς συμπληρώνετε τα πεδία στη φόρμα.
- Το σύστημα *παρέχει (στο κάτω μέρος της σελίδας κάθε βήματος)* ένα πλήκτρο με τον τίτλο "**Refresh**" που σας επιτρέπει ανά πάσα στιγμή να ανανεώσετε το περιεχόμενο της σελίδας στην οποία βρίσκεστε (χρήσιμο αν νομίζετε ότι κάτι δεν λειτουργεί όπως θα έπρεπε, όπως πχ. να μην αποθηκεύεται σωστά το περιεχόμενο στα πεδία τεκμηρίωσης).

Figure 5: Editing fields before submitting the thesis



Προσωποποιημένες υπηρεσίες » Ηλεκτρονική Κατάθεση

Διαδικασία απόθεσης (3 βήματα)

1. Έναρξη κατάθεσης του υλικού | 2. Επεξεργασία και υποβολή του υλικού | 3. Έλεγχος του υλικού από τη βιβλιοθήκη

Βήμα 3 από 3 Έλεγχος από βιβλιοθήκη

Η φόρμα έχει υποβληθεί στην αρμόδια βιβλιοθήκη κατάθεσης

Βιβλιοθήκη και Κέντρο Πληροφόρησης
Βιβλιοθήκη Σχολής Θετικών Επιστημών

Το σύστημα σας έχει αποστείλει αυτόματα σχετικό μήνυμα ηλεκτρονικού ταχυδρομείου.

Ανανέωση

Μετά την οριστική υποβολή της εργασίας σας:

- Το σύστημα εκδίδει αυτόματα ενημέρωση ηλεκτρονικής υποβολής αμέσως μετά την οριστικοποίηση και υποβολή του υλικού σας. *Η ενημέρωση θα σας σταλεί με αντίστοιχο μήνυμα (e-mail) στην ηλεκτρονική διεύθυνση που έχετε δηλώσει στο 1ο βήμα.*
- Αφού ολοκληρωθεί ο έλεγχος του υποβληθέντος υλικού από τη Βιβλιοθήκη, θα ενημερωθείτε για το αποτέλεσμα με αντίστοιχο *μήνυμα (e-mail) στην ηλεκτρονική διεύθυνση που έχετε δηλώσει στο 1ο βήμα.*
- Αν η υποβολή της εργασίας απορρίφθηκε, θα κληθείτε να διορθώσετε τα τυχόν λάθη και να υποβάλετε εκ νέου.
- Το σύστημα παρέχει (στο κάτω μέρος της σελίδας κάθε βήματος) ένα πλήκτρο με τον τίτλο "**Refresh**" που

Figure 6: Submitting the thesis

3.2. General form submission workflow

Likewise, the general form submission workflow that is ingrained in business applications poses similar challenges. Imagine an application with a deep and complex object hierarchy. For instance, consider an application that manages `Monument` objects. `Monument` objects reference multiple other objects, for example `City` objects in a field named `city` which corresponds to the city a monument resides in Listing 1.

```
// A class representing a city
class City {
  String name; // The name of the city
}

// A class representing a monument
class Monument {
  String name; // The name of the monument
  City city; // A reference to the city object this monument resides in
}
```

Listing 1: Example data model

Form submissions like this Figure 7 often need to be reviewed before being promoted to publicly accessible in order to ensure that the information they represent is factually correct. Consequently, application developers need to take this requirement into consideration when creating the domain model of the application. Moreover, they always need to be aware of the state of an object before operating on it. Thus, the problems outlined in the Pergamos example also apply to the general case of a form submission workflow.

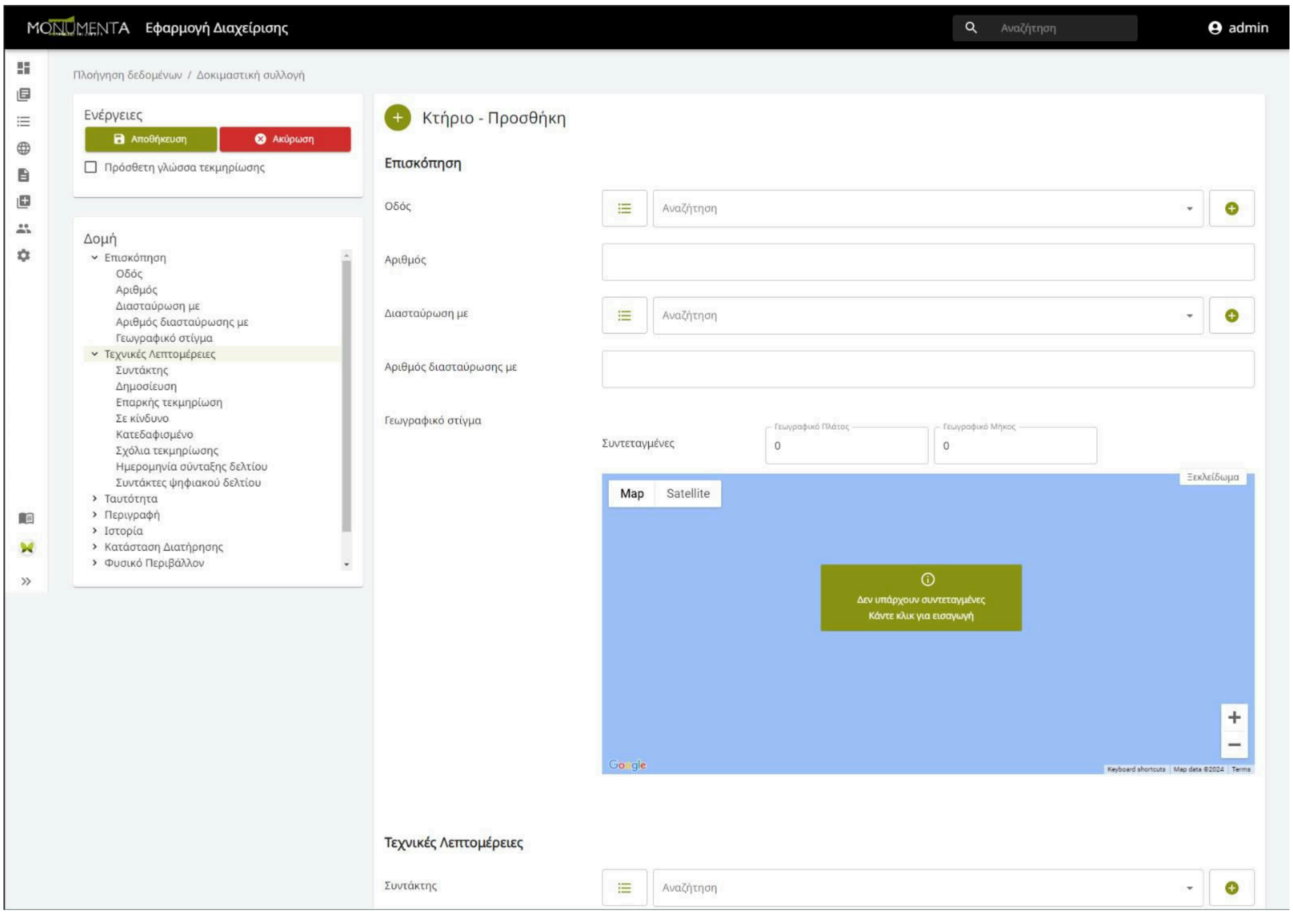


Figure 7: A monument object creation form

3.3. Proposing vcs concepts for business workflows

Both the specific thesis submission workflow and the broader form submission workflows encountered in business applications exhibit challenges related to managing complex workflows. Introducing version control capabilities to the software systems involved could significantly mitigate these challenges. By implementing version control features, developers can establish a clear historical view of the submission process, track changes, and manage the various states that objects undergo during the workflow. For instance, in the case of thesis submissions, version control mechanisms could facilitate tracking revisions, documenting reviewer comments, and streamlining the acceptance process. Similarly, in the context of general form submissions, version control could enhance the management of complex object hierarchies, ensure accurate data representation, and simplify the review and promotion process. A prime example of form heavy and business object rich applications are applications that are employed by public services. Such applications oftentimes provide forms and workflows that allow users to create or update documents related to real world bureaucratic processes. Thus, keeping a trail of every change any person caused to such business objects could prove useful for transparency, bookkeeping and recovery use cases. Ultimately, integrating version control concepts into business applications offers a promising avenue to enhance workflow efficiency, facilitate auditing and analytics, and alleviate the burden on developers by providing a structured approach to managing complex processes and data states.

4. VCS OVERVIEW

The history of version control systems starts with the *Source Code Control System* developed by Marc J. Rochkind at Bell Labs in 1973 [10]. It was primitive, allowed only one user to make changes at a time and required changes to the source code of programs in order to use. However, it proved that the idea of version control systems was sound and sparked the development of what today is considered an industry. From there, numerous other projects emerged with varying degrees of success, and number of features. The real breakthrough however came with the advent of distributed version control systems [11]. Up until their creation, VCS required a centralized server that coordinated access to the repository. On the other hand, distributed VCS allowed developers to work on and share their changes with each other without the requirement for a centralized component. Thus, teams with a large number of code contributors could work more efficiently. One of these teams was that of the *Linux* kernel. Its creator Linus Torvalds, after a spat with the proprietary *BitKeeper* VCS in 2005 went on to create what is now the most widely used VCS, *Git* [12].

4.1. Git basics

Modern VCS are designed with a set of primitives that are composed to provide version control capabilities. The implementation of this thesis uses the nomenclature used by modern VCS, with preference to that used by *Git*. The root directory which contains all the files that will be tracked by the VCS is called a **repository**. The atomic unit of change processed by a VCS is called a **commit**. It consists of the set of changes to the files **tracked** by the VCS along with a **commit message**. That message explains the motivation behind the change, the person who made the change called the **committer** and the time at which the change was made. Commits refer to the commit that came directly before them, forming a chain and allowing users to traverse the chain and inspect the state of the repository at any point in time. Commits can belong to a **branch**, a parallel chain of commits to the main chain that can be committed to concurrently with the main chain and can be merged back to it after its lifetime. Branches are commonly used to work on single features in order to isolate the work from other unrelated changes and to prevent breakages and impediments to the main branch which is the basis of other branches. Finally many VCS allow users to **tag** commits with a user defined tag id which allows commits to represent special points in the life of a repository like software releases.

Let's take a look at an example. First, Alice creates the repository for her project and commits the initial files. Then, she commits the implementation of a feature called foo in a separate branch. Concurrently, another developer called Bob begins work on a feature called bar in another separate branch Alice finishes with the work on the foo feature and merges the foo branch to the main branch. Additionally, when Bob finishes the work on the bar feature he also merges to the main branch. Finally, in a new commit, Alice adapts foo to use bar and then tags that commit as the first version of the software. Thus, Alice and Bob have managed to coordinate work on the same software project. More importantly however, they have a complete historical record of every change that has led to the current state of the repository.

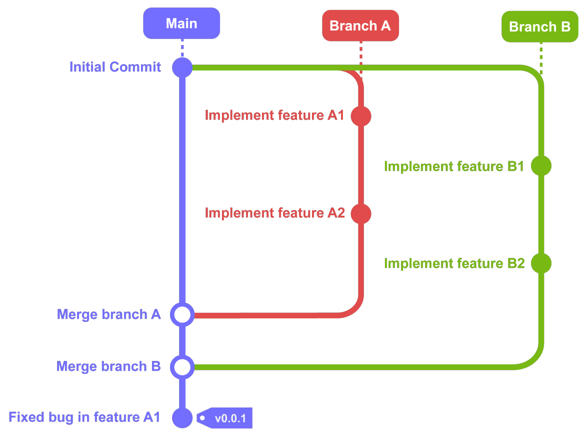


Figure 8: An example of a repository with its commit history

5. DOLAR OVERVIEW

The implementation of the version controlled object database uses the DOLAR project in order to provide its data store agnostic capabilities. DOLAR provides its own set of primitives to abstract and provide a unified API over data stores and the objects stored in them. The first and foremost primitive is the `Store` which represents a repository of objects. The `Store` is configured with a backing storage system which it uses to read and write objects to, for example the MySQL database server. Objects are described by a `Prototype` that defines the data the object consists of as perceived from the application side (Listing 2).

```
type Person {
  name String
}

type Driver inherits Person {
  yearsDriving Number
}

type Manufacturer {
  name String
}

type Car {
  modelName String
  manufacturer Manufacturer
  drivers List(Driver)
}
```

Listing 2: Example DOLAR prototype

An instance of an object is called a `VirtualObject`. `VirtualObject`s are associated with a `Prototype` and are created by retrieving the object from the backing data store and mapping it to the `VirtualObject` using the `Prototype`. Additionally, `VirtualObject`s can be created by application code in order to store new objects in the backing data store (Listing 3).

```
Car createCar(Manufacturer m, String modelName) {

  // Instantiate a new instance of the desired Prototype that resides in memory.
  // The returned instance is a VirtualObject
  var car = DOLAR.newObject(Car.class);

  // Set the relevant fields
  car.setModelName(modelName);
  car.setManufacturer(m);

  // Return the VirtualObject to the caller. Do note that until the save()
  // method is called on the VirtualObject it is not persisted in the backing
  // data store.
  return car;
}

void doStuffWithCar() {
  // Get an existing object using its unique identifier
  var manufacturer = DOLAR.getObjectByID("manufacturerA")

  var car = createCar(manufacturer, "MK-1")

  // Save the created object to the configured backing data store.
  // Here the configuration of the datastore is removed for brevity.
  car.save();
}
```

Listing 3: Example DOLAR Java SDK usage

Objects can be retrieved by their unique identifier from the `Store` and can reference other objects in the `Store`. Furthermore, objects can be inserted in batches that will leverage the transaction capabilities of the backing storage system, where available. Moreover, the `Store` can be configured with a backing index system that can be leveraged to query the objects in the `Store` with more sophisticated criteria. Finally, DOLAR provides the application with hooks that it can use to react to events in the lifetime of a `VirtualObject`.

It is evident that DOLAR plays a significant role in our implementation as it enables us to offer a store-agnostic version control primitives atop of any underlying DOLAR store. Its most important element however is its batch processing capabilities which the implementation relies on and extends. The DOLAR batch object processing system allows data manipulation of multiple objects at once Figure 9. This includes inserting new objects to the data store, updating objects that already exist and deleting them.

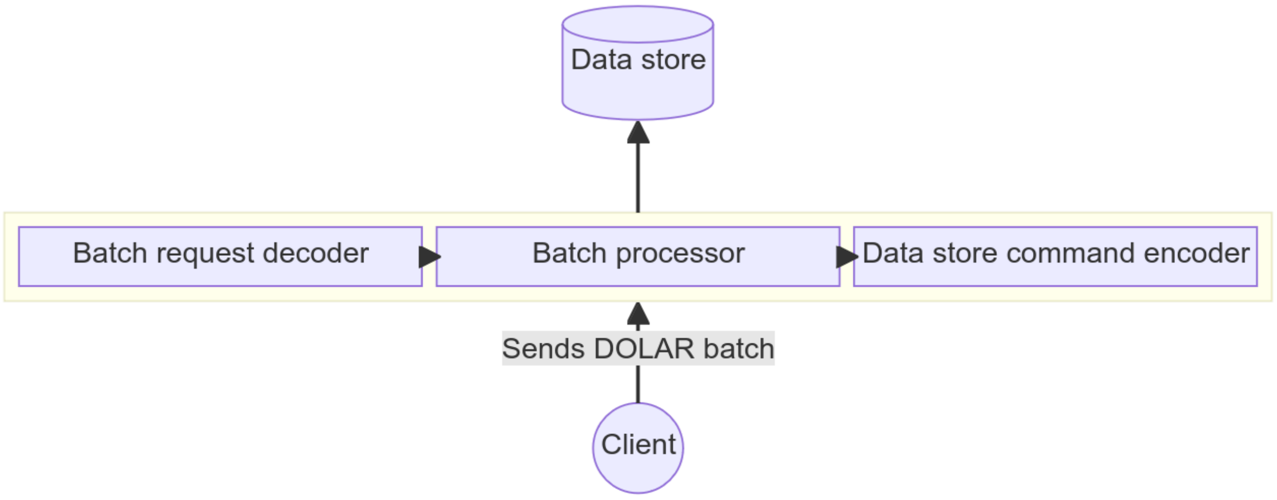


Figure 9: The DOLAR batch object processing system

There are multiple ways to construct a DOLAR batch operation request. Here its JSON document format will be used as JSON is the defacto contemporary industry standard for data exchange in web applications. DOLAR batches support a number of operations with the most important ones being “create”, “update” and “delete”. The user can provide the data to each operation by creating an object named after the operation. This operation object in turn can feature any number of lists named after the `Prototype` of the objects that the operation will act upon. Each list consists of any number of objects that contain the data for the `VirtualObject` the operation will act on. Objects in the batch can reference other objects in the batch by using a special referral ID. That ID is created using the following syntax `@<OPERATION_NAME>.<PROTOTYPE>.<INDEX_IN_LIST>`. Consider the following example. The data model we will use is described using pseudo-java Listing 4.

```
class City {
    String name;
}

class Monument {
    String name;
    City city;
}

class FeaturedMonument {
    Monument monument;
}
```

Listing 4: Example data model

The data model describes a simple hierarchy of objects that could be used by a monument cataloging application. Lets imagine that a `FeaturedMonument` with an ID set to `featured_monument_greece` has already been inserted into the system. Next, the DOLAR batch facility will be supplied using a JSON document Listing 5.

```
{
    "CREATE": {
        "Monument": [
            {
                "name": "Temple of Poseidon",
                "city": "@CREATE.City.0"
            }
        ],
        "City": [
            {
                "name": "Sounio"
            }
        ]
    },
    "UPDATE": {
        "FeaturedMonument": [
            {
                "id": "featured_monument_greece",
                "monument": "@CREATE.Monument.0"
            }
        ]
    }
}
```

Listing 5: The DOLAR batch input document

After the batch system completes the request the backing data store will store two new objects and the updated third object. The newly created `City` object will reference the newly created `Monument` object. Likewise, the already existing `FeaturedMonument` object will reference the newly created `Monument` object. More importantly, all this will happen in a single database transaction, if the backing data store provides such capability. Ultimately, the batch processing feature of DOLAR is a powerful mechanism that the implementation of the thesis uses to transparently provide its version control features.

6. DESIGN & IMPLEMENTATION

The implementation borrows some terminology used by *git* internally for its primitives. Firstly, a DOLAR `Store` represents a repository. Next, the implementation uses some `Prototype`s to describe and store its primitives.

- `Commit` Listing 6

`Commit` objects represent an atomic snapshot of the state of the objects in the lifetime of the repository. In order to represent the evolution of that state, `Commit` objects reference the `Commit` objects that they derive their previous state from using a field called `parents`. Objects created in the repository belong to a `Commit`.

```
class Commit {
  String id;
  List<Commit> parents;
}
```

Listing 6: The data model of the `Ref` object in pseudo-java

- `Ref` Listing 7

`Ref` objects represent a named reference to a particular state of the repository. For instance, the `main` `Ref` represents the latest state of the repository and is created automatically when the repository is initialized. `Ref` objects reference the latest `Commit` in the particular state they refer to using a field called `head`. The latest state of an object (as of a specific `Ref`) belongs to a `Ref`.

```
class Ref {
  String id;
  Commit head;
}
```

Listing 7: The data model of the `Ref` object in pseudo-java

The implementation strives to be independent from the data stored in existing systems such that such systems can work with the implementation without needing a complete rewrite, so it does not introduce extra fields to the objects stored in the repository. Instead, it uses the identifiers of the objects in question to augment their meaning and provide the version control capabilities. For instance, the identifier of an object that is part of a `Commit` is `<object ID>:commit:<Commit ID>`. Likewise, the identifier of an object that is latest as of a particular `Ref` is `<object ID>:ref:<Ref ID>`. Consider the example state of a repository in Figure 10. Objects implicitly refer to the VCS object they belong to (here marked by the dotted line). Conversely, VCS objects have an explicit reference to each other.

- `object1` and `object2` were created first and their snapshots `object1:commit:1` and `object2:commit:1` belong to `commit:1`. At first the main `Ref` `ref:main` was created and referenced `commit:1` while `object1` and `object2` implicitly belong to `ref:main`.
- `object3` was created along with its snapshot `object3:commit:2` and a new `Commit` object `commit:2`. The `ref:main` object was updated to reference the newly created `commit:2`.

The final state of the repository contains every state the repository has gone through.

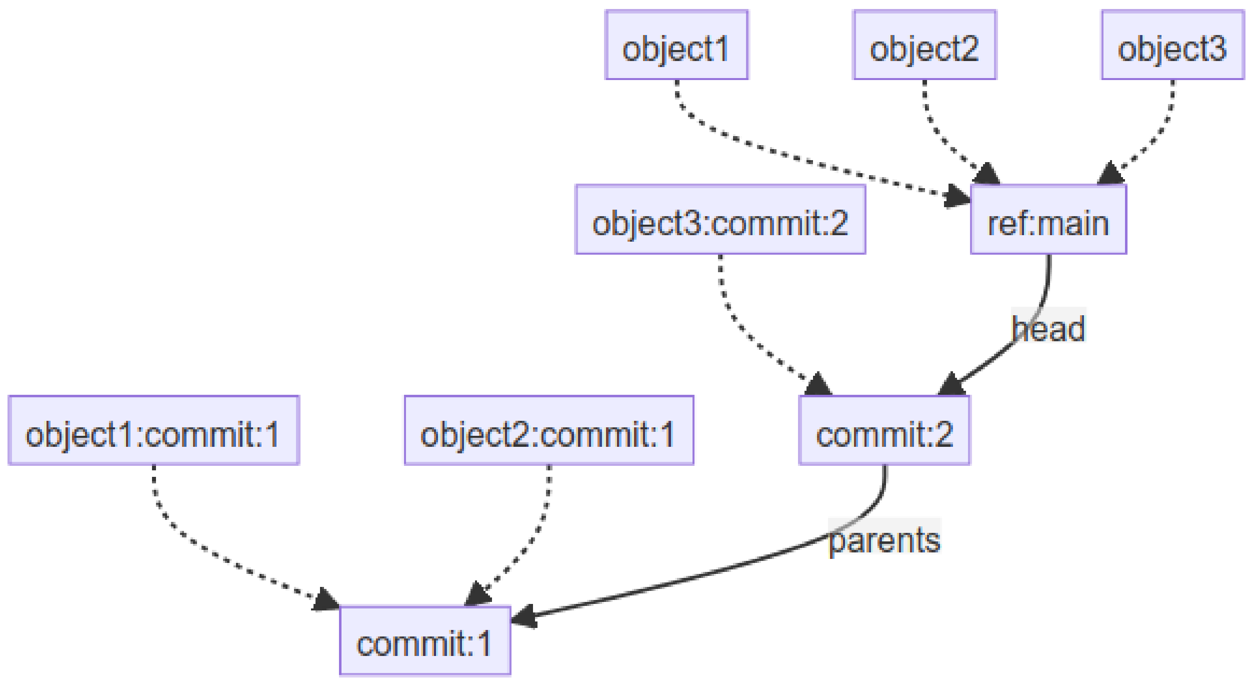


Figure 10: The representation of the history of the main branch

On the other hand, branches to the history of the repository are nothing more than a `Ref` other than `ref:main`. In Figure 11 a branch named `foo`, represented by the `Ref` `ref:foo`, was created at some point along the history of the repository. Objects at the latest state of the repository as of the `ref:foo` branch are represented by simply concatenating the `ref:foo` suffix to their identifier.

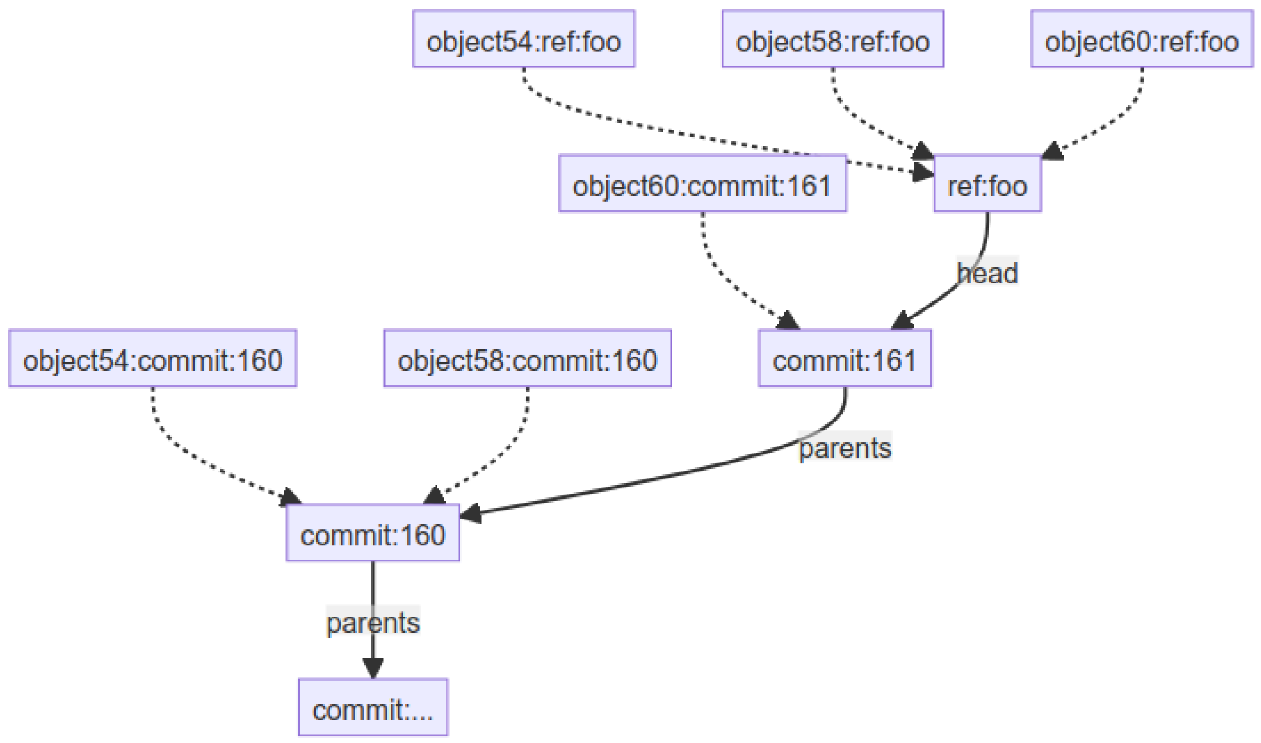


Figure 11: The representation of the history of an auxilliary branch

Having described the primitives used by the implementation and the way they work the operations that it provides can be introduced. These are

- **Committing** one or more new objects or updates to existing objects to a branch of the repository.
- **Merging** a branch onto another one.

The **commit** VCS operation provided by the implementation is implemented by extending the DOLAR batch processing system with a new batch processor. Clients must submit an input DOLAR batch request to the **commit** operation which in turn will transform it before it is processed by the rest of the DOLAR batch system. Initially the batch processor transforms the batch request by adding the creation of a `Commit` object along with the update of the relevant `Ref` object to reference the newly created `Commit`. Next, for every object that is updated or created in the batch the VCS batch processor adds the creation of a second identical object with its identifier set to reference the commit that is created to the batch request. Similarly, the identifier of the created or update object is set to reference the `Ref` object that this batch is a part of. A notable exception is the case of the `main` `Ref` for which the identifier of the object is left as is so as to facilitate the data independence guarantees of the implementation. For example consider the input DOLAR batch for the **commit** operation on the `main` branch (Listing 8).

```
{
    "CREATE": {
        "Monument": [
            {
                "name": "Temple of Poseidon",
                "city": "@CREATE.City.0"
            }
        ],
        "City": [
            {
                "name": "Sounio"
            }
        ]
    },
    "UPDATE": {
        "FeaturedMonument": [
            {
                "id": "featured_monument_greece",
                "monument": "@CREATE.Monument.0"
            }
        ]
    }
}
```

Listing 8: An example input DOLAR batch

When this batch is submitted the VCS batch processor transforms it to batch in Listing 9.

```
{
    "CREATE": {
        "Commit": [
            {
                "id": "500",
                "parents": "499"
            }
        ],
        "Monument": [
            {
                "id": "temple_of_poseidon",
                "name": "Temple of Poseidon",
                "city": "@CREATE.City.0"
            }
            {
                "id": "temple_of_poseidon:commit:500",
                "name": "Temple of Poseidon",
                "city": "@CREATE.City.0"
            }
        ],
        "City": [
            {
                "id": "sounio",
                "name": "Sounio"
            },
            {
                "id": "sounio:commit:500",
                "name": "Sounio"
            }
        ],
        "FeaturedMonument": [
            {
                "id": "featured_monument_greece:commit:500",
                "monument": "@CREATE.Monument.0"
            }
        ]
    },
    "UPDATE": {
        "Ref": [
            {
                "id": "main",
                "head": "@CREATE.Commit.0"
            }
        ],
        "FeaturedMonument": [
            {
                "id": "featured_monument_greece",
                "monument": "@CREATE.Monument.0"
            }
        ]
    }
}
```

Listing 9: The transformation applied to the example input batch

Regarding the **merge** operation, clients can request it by specifying the **source** and **target** branches. The VCS system accomplishes the **merge** operation by creating and submitting a DOLAR batch operation. The DOLAR batch contains the creation of a `Commit` object that references the latest `Commit` objects of both branches. Next, the `Ref` of the **target** branch is updated to reference the newly created `Commit` object. Finally, every object that was created or updated in the **source** branch is inserted to the DOLAR batch. Thus, the state of the repository after the **merge** operation is the union of the states of both the **source** and **target** branches with the intersection keeping the objects of the **source** branch Figure 12.

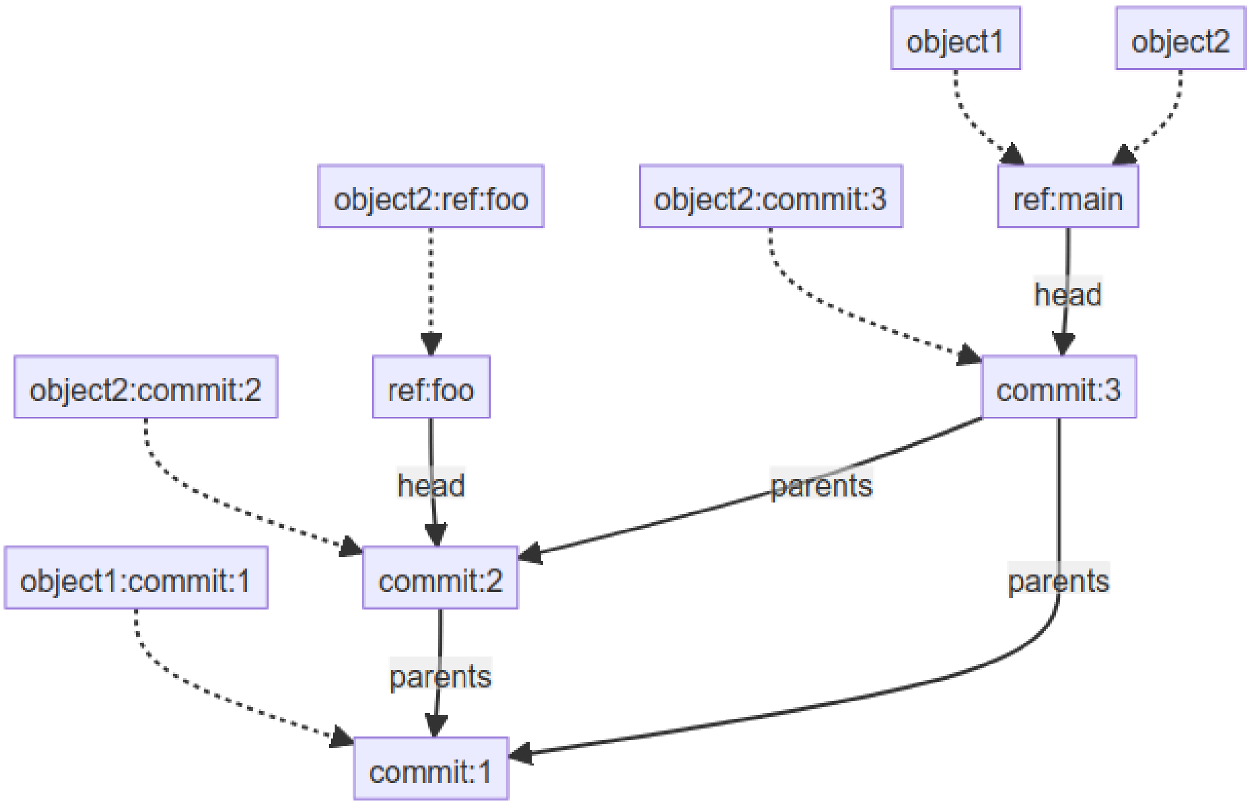


Figure 12: The representation of the history after a merge

7. EVALUATION

The implementation provides application developers with a DOLAR system that enables the use of version control primitives and operations to model or extend their systems with. Application developers could introduce DOLAR along with the VCS extensions presented here to the write code paths of their applications thus benefiting from the history tracking. The evaluation of the implementation however, is accomplished by examining a possible rework of the Pergamos thesis submission workflow with the primitives and operations provided by it. First, the process begins by the submitter submitting their draft to Pergamos. In turn, Pergamos creates a branch named after the thesis and commits the thesis draft. This branch will represent the thesis in an unpublished and under review state. Next, Pergamos notifies the reviewer of the thesis submission. The reviewer will review the submission and accept or reject it with comments. Upon rejection, Pergamos commits the comments made by the reviewer and notifies the submitter. Then, the submitter fixes their submission and resubmits to Pergamos which in turn commits the new submission. Once the submitter has fixed all the comments made by the reviewer and the reviewer has accepted the submission, Pergamos merges the submission branch to the main branch Figure 13.

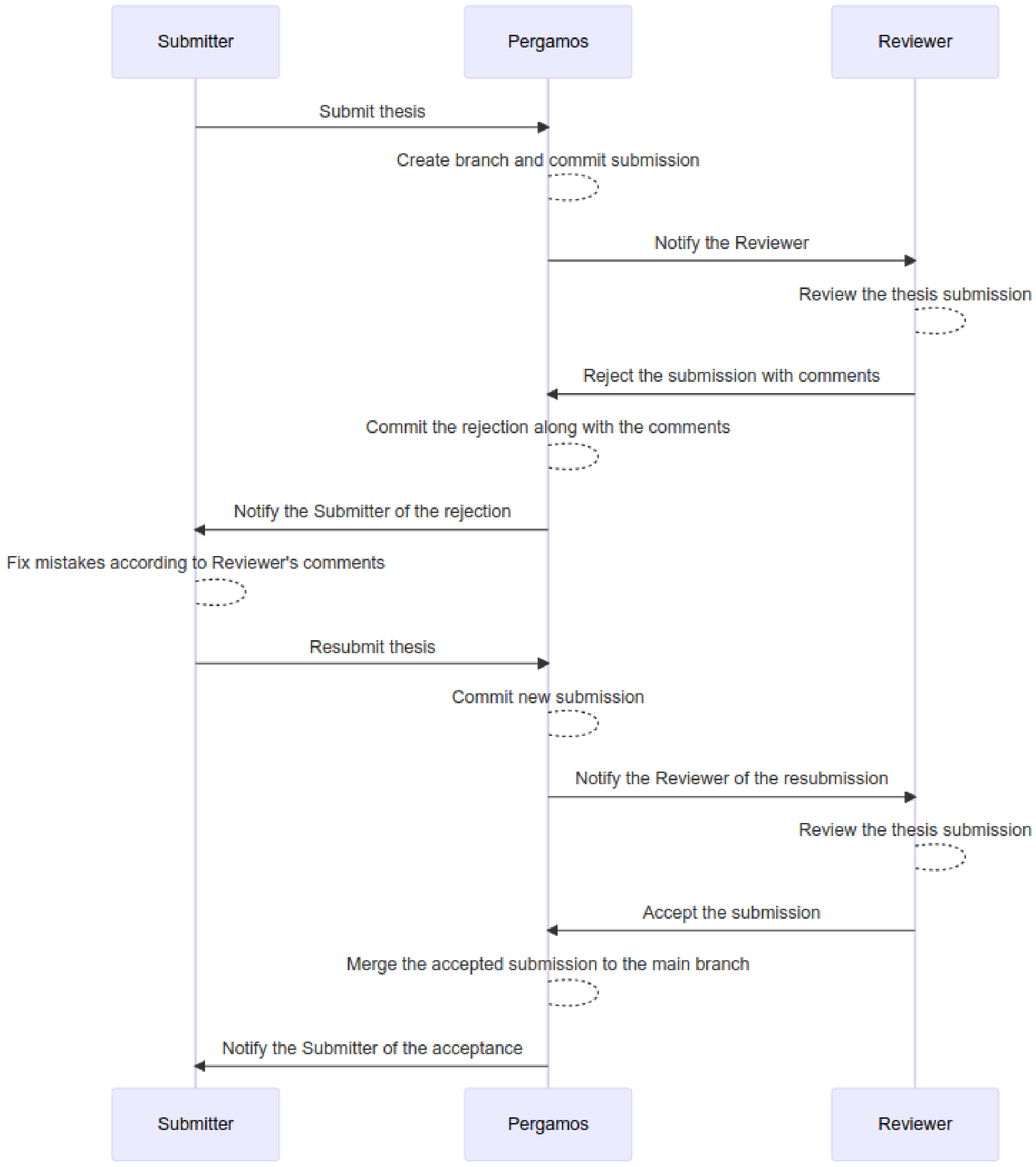

Submitter
Pergamos
Reviewer
Submit thesis
Create branch and commit submission
Notify the Reviewer
Review the thesis submission
Reject the submission with comments
Commit the rejection along with the comments
Notify the Submitter of the rejection
Fix mistakes according to Reviewer's comments
Resubmit thesis
Commit new submission
Notify the Reviewer of the resubmission
Review the thesis submission
Accept the submission
Merge the accepted submission to the main branch
Notify the Submitter of the acceptance
Submitter
Pergamos
Reviewer

Figure 13: The Pergamos thesis submission process using version control features

Thus, the main branch only contains published theses. This reduces the number of states a thesis can be in while on the main branch. Recall the possible states a thesis can take in the simple implementation discussed earlier Figure 3. Compared to that, the current revised implementation can forgo the `state` field entirely. Instead, the thesis is in the - now implicit - finalized state if and only if it is present in the main branch. Conversely, theses on other branches and by definition in the `UNDER_SUBMISSION` state. Consequently, the cognitive load imposed on developers was reduced. More importantly however, the data model and its state space was greatly simplified Figure 14.

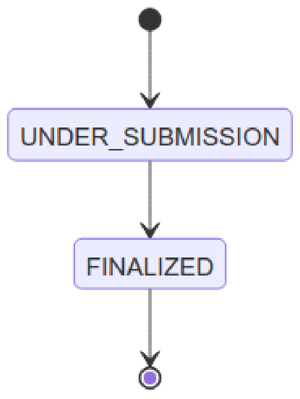


Figure 14: The state diagram of a Pergamos thesis after the domain simplification

This results in a more robust solution, since edge case states don't need to be included in every user facing query, and a more scalable solution since developers can implement new workflows using a simplified data model. Moreover, each state a thesis has been in across its lifetime has been recorded it the system and can be resurfaced easily.

Similarly, the more general form submission workflow discussed in Chapter 3 could benefit from the introduction of a VCS workflow. Like the Pergamos case, objects created or updated as part of a form submission are persisted in a branch. When the review process completes the submission branch, now at the desired state, is merged to the main branch populating it with the objects in a reviewed and correct state.

This improves the general form submission workflow in a number of ways. First, objects belonging to submissions pending review are separated from the working set of objects thus simplifying the work of the application developers. Secondly, the versions the object goes through in the life cycle of the application are preserved and can be inspected on demand. Finally, the code that needs to be written by application developers is simpler thus allowing for more and more complex business workflows to be implemented in a robust way.

However, the current design has some limitations and the implementation makes some trade offs in order to meet its stated goals. First, objects are essentially copied on update. This means that frequently updated objects will end up increasing the total storage space used significantly. Next, in order to support merging, some metadata needs to be tracked for each branch. This results in even more storage space use as well as slower operations since each operation needs to also write the metadata to the backing data store. Finally, data independence is achieved by repurposing the identifier of an object instead of adding additional fields or relations. Consequently, searching using version control information would be hard while unrelated searches would result in unwanted version control objects. Thus, the current implementation elects to not index version control objects at all in order to keep the search system backwards compatible.

8. CONCLUSION

In conclusion, the implementation equips application developers with the tools necessary to enrich their applications with version control capabilities. It should be noted that our work does so with full backwards compatibility and data independence from the existing system and without requiring applications to adapt their data models to the requirements of the implementation. However, the implementation trades off its flexibility and backwards compatibility with increased storage overhead because of the extra copies of objects it stores.

The current implementation leaves much room for improvement in the future. First, no work is done on conflict detection and resolution. Implementing such a feature, while maintaining the backwards compatibility guarantees of the current implementation, will pose interesting challenges as changes to the underlying data model will be needed. Secondly, various other features could be built upon the core implemented in this thesis. For example, a UI showing the history of a branch along with the changes introduced by each commit. Moreover, the operations provided by the implementation could be enhanced providing more VCS operations like rebasing and cherry picking. Finally, an indexing system that allows the dynamic search of the history of a repository would allow even better history querying capabilities.

BIBLIOGRAPHY

[1] C. Birchall, "Re-Engineering Legacy Software," Manning, 2016.

[2] K. Saidis, Y. Smaragdakis, and A. Delis, "DOLAR: virtualizing heterogeneous information spaces to support their expansion," *Software: Practice and Experience*, vol. 41, no. 11, pp. 1349–1383, 2011.

[3] "CoreObject." Accessed: Mar. 18, 2024. [Online]. Available: http://coreobject.org/

[4] "XTDB." Accessed: Mar. 18, 2024. [Online]. Available: https://www.xtdb.com/

[5] "temporal_tables." Accessed: Mar. 18, 2024. [Online]. Available: https://github.com/arkhipov/temporal_tables

[6] "How S3 Versioning works." Accessed: Mar. 18, 2024. [Online]. Available: https://docs.aws.amazon.com/AmazonS3/latest/userguide/versioning-workflows.html

[7] "Object versioning." Accessed: Mar. 18, 2024. [Online]. Available: https://cloud.google.com/storage/docs/object-versioning

[8] "postgresql-tableversion." Accessed: Mar. 18, 2024. [Online]. Available: https://github.com/linz/postgresql-tableversion

[9] "Pergamos." Accessed: Mar. 18, 2024. [Online]. Available: https://pergamos.lib.uoa.gr/uoa/dl/frontend/index.html

[10] M. J. Rochkind, "The source code control system," *IEEE Transactions on Software Engineering*, no. 4, pp. 364–370, 1975.

[11] R. Majumdar, R. Jain, S. Barthwal, and C. Choudhary, "Source code management using version control system," in *2017 6th International Conference on Reliability, Infocom Technologies and Optimization (Trends and Future Directions) (ICRITO)*, 2017, pp. 278–281. doi: 10.1109/ICRITO.2017.8342438.

[12] L. Torvalds, J. Hamano, and others, "Git," *Software Freedom Conservancy*, p. 20–21, 2005.