Développement d'historiques de commandes avancés pour améliorer le processus d'édition numérique

Philippe Schmid

Université de Lille

CRIStAL UMR 9189 — ED SPI MADIS 631

Unité de recherche
Centre Inria Lille Nord Europe

Thèse présentée par

Philippe Schmid

Soutenue le **9 juin 2023**

En vue de l'obtention du grade de docteur de l'Université de Lille

Discipline **Informatique**
Spécialité **Interaction Humain – Machine**

Développement d'historiques de commandes avancés pour améliorer le processus d'édition numérique

Développement d'un modèle et d'une architecture d'historique de commandes multi-fonction

Thèse dirigée par	**Stéphane Huot**	directeur	
	Mathieu Nancel	co-encadrant	
Composition du jury			
Rapporteurs	**Arnaud Blouin**	MCF HDR à l'INSA Rennes	
	Stéphane Conversy	professeur à l'ENAC-LII, Université de Toulouse	
Examinateurs	**Laurence Nigay**	professeure à l'Université de Grenoble Alpes	
	Romain Rouvoy	professeur à l'Université de Lille	*président du jury*
Directeurs de thèse	**Stéphane Huot**	directeur de recherche à l'Inria Lille Nord Europe	
	Mathieu Nancel	chargé de recherche à l'Inria Lille Nord Europe	

Colophon

Mémoire de thèse intitulé «Développement d'historiques de commandes avancés pour améliorer le processus d'édition numérique ; Développement d'un modèle et d'une architecture d'historique de commandes multi-fonction», écrit par Philippe Schmid, composé au moyen du système de préparation de document LaTeX(distribution complète TexLive 2023), et de la classe yathesis dédiée aux thèses préparées en France. Achevé le 9 septembre 2023.

Université de Lille

CRIStAL UMR 9189 — ED SPI MADIS 631

Unité de recherche
Centre Inria Lille Nord Europe

Thèse présentée par

Philippe Schmid

Soutenue le **9 juin 2023**

En vue de l'obtention du grade de docteur de l'Université de Lille

Discipline **Informatique**
Spécialité **Interaction Humain – Machine**

Développement d'historiques de commandes avancés pour améliorer le processus d'édition numérique

Développement d'un modèle et d'une architecture d'historique de commandes multi-fonction

Thèse dirigée par	**Stéphane Huot**	directeur	
	Mathieu Nancel	co-encadrant	
Composition du jury			
Rapporteurs	**Arnaud Blouin**	MCF HDR à l'INSA Rennes	
	Stéphane Conversy	professeur à l'ENAC-LII, Université de Toulouse	
Examinateurs	**Laurence Nigay**	professeure à l'Université de Grenoble Alpes	
	Romain Rouvoy	professeur à l'Université de Lille	*président du jury*
Directeurs de thèse	**Stéphane Huot**	directeur de recherche à l'Inria Lille Nord Europe	
	Mathieu Nancel	chargé de recherche à l'Inria Lille Nord Europe	

Université de Lille

CRIStAL UMR 9189 | ED SPI MADIS 631

Unité de recherche
Centre Inria Lille Nord Europe

Thesis defended by

Philippe Schmid

Defended on **June 9, 2023**

In order to become Doctor from Université de Lille

Academic Field **Computer Science**
Speciality **Human – Computer Interaction**

Developing advanced command histories to improve digital editing processes

A Swiss army knife of command histories model and architecture

Thesis supervised by	**Stéphane Huot**	Supervisor	
	Mathieu Nancel	Co-Monitor	
Committee members			
Referees	**Arnaud Blouin**	HDR Associate Professor at INSA Rennes	
	Stéphane Conversy	Professor at ENAC-LII, Université de Toulouse	
Examiners	**Laurence Nigay**	Professor at Université de Grenoble Alpes	
	Romain Rouvoy	Professor at Université de Lille	*Committee President*
Supervisors	**Stéphane Huot**	Senior Researcher at Inria Lille Nord Europe	
	Mathieu Nancel	Junior Researcher at Inria Lille Nord Europe	

Mots clés : ihm, historiques de commandes, structures de données, édition, exploration, collaboration

Keywords: hci, command histories, data structures, editing, exploration, collaboration

Cette thèse a été préparée dans les laboratoires suivants.

Centre Inria de l'Université de Lille

Parc scientifique de la Haute-Borne
40, avenue Halley - Bât. A - Park Plaza
59650 Villeneuve d'Ascq
France

☎ (33)(0)3 59 57 78 00
✉ contact-lille@inria.fr
Site inria.fr

Laboratoire CRIStAL - UMR 9189

Université de Lille - Campus scientifique
Bâtiment ESPRIT
Avenue Henri Poincaré
59655 Villeneuve d'Ascq
France

Site www.cristal.univ-lille.fr

ANR JCJC Causality

Integrating Temporality and Causality to the Design of Interactive Systems. ANR-18-CE33-0010-01.

Région Hauts–de–France

Ce projet a bénéficié du soutien financier de la Région Hauts–de–France

À ma famille

À mamie et Michèle

Développement d'historiques de commandes avancés pour améliorer le processus d'édition numérique

Développement d'un modèle et d'une architecture d'historique de commandes multi-fonction

Résumé

Les historiques de commandes sont omniprésents dans le monde de l'édition. Au cours du temps, plusieurs modèles d'historiques ont émergé, abordant différentes fonctionnalités comme des types d'annulations et de ré-exécution, ainsi que la gestion des conséquences de leur utilisation. Ces modèles, bien que souvent adaptés à un cas d'usage spécifique, partagent des caractéristiques structurelles et fonctionnelles communes. Ces caractéristiques peuvent être catégorisées pour étudier les différences entre modèles. Certains de ces choix de caractéristiques vont avoir des conséquences sur le comportement des fonctionnalités proposées, c'est à dire qu'une fonctionnalité aura des effets différents selon certaines caractéristiques du modèle la proposant.

Cette thèse porte sur le lien entre les fonctionnalités proposées par les historiques de commande et la structure de ces historiques. L'objectif de cette thèse est de présenter une architecture logicielle d'historique de commandes laissant au logiciel le choix du comportement des fonctionnalités parmi au moins ceux étudiés dans la littérature. Cela demande de pouvoir les implémenter de manière non exclusive et combinable avec une même structure d'historique.

En effet, les différents comportements et fonctionnalités proposés par les différents modèles s'avèrent être utiles pour l'utilisateur lors d'une tâche d'édition. Ces modèles ne sont cependant pas combinables, forçant un choix en amont par le développeur du logiciel d'édition, et limitant ainsi les capacités de l'historique, qui ne pourra alors pas répondre à certains besoins de l'utilisateur lors de son travail d'édition. Ce constat est la motivation principale de cette thèse, et a mené à *trois contributions* :

- **La catégorisation des caractéristiques des historiques de commandes** permet de mettre en relation et d'organiser les différents modèles et discussions de l'état de l'art dans le domaine des historiques de commandes comprenant de nombreuses sous parties. Elle établit la liste des choix à faire lors de l'implémentation d'une fonctionnalité et les options existantes qui mènent vers des comportements différents. Elle propose une manière d'approcher le sujet dans sa globalité, et établit ainsi une fondation pour un modèle unifié.
- **Le modèle abstrait ESCI** unifie le fonctionnement des historiques en présentant une structure d'historique qui permet d'implémenter les différents comportements des fonctionnalités de la littérature. Il reprend les concepts fondamentaux de Causality et les étend. La différence principale entre les deux modèles réside dans leur gestion des états, ESCI ajoutant de la flexibilité quant au suivi de l'évolution de données éditées.
- **L'architecture logicielle ReTracer** propose une manière d'implémenter ESCI. Elle permet d'apporter quelques précisions et aborde certains détails d'implémentation du modèle. L'implémentation du modèle permet de mener des expériences pour étudier les apports d'un tel historique au travail d'édition de l'utilisateur ainsi que les interfaces d'interaction adaptées, ce qui fait l'objet de travaux futurs.

Mots clés : ihm, historiques de commandes, structures de données, édition, exploration, collaboration

Centre Inria de l'Université de Lille
Parc scientifique de la Haute-Borne – 40, avenue Halley - Bât. A - Park Plaza – 59650 Villeneuve d'Ascq – France

Developing advanced command histories to improve digital editing processes
A Swiss army knife of command histories model and architecture

Abstract

Command histories are ubiquitous in the field of editing. Several history models have emerged with time, exploring various functionalities like different types of undoing and re-execution as well as managing the consequences of their usage. These models share structural and functional characteristics, albeit often adapted to a specific use case. These characteristics can be categorised to study the differences between models. Choosing between the possible characteristics will have consequences on the behaviour of the proposed functionalities, that is to say a functionality will have different effects depending on the model's characteristics.

In this thesis, we study the link between the command histories' functionalities and the structure of these histories. The objective is to present a command history software architecture giving the system the choice over the functionalities' behaviour. This requires implementing them in a non exclusive and combinable way through one unique history structure.

Being able to choose between the various behaviours can become useful for the user during an editing task. However, the models in the literature aren't combinable, forcing the software developer to make a behaviour choice ahead of time. This limits the history's capabilities, preventing it from answering some user needs. *Three contributions* derive from this statement:

- **A categorisation of command history characteristics** links together the various models and discussions about command histories in the literature. A list of choices to be made at the time of implementation of a functionality and the existing options available is established. This sets a foundation for building a unified model.
- **The abstract command history model ESCI** unifies the various models into one structure capable of replicating the various behaviours presented in the literature. This model is based on Causality and extends its concepts. The main difference between these two models is their state management. ESCI also adds some flexibility in the tracing of data evolution.
- **The ReTracer software architecture** is a way to implement ESCI. It goes into technical details. Implementing this model allows for studies about the benefits of such a command history as well as adequate interaction interfaces, which is a future project.

Keywords: hci, command histories, data structures, editing, exploration, collaboration

Remerciements

Cette thèse m'a permit de travailler sur un sujet passionnant et très riche, à l'intersection entre le domaine de l'IHM (interaction humain machine) et du GL (génie logiciel). Les historiques de commandes m'ont intéressé depuis que j'ai réalisé le potentiel qu'ils avaient au cours d'une session de modélisation 3D avec Blender. Ce logiciel possédait (mais j'y reviens plus tard) un historique qui dépasse la fonctionnalité classique d'Annuler/Refaire globale au document, et permet d'annuler par objet les opérations d'édition. La liberté d'édition que cela me procurait m'a fait réfléchir sur la manière d'implémenter un tel historique, et les autres fonctionnalités que l'on pourrait implémenter avec. C'est pourquoi lorsque j'ai découvert cette offre de thèse, j'ai tout de suite été fortement intéressé. Merci sincèrement à mon directeur de thèse Stéphane et mon encadrant Mathieu de m'avoir donné la chance de pouvoir travailler sur ce projet.

Mais même si le sujet est au cœur du doctorat, j'en ressors avec bien plus qu'une thèse. L'apprentissage du métier de chercheur, la méthodologie de travail, la gestion d'un projet et bien sûr la rédaction de documents sont autant de domaines dans lesquels j'ai beaucoup appris. Donc encore merci à Stéphane et Mathieu de m'avoir guidé tout au long du doctorat et de m'avoir donné les ficelles pour que je prenne le relai. Merci d'avoir toujours été ouvert au dialogue, ce qui a permit de surmonter et d'apprendre des défis qui ont jalonné ces années.

Merci aussi à tous les membres de l'équipe Loki pour votre accueil, les discussions passionnantes très variées que l'on a pu avoir, et la bonne ambiance à l'épreuve de toute les situations ! Merci à mes collègues doctorants, pré-doctorants (stagiaires) et post-doctorants pour les échanges sur nos différents sujets de thèse, les petites discussions et l'entraide au quotidien.

Un avantage de ce sujet est qu'il est simple à expliquer, ou en tout cas à contextualiser. Toute personne ayant édité des données a de fortes chances d'avoir déjà utilisé Ctrl+Z. Cela m'a permit de discuter et de partager avec mes proches et mes amis ce sur quoi je travaille, et leur intérêt pour le projet m'a motivé encore plus ! Merci à eux pour leur mots d'encouragement, et pour ceux qui l'ont pu pour leur suivi de ma soutenance.

J'ai été heureux de pouvoir présenter mes travaux et défendre ma thèse. Merci au jury de thèse qui a posé des questions très intéressantes qui m'ont permit de compléter ce manuscrit. Je souhaite en particulier remercier mes rapporteurs pour leur retours qui m'ont permit de prendre du recul sur le manuscrit lu, et ainsi voir certains problèmes.

Je remercie chaleureusement ma famille qui m'a soutenu tout au long de ces années,

qui m'a transmit la passion pour la recherche et l'informatique et m'a aidé pour les relectures.

Un dernier merci pour les communautés Open Source sans lesquelles ce document ne serait pas le même, et en particulier Blender bien sûr.

Sommaire

Introduction

Contexte : L'édition numérique

L'exercice de communication est décrit comme étant celui de la transmission d'informations, informations qui doivent être pour cela exprimées au préalable. L'expression peut prendre différentes formes. On peut citer l'expression verbale, orale, visuelle ou par le toucher, l'écriture, le dessin, la peinture, la musique, la danse ou encore la sculpture. Certaines formes usent de supports intermédiaires pour stocker l'information, comme le papier, un canevas, un mur, du sable, de la pierre, de l'argile, un magnétophone, un disque vinyle, ... Nous allons dans cette thèse nous intéresser tout particulièrement à l'un de ces supports : le support numérique.

Le support numérique est assez particulier. Là où un coup de pinceau sur un canevas suit des lois décrites par la physique, un "coup de pinceau" numérique, effectué sur un "canevas" numérique, suit d'autres règles de structuration de l'information. Un canevas numérique sera ainsi discrétisé en une grille de pixels, chacun contenant une couleur, valeur discrète prise de l'espace de couleur utilisé. Les informations sont ainsi encodées, représentées par des données numériques. On peut très bien prendre un support non numérique, et l'encoder de manière automatique ou manuelle, à l'aide d'un scanner par

exemple. Dans le cadre de cette thèse, nous allons nous intéresser au cas de l'édition numérique, où l'on crée directement une version numérique des informations.

J'utilise dans ce manuscrit le terme d'édition numérique pour désigner le processus d'expression d'une information à communiquer sur un support numérique. Saisir du texte à l'ordinateur, quel qu'il soit, est l'expression d'informations à travers des mots, sur un support numérique. L'utilisateur exprime l'information à stocker à travers des actions –les appuis de touches par exemple– que l'ordinateur interprète et enregistre en données numériques. L'appui d'une touche devient ainsi une lettre ajoutée à un document texte. Ce processus d'édition est rendu possible par des outils numériques, tels que les logiciels d'édition (logiciels de traitement de texte, de peinture, de modélisation 3D, ...), les logiciels de messagerie (boîtes mel, messageries instantanées, ...) ou tout autre logiciel permettant à l'utilisateur d'exprimer une information, qu'elle soit destinée à autrui au travers de l'ordinateur (messages, ouvrages, œuvres, ...) ou au logiciel même (informations à saisir dans un formulaire ou pour se connecter, pour choisir l'emplacement et le nom d'un fichier à lire ou sauvegarder, ...). Dans ces différents contextes (des *contextes d'édition numérique*), les logiciels proposent à l'utilisateur des outils pour éditer des données.

Sujet : Les historiques de commandes

Les contextes d'édition numérique sont généralement présentés comme proposant une palette de commandes, permettant à l'utilisateur d'éditer les données numériques. Un logiciel de traitement de texte mettra à sa disposition des commandes permettant entre autres d'insérer du texte, d'en supprimer, de modifier leur formatage et de modifier la mise en page du document. L'utilisateur pourra ainsi construire petit à petit ses informations sur le support numérique. Cette palette fixe ainsi le vocabulaire d'édition dont disposera l'utilisateur. Ce vocabulaire est spécifique à un type d'édition et à un logiciel. L'édition de texte ne nécessitera pas les mêmes commandes que celle nécessaires à de la peinture numérique par exemple. Certaines commandes sont plus indépendantes du type d'édition et du logiciel que d'autres, comme Sauvegarder, Copier, Coller, Ouvrir, Fermer, Défaire, Refaire ou encore Répéter. Leur sens reste inchangé, même si leur implémentation peut varier entre logiciels. Parmi ces commandes, Défaire, Refaire et Répéter nécessitent de connaître les commandes précédemment exécutées pour pouvoir être exécutées. Défaire permet d'annuler les effets d'une commande précédemment exécutée, à l'inverse de Refaire qui permet de les réintroduire, et Répéter permet d'exécuter à nouveau une commande déjà exécutée par le passé. Les historiques de

commandes gardent une trace d'exécution, permettant ainsi d'implémenter ce genre de commandes. Cette thèse porte sur la structure et les fonctionnalités qu'un historique peut proposer.

Rôles et modèles d'historiques de commandes

L'utilité des historiques de commandes La fonctionnalité d'annulation que proposent les historiques de commandes a été décrite comme l'un des trois services indispensables à l'IHM par Fekete[Fek96]. Schneiderman affirme que la capacité d'annuler une action effectuée diminue l'inquiétude de l'utilisateur liée à l'exécution d'une opération [Fek96 ; Shn83]. Restituer un état passé permet de corriger des erreurs, mais également de se souvenir de données supprimées et d'explorer d'autres chemins alternatifs d'édition des données. La capacité de naviguer entre versions alternatives est importante pour les travaux de conception dont le but à atteindre évolue avec le temps [MP19], par opposition aux travaux dits de "production" plus dirigés qui nécessitent moins d'explorations tels que le remplissage d'un formulaire [TM02]. Restituer un état passé permet ainsi de corriger des erreurs, mais également de se souvenir de données supprimées et d'explorer d'autres chemins alternatifs d'édition des données. La capacité de naviguer entre versions alternatives est importante pour les travaux de conception dont le but à atteindre évolue avec le temps [MP19], par opposition aux travaux dits de "production" plus dirigés qui pour lesquels l'exploration est moins importante comme la saisie d'un formulaire [TM02]. Cette restitution d'état peut être effectuée manuellement en rechargeant un fichier contenant une ancienne version d'un document, ce qui nécessite d'avoir au préalable enregistré une telle copie. Restituer un état en effectuant manuellement des opérations inversant le travail effectué jusque là n'est pas trivial. Annuler la saisie d'un paragraphe revient à le supprimer. Annuler une suppression de paragraphe nécessite cependant de le saisir à nouveau, ce qui nécessite de s'en souvenir en premier lieu. Annuler des opérations de dessin numérique est bien plus complexe, les tracés se mélangeant sur le canevas. Une fonctionnalité proposant de restituer un état passé des données est alors nécessaire.

"Linear Undo" Les historiques de commandes sont devenus omniprésent dans les logiciels d'édition numérique, mais ils ont peu évolué depuis leur apparition. Les historiques sont le plus souvent des implémentations du modèle "Linear Undo" [CFP06 ; MSI17 ; YAN19 ; NC14] (aussi nommé "Linear History" [BG ; HA17]), qui permet d'implémenter les commandes Défaire et Refaire, nommées Undo et Redo en anglais. Ce modèle

permet d'annuler l'effet des commandes exécutées en dernier à l'aide d'Undo, et de les réappliquer à l'aide de Redo. L'utilisateur peut ainsi corriger des erreurs en annulant les dernières commandes indésirables à l'aide d'Undo. Il peut également consulter des états passés des données éditées en annulant l'effet des commandes exécutées depuis, puis les réappliquer à l'aide de Redo pour reprendre le cours de l'édition. Le modèle "Linear Undo" impose cependant certaines contraintes, limitant son utilité :

— **L'annulation est séquentielle :** Elle ne permet pas d'annuler les effets d'une commande passée autre que la plus récente pas encore annulée. Or la correction d'erreurs nécessite parfois d'annuler sélectivement une commande, où quelle que soit sa position dans l'historique, comme l'ont montré Yoon et coll dans leur étude portant sur les stratégies d'informaticiens dans leurs tâches de développement [YM14]. Cet autre type d'annulation est nommé annulation sélective, SelectiveUndo [Mye+15 ; YM19 ; BG ; Ber94].

— **Des données sont automatiquement perdues :** L'exécution d'une commande autre qu'Undo ou Redo déclenche la suppression de toutes les commandes précédemment annulées de l'historique. En plus d'imposer ainsi la suppression de données potentiellement utiles à l'utilisateur, cela rend la consultation d'états passés risquée, car le déclenchement involontaire d'une commande lui ferait perdre tout ce qu'il avait annulé pour consulter l'état en question.

Les modèles d'historiques avancés D'autres modèles de la littérature exploitent plus amplement le potentiel d'un historique de commandes, ajoutant des fonctionnalités supplémentaires qui seront présentées dans le chapitre 1. On peut mentionner la persistance des données enregistrées [GLL84 ; YAN19 ; AD92], l'annulation sélective [Vit84 ; PK92 ; Ber94] et la gestion de paradoxes [CF06]. D'autres fonctionnalités ont également été développées, le plus souvent en réponse à un problème spécifique à un domaine. On peut citer la gestion de l'annulation régionale pour le traitement de texte [MSI17 ; YM19 ; Sei+12], la gestion des exécutions concurrentielles dans l'édition collaborative [EG89 ; Sun00 ; PK94 ; PK92], l'exploration de variations [Har+08 ; Ter+04 ; KHM17], ainsi que l'insertion de commandes dans l'historique et leur modification dans les logiciels de CAD [Sys ; Fus].

Problématique : La compatibilité entre implémentations de fonctionnalités

Incompatibilités entre modèles Les différentes solutions ont souvent été développées dans un modèle dédié à un usage spécifique et n'implémentent qu'une sélection de

fonctionnalités. Or leurs implémentations sont parfois incompatibles [NC14].

Combinaison des fonctionnalités Le sujet des historiques de commandes a été étudié au moins depuis le début des années 1980. Depuis, les modes d'utilisation des systèmes informatiques ont évolué. Les systèmes interactifs ont pris le pas sur les systèmes réactifs [MDL01], leur utilisation s'est popularisée et les outils numériques font maintenant partie de notre vie quotidienne. L'un des changements notables de ces dernières années est l'augmentation du nombre de logiciels permettant l'édition collaborative. Qu'il s'agisse d'éditeurs de code, de suites bureautiques, d'éditeurs de vidéo ou de dessin numérique, la collaboration est en passe de devenir une fonctionnalité incontournable des logiciels d'édition, et le développement de versions web des logiciels y contribue.

Ces évolutions imposent aux développeurs d'implémenter des historiques de commandes capables de gérer la complexité accrue de l'annulation dans le cadre de l'édition collaborative, tout en fournissant les fonctionnalités d'annulation et de réutilisation de commandes permettant de corriger des erreurs [PK92 ; EG89]. Les historiques doivent aussi permettre d'apprendre à connaître les commandes et d'explorer des idées [Ter+04] ou encore de récupérer des données supprimées de la version actuelle du document [KHM17]. Autant de besoins auxquels il est important d'apporter des réponses, tout en respectant les contraintes imposées par l'environnement, potentiellement collaboratif. Pour ce faire, il est nécessaire de combiner les fonctionnalités de la littérature en un modèle d'historique de commandes.

Il y a entre les modèles des différences d'implémentation, mais également des différences au niveau des effets des fonctionnalités. L'annulation sélective effectuée dans une même situation n'aura pas le même résultat selon que l'on utilise un modèle ou un autre [NC14]. Chacun de ces comportements est présenté comme étant utile, parfois avec des études utilisateur qui démontrent leur intérêt. Proposer ces différents comportements dans un même modèle permettrait de choisir celui qui convient le mieux pour une situation donnée. Cela permettrait également de les comparer, et par la suite de n'implémenter que les plus pertinents.

Le modèle abstrait d'historique de commandes Causality de Nancel et coll propose d'unifier les modèles, combinant les types d'annulations et les stratégies de gestion de paradoxes de la littérature [NC14]. Le logiciel, et par extension l'utilisateur, peut alors choisir librement le comportement adapté à une situation donnée. Aucune donnée n'est supprimée automatiquement, ce qui donne le contrôle des données et de leur utilisation au logiciel et à l'utilisateur. La question de la gestion de l'édition collaborative n'est cependant pas approfondie, et l'implémentation du modèle n'est pas développée.

Adoption et implémentation des modèles Des implémentations d'une forme de fonctionnalité d'annulation remontent au moins aux années 1940 [Lee86], et cette fonctionnalité est maintenant omniprésente. Malgré une riche littérature dans le domaine des historiques de commandes, rare sont les logiciels possédant des historiques dépassant les capacités rudimentaires du modèle "Linear Undo". Les limites de ce modèle ont pourtant été reconnues comme limitantes et des solutions ont été apportées au travers d'autres modèles. Certains articles proposent un modèle abstrait [GLL84 ; PK92 ; NC14], tandis que d'autres développent l'architecture d'implémentation de leur modèle [Vit84] ou proposent une implémentation de leur modèle dans un logiciel [BG93a ; MK96 ; MSI17 ; YM19]. Il se peut que l'absence d'adoption de ces modèles provienne du manque d'investissement de temps dans leur intégration aux logiciels, ceux-ci proposant d'autres moyens d'intégrer l'historique de commandes à l'architecture du logiciel. Fekete dit qu'il faut convaincre la communauté des langages et systèmes que ces services –dont l'annulation– sont indispensables [Fek96], insistant sur le fait que leur importance est sousestimée et qu'ainsi peu d'attention y soit portée. On remarque une évolution progressive de cette situation, où certains logiciels étendent leur historique avec des fonctionnalités comme la persistance de l'historique [Micb], en plus d'être confrontés aux limites du modèle utilisé lors de l'ajout de fonctionnalités de collaboration.

Thèse : Unification des modèles, ESCI et ReTracer

Mes travaux de thèse ont pour but d'apporter une réponse au besoin de pouvoir implémenter différentes fonctionnalités d'historiques de commandes dans un même système ou outil, à l'aide d'un nouveau modèle conceptuel d'historiques de commandes nommé ESCI. Ce modèle est décliné en une architecture logicielle nommée ReTracer proposant une manière générique de l'implémenter. À partir de cette architecture, j'ai développé la librairie ReTracer-rs, utilisable par les développeurs de logiciels pour y intégrer un historique suivant le modèle ESCI. Ces travaux s'adressent d'une part aux chercheurs étudiant le support des fonctionnalités par les modèles d'historiques ou souhaitant effectuer des tests sur ces fonctionnalités, à des fins de comparaison par exemple. Ces travaux proposent également aux développeurs de logiciels une solution pour intégrer un historique avancé à leur logiciel.

Je soutiens dans cette thèse que le modèle ESCI ainsi que son implémentation ReTracer permettent de reproduire les comportement de fonctionnalités d'historique de commandes de manière intercompatible, en donnant à l'utilisateur le pouvoir de choisir la fonctionnalité à utiliser en fonction du résultat souhaité. Pour cela, le modèle présente certaines propriétés.

1. **Universalité des éléments et des commandes :** Le modèle permet au logiciel d'enregistrer toute opération d'édition sur toute donnée éditable. Il permet également de définir la granularité des données et opérations gérées ; l'historique ne considère pas le document comme un seul élément, mais un élément composé récursivement de sous-éléments, et il en va de même pour les commandes modifiant ces éléments.
2. **Pérennité :** Tout l'historique est conservé au cours de l'édition et d'une session d'édition à une autre.
3. **Contrôle :** Le logiciel, et par extension l'utilisateur, contrôle la suppression définitive de données de l'historique. Aucune suppression de données n'est imposée par le modèle de l'historique.
4. **Accessibilité / Organisation des données :** Toutes les données de l'historique sont accessibles chronologiquement et par les liens de causalité qu'elles ont avec les autres.
5. **Altérabilité :** Ces données sont réutilisable, et il est possible de les modifier avant réutilisation.
6. **Cohérence :** L'historique possède des systèmes permettant d'identifier et de gérer les situations incohérentes (paradoxes), et de garantir la validité de sa structure à tout moment.

Contributions et plan du manuscrit

Chapitre 1 : Caractérisation des historiques et de leurs fonctionnalités Dans une première section, je décris plus en profondeur l'édition numérique, qui est le contexte de prédilection de l'étude et l'utilisation d'historiques de commandes. Je commence par étudier les propriétés de l'édition en relation aux historiques de commandes. Je continue par décrire les composants principaux du processus d'édition et présenter le rôle des historiques de commandes au sein de ce processus. Je finis par un exemple de scénario d'édition illustrant concrètement ce contexte d'édition.

La seconde section établit deux états de l'art au sujet des historiques de commandes. Le premier porte sur les types d'interfaces utilisateur d'historiques de commandes, qui sont l'apparence externe des historiques dans les logiciels. Nous y verrons deux manières différentes pour l'utilisateur d'interagir avec un historique de commandes, l'une orientée états, et l'autre orientée actions. Les fonctionnalités mises à disposition par ces interfaces utilisateur seront ensuite présentées, après quoi j'établirais un second état de l'art sur les modèles d'historiques de commandes. La structure et les fonctionnalités de plusieurs modèles y sont présentées. Le but de cet état de l'art est d'organiser ces différents modèles en fonction de caractéristiques structurelles et fonctionnelles, permettant ainsi de les relier entre eux et d'identifier les choix faits par ces modèles qui mènent à des variations dans le comportement de leurs fonctionnalités.

La troisième section établit un état de l'art au sujet des besoins utilisateur et logiciel dans le cadre de l'édition. Le but ici est d'identifier les besoins qui peuvent être répondus par des fonctionnalités d'historiques de commandes. Cela motive la nécessité de créer un modèle unifiant les fonctionnalités et comportements des autres, permettant à l'utilisateur de les choisir en fonction de ses besoins. Je présente ensuite les concepts principaux du modèle Causality, proposant une telle unification. Une synthèse des caractéristiques des modèles d'historiques de commandes et une taxonomie résumant ces modèles en fonction de leurs choix structurels sont ensuite proposées. Pour terminer, six propriétés permettant d'unifier les différentes fonctionnalités au sein d'un même modèle sont déduites à partir de ces caractéristiques.

Chapitre 2 : Modèle d'historiques de commandes ESCI Je commence par présenter, après un rapide aperçu du modèle, ses composants fondamentaux. Chaque composant est défini et relié aux autres, construisant progressivement la structure de l'historique. Après avoir présenté cette structure, nous passerons à l'interface de l'historique présentée au logiciel. Le fonctionnement de l'enregistrement des données est présenté, suivi par

les fonctionnalités de réutilisation des données enregistrées ainsi que la gestion des paradoxes.

La seconde section présente à travers un exemple de session d'édition le fonctionnement du modèle. Cette section illustre l'utilisation d'un historique ESCI pour répondre à des besoins utilisateur présentés dans le chapitre 1.

Une dernière section présente des exemples d'implémentation de commandes avec ESCI reproduisant les fonctionnalités de modèles d'historiques présentés dans le chapitre 1.

Chapitre 3 : Architecture logicielle ReTracer et implémentation Je commence par introduire les composants ESCI présentés dans le chapitre 2 dans l'architecture ReTracer. J'apporte dans cette section les détails d'implémentation de ces composants.

Après avoir présenté l'architecture, la seconde section présente l'API de ReTracer. Le parcours des données de l'historique est d'abord décrit, puis la préparation et l'exécution de commandes.

La troisième section présente une librairie utilisant l'architecture ReTracer pour implémenter un historique ESCI. Un exemple simple d'utilisation est d'abord donné, pour ensuite développer sur certains choix d'implémentation. Pour terminer des outils prédéfinis sont présentés, dont des définitions de commandes d'historique.

Chapitre 4 : Discussion : Validation du modèle ESCI et de l'architecture ReTracer Ce chapitre est une discussion sur l'utilisation, les apports et les limites de ReTracer et ESCI.

La première section présente l'intégration d'un historique ReTracer à un logiciel d'édition.

La seconde section présente l'utilité pour un utilisateur d'utiliser un logiciel d'édition possédant un historique ReTracer et plus généralement ESCI.

La dernière section présente des interfaces utilisateur existantes permettant d'exposer à l'utilisateur des fonctionnalités avancées d'un historique ESCI.

La conclusion résume les apports de cette thèse et ouvre sur les travaux à venir.

Mode d'opération

Mes travaux de recherche ont eu pour point de départ le modèle abstrait Causality. Dans un premier temps, j'ai isolé les concepts fondamentaux du modèle, tels que les

éléments, les commandes, les liens de causalité, et ai commencé à étudier les contraintes d'implémentation imposées par le modèle. J'ai continué par effectuer un état de l'art des modèles et des fonctionnalités des historiques de la littérature, venant enrichir celui de l'article Causality d'une classification des caractéristiques d'historiques, d'une étude de l'interface entre historique et logiciel, et d'une étude des besoins utilisateur et logiciel.

Dans un second temps, j'ai développé l'architecture ReTracer dont la structure permet d'implémenter les fonctionnalités de la littérature. Cela s'est fait en plusieurs itérations, enrichissant l'architecture avec des fonctionnalités des travaux de la littérature. Une fois l'architecture stabilisée, j'ai commencé à développer une librairie d'historiques de commandes l'utilisant. En parallèle, les concepts s'étant distanciés du modèle Causality, j'ai décrit le nouveau modèle d'historique ESCI découlant des capacités de l'architecture ReTracer. Cela m'a permi de me détacher des aspects techniques pour pouvoir décrire le fonctionnement du modèle à un niveau plus théorique. De cette manière, il a été possible d'aborder d'un côté les exigences liées aux fonctionnalités de l'historique à proposer et leur impact sur la structure de l'historique, et de l'autre côté les contraintes techniques qui influencent l'implémentation d'un tel historique et son intégration à un logiciel d'édition.

CHAPITRE 1

Caractérisation des historiques et de leurs fonctionnalités

Dans ce chapitre, j'identifie les différents types d'historiques de commandes proposés dans la littérature et leurs caractéristiques. Chaque modèle possède des comportements qui lui sont propre, mais l'utilisateur peut avoir besoin de plusieurs de ces comportements

au cours d'une même tâche d'édition, en fonction de ses besoins. Or les modèles ne sont pas compatibles. C'est pourquoi le modèle Causality visant à unifier ces différents comportements a été conçu.

Les caractéristiques des différents modèles sont extraites, décrites et organisées en une taxonomie. Cette taxonomie est la base pour la définition des propriétés qu'un modèle unifié doit avoir.

1.1 Définitions

1.1.1 Contexte d'édition

Les publications traitant d'historiques de commandes utilisent souvent le cas des logiciels d'édition pour les mettre en situation ([NC14; MSI17; Chi+98; BG; BR00] entre autres). Des exemples courants sont les logiciels de traitement de texte comme [PK94; MK96; RG99] ou de code comme [HA17; MSI17; YM19; KHM17] et les logiciels de dessin matriciel dont par exemple [NC14; Zha+15; CFP06; Chi+98; Edw+00; GMF10; PK94] ou vectoriel [Ber94; Ham+06; Bue+11]. Alan Dix mentionne dans son livre *Formal methods for interactive systems* [Dix91] la tendance à utiliser les éditeurs de texte comme exemple lorsqu'il s'agit d'étudier les interfaces interactives. Pour mieux comprendre le choix des logiciels d'édition comme contexte d'étude des historiques de commandes, j'ai cherché à caractériser la tâche d'édition numérique en général.

La communication permet de transmettre des informations. Ces informations peuvent être stockées sur un support, tel que du papier, un canevas ou de l'argile sous forme de texte, de partitions, de dessin, de peinture ou de sculpture. Les systèmes numériques sont par essence des supports d'information. Et là où l'expression d'une information vers une feuille de papier peut se faire à l'aide d'un stylo, celle vers un ordinateur nécessite d'autres outils. L'utilisateur fera ainsi usage de périphériques d'entrée tels qu'un clavier, une souris, une tablette graphique ou n'importe quel autre périphérique lui permettant d'exprimer son information. Du côté de l'ordinateur, un logiciel met à disposition de l'utilisateur des commandes, comme l'insertion ou la suppression de texte. Ce logiciel réceptionnera les actions de l'utilisateur, et en déduira la commande à exécuter. Il mettra ensuite à jour les périphériques de sortie, tels que l'écran, pour que l'utilisateur puisse prendre connaissance du nouvel état des données, et ainsi le changement qui a eu lieu. Dans ce manuscrit, j'utilise le terme d'*édition* pour désigner le processus permettant à l'utilisateur d'exprimer une information, stockée sous forme de données numériques dans l'ordinateur. Le terme "éditer" est à prendre dans le sens de

modifier, travailler, construire, créer. L'"édition" ne se rapporte ici pas spécifiquement au processus de préparation à la publication d'un ouvrage rédigé.

Parmi les logiciels existants, il en existe certains qui sont spécialisés dans l'édition, comme les logiciels de traitement de texte, de peinture, de modélisation 3D, de musique, d'animation, de sculpture et bien d'autres. Ces logiciels sont nommés *logiciels d'édition*. Ce manuscrit, me permettant de communiquer les résultats de mes recherches, est un exemple de résultat d'édition numérique. Le texte a été saisi dans un logiciel d'édition de texte (NeoVim), et la plupart des illustrations ont été éditées à l'aide d'un logiciel de dessin vectoriel (Inkscape). D'autres logiciels proposent d'éditer des données de manière plus ponctuelle, comme l'édition d'un message dans un logiciel de messagerie instantanée ou une boîte mel par exemple. Là où le but d'un logiciel d'édition est d'éditer des données, de permettre d'exprimer une information, de produire quelque chose, l'objectif d'une boîte mel n'est pas uniquement d'éditer un message, mais de l'envoyer et d'en recevoir, ainsi que de les lire et les archiver. L'envoi et la réception de messages ne sont pas des actions d'édition, d'expression. Par contre, on entreprend une tâche d'édition lorsque l'on se met à rédiger le message à envoyer, message que l'on veut exprimer, produire. D'autres exemples de tâches d'édition sont la saisie de mots de passe, le choix de l'emplacement et le nom d'un fichier à lire ou à sauver, la modification de l'URL dans un navigateur web, la saisie de la destination à atteindre pour un GPS, ... Donc même si les logiciels d'édition sont spécialisés dans l'édition de données numériques, d'autres logiciels peuvent également proposer d'effectuer des tâches d'édition. Je nommerai ces contextes où l'on peut effectuer une tâche d'édition des *contextes d'édition*, un logiciel d'édition étant un contexte d'édition.

Pour en revenir à la raison d'utiliser la tâche d'édition comme objet d'étude principal, on peut remarquer que les contextes d'édition sont omniprésents dans l'interaction Humain–Machine. Dès que le système requiert des informations de la part de l'utilisateur qui peuvent nécessiter une expression plus complexe que le simple appui sur un bouton "Oui" ou "Non", il propose un contexte d'édition, permettant de saisir du texte par exemple. Ces contextes peuvent être simplement décrits comme un moment où le logiciel passe la main à l'utilisateur, qui va alors s'exprimer à l'aide des outils mis à disposition par le logiciel, et au terme de son expression rendre la main au logiciel, à l'aide d'un bouton "Valider" ou "Terminer" par exemple. Alan Dix mentionne également l'"universalité" de l'édition [Dix91], soulignant le fait que beaucoup de logiciels peuvent être considérés comme des logiciels d'édition, même s'il ne sont pas présentés comme tels. Le navigateur de fichiers en est un exemple, permettant d'éditer les données du système de fichiers.

L'omniprésence des contextes d'édition peut être une raison de leur utilisation pour l'étude d'historiques de commandes. Comme le soulignent Dix [Dix91] et Yang [YAN19], les données que l'on édite sont "contenues" dans un environnement fermé ("closed domains" [Dix91]), isolées d'influences extérieures autres que les exécutions de commandes du système, initiées par les actions de l'utilisateur et, dans le cas particulier de l'édition collaborative, de collaborateurs. L'état de ces données est ainsi librement modifiable, et notamment réversible à un état passé. On peut restituer l'état passé d'un document texte édité à tout moment, là où restituer l'état passé d'un logiciel de messagerie peut introduire des conflits, étant dépendante de l'état des messageries des interlocuteurs. L'utilisateur étant à l'origine des modifications des données, l'historique de ses actions résume l'évolution de leur état [1]. Le fait que l'état des données éditées ne soit pas dépendant de contraintes externes permet de le modifier plus librement. Cette liberté de manipulation peut être une autre motivation pour choisir l'édition comme contexte d'étude des historiques.

1.1.2 Cycle d'édition

Je vais maintenant présenter les composants principaux d'un contexte d'édition, et y introduire les historiques de commandes.

Élément

Au cours de l'édition, le logiciel exécute des *commandes* sur des *éléments*. Ces éléments sont des conteneurs de données qui représentent leur *état*. Les commandes sont des opérations modifiant l'état d'éléments. Les contextes d'édition ont donc des éléments possédant un état et des commandes permettant de modifier ces états.

Il existe plusieurs sortes d'éléments. Dans un logiciel de dessin, le canevas est un élément, lui même composé de sous éléments, ses calques. Ces éléments constituent l'*artefact*, qui sera enregistré dans un document, pour être utilisé ou édité ultérieurement. En complément de l'artefact, le logiciel propose des *outils*, tels qu'un pinceau, dont on peut modifier la taille, la couleur, l'opacité et bien plus. Il peut également proposer des palettes de couleurs ainsi que d'autres outils tels que l'outil de sélection, de remplissage de surface, de rognage, ... Ces outils sont aussi des éléments, éditables à l'aide de commandes. Ils ont des paramètres que l'on peut éditer pour préparer l'édition d'un calque à l'aide de tracés par exemple. On changera ainsi la couleur du pinceau (un outil) avant d'effectuer le trait sur le calque (partie de l'artefact). L'interface graphique peut

1. L'édition collaborative est un cas particulier où il faut gérer plusieurs sources de modifications.

elle-même être éditable, en déplaçant les panneaux amovibles qu'elle contient, ou en zoomant sur le canevas. Elle a donc elle même un état, et peut alors être considérée comme un élément éditable par l'utilisateur. Yang utilise le terme "objet" pour identifier les éléments [YAN19]. Comme certains aspects des éléments diffèrent du concept d'objet en programmation orientée objets, j'utiliserai ici le terme d'éléments.

Commande

À l'exception de quelques modèles, comme le modèle Event-Object prenant une approche plus orientée objet [WG91], la commande est un concept commun à la majorité des articles de la littérature.

Le logiciel définit les éléments et les commandes du contexte, ce qui fixe la granularité de l'édition. Il définit également la liste des commandes dont l'exécution sera signalée à l'historique. Certaines commandes comme Enregistrer ou Copier sont souvent ignorées. Les commandes affectant les éléments constituant l'artefact sont en général pris en compte par l'historique. La prise en compte des commandes n'affectant que les éléments constituant les outils varie fortement d'une implémentation à l'autre. Les logiciels de dessin comme Krita par exemple n'enregistrent en général pas les changements de couleur, de taille, d'opacité ou de forme du pinceau. Les logiciels de traitement de texte comme LibreOffice ont tendance à enregistrer les changements de couleur, de taille et de police de caractères du texte. Le choix des commandes traitées par l'historique détermine le cadre dans lequel il va pouvoir agir.

Interaction utilisateur–logiciel

Les éléments et les commandes sont manipulés par l'utilisateur en interagissant avec le logiciel d'édition. Cette interaction a été présentée sous la forme d'un cycle par Norman [Nor86] et Abowd et Beale. Yang décrit ce cycle comme possédant deux phases, une phase de planification et une autre d'exécution [YAN19]. L'utilisateur évalue l'état courant des données et décide des actions à mener lors de la phase de planification. Cette phase se termine lorsque l'utilisateur lance l'exécution d'une commande à l'aide des périphériques d'entrée de l'ordinateur. Commence alors la phase d'exécution, où le logiciel procède à l'exécution de cette commande, et met à jour ses périphériques de sortie en fonction des résultats de cette exécution. Les historiques de commandes interviennent dans la phase d'exécution, enregistrant une trace de l'exécution de la commande en question et proposant d'autres commandes à exécuter.

Ce cycle se répète, créant ainsi un dialogue d'interaction [YAN19]. L'historique

de commandes enregistrant une trace de ce dialogue permet alors d'exécuter des commandes faisant usage de cette trace. La commande la plus étudiée est l'Undo, permettant d'annuler les effets d'une commande passée.

1.1.3 Les historiques de commandes dans les contextes d'édition

Les historiques ont deux rôles : l'enregistrement d'une trace d'édition, et la mise à disposition d'outils d'exploitation de celle-ci. Le processus d'enregistrement se déclenche automatiquement à l'exécution d'une commande. Du choix des données enregistrées et de la manière de les enregistrer vont dépendre les fonctionnalités de l'historique. Les fonctionnalités de l'historique correspondent à ses capacités, comme ré-exécuter une commande, ou inverser ses effets. L'historique met ces fonctionnalités à disposition du logiciel, qui pourra les présenter sous la forme de commandes d'historique comme Undo et Redo à l'utilisateur à travers une interface utilisateur. Les historiques peuvent ainsi être étudiés à plusieurs niveaux, également identifiés par Yang [YAN19]. La partie manipulant les données enregistrées est la "structure de données et algorithmes" ("data structure and algorithm"), le *modèle d'historique*. Ce modèle décrit le fonctionnement interne de l'historique, contenant les données ainsi que les algorithmes permettant d'implémenter les *fonctionnalités* de l'historique, dont les "techniques de restitution" ("recovery techniques"). Ces différentes fonctionnalités sont mises à disposition du logiciel par l'"*interface de l'historique*" ("internal interface"). Le logiciel mettra à son tour à disposition de l'utilisateur ces fonctionnalités au travers d'une *interface utilisateur* cohérente et compréhensible pour celui ci, la "présentation" ("presentation") de l'historique.

1.1.4 Scénario d'édition

Au cours de ce manuscrit, je vais utiliser des exemples d'édition pour illustrer mes propos. Selon la nature des données éditées, certains concepts seront plus clairement illustrables, c'est pourquoi je ferai parfois l'usage de plusieurs scénarios d'édition. Je garde cependant un scénario principal que j'utiliserai tout au long du manuscrit, celui du dessin numérique [2]. Ce scénario permettra par la suite de décrire assez facilement et visuellement une variété de situations. Sa description permet de présenter l'édition du point de vue de l'utilisateur, ainsi que certains usages d'un historique de commandes.

Les logiciels d'édition permettant cette tâche sont par exemple Krita, Photoshop, Clip Studio Paint, Procreate, GIMP, TV Paint, MyPaint ou encore Paint.net, même si leur spécialité varie (dessin numérique, retouche photo, animation, ...). Voici donc le

2. aussi nommé peinture numérique

scénario d'une session d'édition d'une personne que je nommerai David, effectuant du dessin numérique. Dans ce scénario, David dessine un paysage à l'aide de l'un de ces logiciels.

Croquis David a décidé de dessiner des montagnes avec une montgolfière, et a une idée de la composition qu'il souhaite obtenir, avec une grande montagne au centre. Il commence par faire un croquis grossier de l'idée qu'il a. Il esquisse les différentes parties de la scène, et hésite entre l'ajout d'un soleil ou bien ne mettre que des nuages. Il se décide pour un soleil à droite de l'image et expérimente avec la position et le nombre de nuages dans le ciel, ainsi que leur forme pour obtenir un ciel satisfaisant. Le résultat de certaines modifications ne sera pas convainquant, comme l'ajout d'un sapin, et il décidera alors de revenir à une version antérieure. Il va itérer plusieurs fois sur ce croquis pour obtenir un résultat qui lui plaît, précisant progressivement l'objectif à atteindre.

Détails Son paysage est maintenant prêt à être détaillé. David commence par dessiner sur un autre calque les détails de la montagne, pour ensuite passer au ciel, puis aux autres montagnes et finalement à la montgolfière. Il insiste sur les contours de la montagne, et si il rate un trait, il annule le tracé effectué et essaye à nouveau. Chacune de ces parties du paysage sont conservées sur des calques à part, ce qui lui permet de travailler le ciel sans altérer les montagnes par exemple. Il doit veiller à toujours dessiner sur le bon calque, sans quoi il devra annuler tout le travail effectué sur le mauvais calque et recommencer sur le bon. Il hachure le pan de la montagne, mais n'est pas convaincu par le résultat et se ravise.

Harmonisation Maintenant que les différentes parties sont détaillées, il passe aux ajustements finaux. Il agrandit un peu la montgolfière et déplace légèrement la montagne vers la droite. La couleur de la montgolfière se marie mal avec le reste et David utilise alors un filtre pour modifier sa teinte. Il finit par fusionner les différents calques ensemble et effectue quelques raccordements.

Le paysage est maintenant fin prêt, et pour partager son processus d'édition, il a filmé sa tâche, et en publie une vidéo accélérée sur internet.

Cet exemple a pour but de contenir dans un même projet simple d'édition graphique une variété de situations qui surviennent lors de tâches d'édition. Nous allons régulièrement

revenir sur les différentes situations rencontrées lors de cet exemple de tâche d'édition et les actions qui y sont entreprises.

1.2 État de l'art : Les capacités des historiques de commandes

Les historiques de la littérature offrent une grande variété de fonctionnalités. Celles-ci sont implémentées par leur modèle et exposées au logiciel au travers des fonctions de leur interface. Le logiciel les présentera à son tour à l'utilisateur au travers de son interface utilisateur. Dans cette section sont étudiées les différentes fonctionnalités que peuvent proposer un historique de commandes à travers un état de l'art de ces différentes couches de manière descendante.

1.2.1 Interfaces utilisateur

L'interface utilisateur la plus répandue actuellement représente l'annulation de commandes comme une navigation vers les états passés des éléments.

Navigation d'états

Cette interface utilisateur d'historique permet d'annuler les dernières commandes effectuées séquentiellement. La fonctionnalité principale ici est l'*annulation séquentielle*. Les termes de "retour en arrière" [MDL01] ou même de "voyage dans le temps" [Rek99 ; GLL84 ; BP02] sont parfois utilisés pour la décrire. La commande Undo permet d'annuler une commande, de défaire ce que l'on a fait. En complément d'Undo, la commande Redo permet de refaire ce que l'on a défait ; c'est l'Undo de l'Undo. La fonctionnalité permettant d'implémenter Redo est la ré-exécution d'une commande. Ce sont les opérations de "navigation" entre états, comme décrites par Heer et coll [Hee+08]. Undo permet ainsi de passer à l'état précédent et Redo à l'état suivant. Les deux commandes sont souvent mises à disposition de l'utilisateur par des boutons dédiés. L'historique est parfois représenté comme une liste des commandes, dotée d'un curseur représentant la "position" courante dans cette liste. La figure 1.1 illustre un logiciel d'édition de dessin présentant une telle interface.

Le curseur pointe vers la dernière commande ayant été exécutée. Undo fait remonter le curseur d'une commande, annulant la commande qui était sélectionnée, et redo le descend d'une commande. Les commandes en dessous du curseur ont été annulées, mais restent dans la liste jusqu'à ce que l'on effectue une commande autre qu'Undo et Redo, après quoi elles seront supprimées de l'historique. Le terme de commandes *inactives*

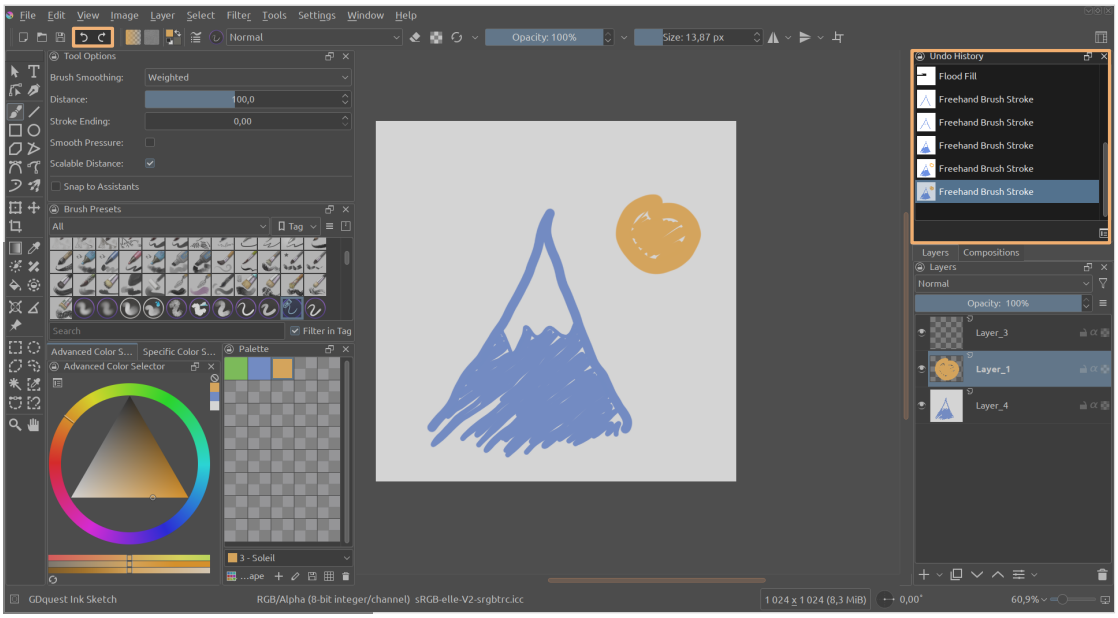

FIGURE 1.1 – Interface graphique de Krita, présentant l'historique de commandes comme une liste des commandes exécutées, et mettant à disposition des boutons pour exécuter la commande Undo et Redo. Le curseur dans la liste est représenté par la commande surlignée en bleu. Les parties de l'interface encadrées en orange permettent d'interagir avec l'historique de commandes.

est parfois employé pour désigner les commandes ayant été annulées, les autres étant alors *actives* [YAN19 ; Kle+02]. Il est possible de cliquer sur une commande pour annuler toutes les commandes situées en dessous.

Cette interface permet d'annuler séquentiellement ; il n'est pas possible d'annuler sélectivement une commande au milieu de la liste sans annuler celles qui la succèdent.

Ce système de navigation entre états est également proposé par d'autres interfaces, comme l'interface UnReT d'Oe et coll [OST13], qui permet d'interagir avec le canevas directement pour sélectionner un état passé.

D'autres interfaces ont adapté le système de navigation pour ne pas perdre l'accès aux états annulés. L'interface graphique de l'historique de Photoshop suit le même modèle d'interaction que l'historique de Krita, et possède en plus une option permettant de désactiver la suppression automatique des commandes inactives à l'exécution d'une nouvelle commande [Phob]. La liste ne fait alors que croître, ajoutant à la fin de celle-ci les nouvelles commandes. Exécuter Undo à partir de la nouvelle commande replacera le curseur sur la commande à partir de laquelle on avait exécuté la commande que l'on vient d'annuler, suivant le lien de causalité entre les deux. Les commandes sont listées

par ordre chronologique, mais Undo suit les liens de causalité. Vim propose de compléter le couple Undo–Redo avec deux autres commandes, faisant naviguer l'utilisateur de manière chronologique et non causale [Vimb]. Later et Earlier font passer le curseur de commande en commande par ordre chronologique, permettant ainsi d'accéder à tous les états du passé. Lorsque l'historique devient conséquent, l'utilisateur peut se sentir perdu, ce qui a donné lieu au développement d'autres interfaces utilisateur, comme les trois vues de Kim et coll proposant un arbre des commandes les regroupant par intervalle de temps, un arbre de commandes regroupées par les tâches effectuées, et une frise chronologique représentant l'édition à l'aide d'aperçus, et pour chaque aperçu ses mots clefs et ses auteurs [KS12]. Ces vues peuvent être parcourues, et à partir de la frise chronologique, il est possible d'ouvrir une vue ciblant une section précise. Il est d'autant plus important de pouvoir comprendre l'évolution du projet dans le contexte de l'édition collaborative, où des modifications peuvent avoir lieu simultanément [KS12].

Si aucune commande n'a été annulée, Redo n'est pas disponible, et certaines interfaces désactivent alors les boutons et raccourcis clavier. Microsoft Word les associent alors à la commande Repeat, permettant de répéter la dernière commande exécutée [Mica].

Le modèle de navigation présente l'historique comme une suite d'états qui ont été créés au cours de l'édition, et permet de les restituer. Même s'il affiche une liste de commandes, et que l'on parle de sélection et de déplacement entre commandes, ces commandes ne font que représenter l'état qu'elles ont généré au moment de leur exécution. On navigue entre états d'éléments. Certains logiciels comme Krita le montrent par l'aperçu de l'état obtenu qui accompagne chaque commande [Kria]. L'accent est mis sur les états, ce que Heer et coll nomment un historique *fondé sur des états* ("state based history") [Hee+08].

La navigation est représentée comme annulant ou rejouant les commandes de la liste séquentiellement par causalité. Il se peut cependant que cette liste ne soit qu'une sélection des commandes de l'historique, choisies en fonction d'un critère de filtrage [BP02]. Dans un contexte d'édition collaborative, elles peuvent avoir été filtrées par utilisateur par exemple, ou par élément, et ne représentent donc pas l'historique complet. Pour l'historique, il s'agit donc d'une annulation sélective de commandes, pas forcément en séquence. Il y a un détachement entre ce que l'interface présente, et la fonctionnalité du modèle d'historique déclenchée.

Édition de l'historique

Heer et coll décrivent également un autre type d'interface d'interaction utilisateur, permettant à l'utilisateur de manipuler le contenu de la liste représentant l'historique [Hee+08]. On retrouve surtout ce type d'interface dans les logiciels de Conception assistée par ordinateur (CAO). Chaque commande de la liste peut être modifiée, déplacée ou supprimée, et une commande peut être insérée n'importe où dans la liste, à l'aide des commandes d'historique Modify, Move, Remove et Insert, respectivement. Cette interface propose plus que l'annulation séquentielle, l'*annulation sélective* de commandes entre autres qui permet d'annuler une commande sans avoir à annuler celles qui la succèdent.

Cette interface donne à l'utilisateur un plus grand contrôle sur l'historique, mais elle introduit également une notion de dépendance inter commandes et de *paradoxes* [NC14]. Toute commande modifiant un objet d'une scène 3D est dépendante de la commande ayant créé cet objet par exemple. Supprimer la commande de création d'un objet mettra les commandes modifiant cet objet dans un état paradoxal, ne pouvant pas s'exécuter comme définit initialement. Un ajustement de ces commandes sera alors nécessaire. SolidWorks permet d'afficher les dépendances entre commandes dans son historique pour pouvoir anticiper ces paradoxes [Sys], et Fusion360 marque les commandes paradoxales en jaune ou rouge, décrivant la source du paradoxe [Fus].

Là où l'interface de navigation présente l'historique comme une archive des états passés qu'elle permet de restituer, l'interface d'édition d'historique le représente comme un élément à part entière, dont la liste de commandes produit un résultat. Changer l'une des commandes de la séquence change le résultat final obtenu lors de l'exécution de cette séquence. Heer décrit ces interfaces comme *fondées sur des actions* ("action based history") [Hee+08], car ce sont les commandes que l'on manipule, et non des états.

L'historique est représenté comme une structure persistante dans le cas de l'interface de navigation [Dri+89], et comme une structure rétroactive dans le cas de l'interface d'édition [DIL07]. Il ne s'agit ici que d'une interface d'historique, et non du modèle de l'historique. Un point commun à ces deux interfaces utilisateur est que les commandes d'historiques ne sont pas enregistrées dans la liste. Dans ces approches, les commandes d'historiques ne sont donc pas considérées comme des commandes classiques, nommées commandes *primitives* par Yang [YAN19], mais comme des *métacommandes* [YAN19]. Ainsi, si l'on utilise l'interface d'édition d'historique, les commandes d'édition affectant l'historique ne sont pas enregistrées par celui-ci. Il n'est donc pas possible d'annuler ces commandes. Pour pallier ce problème, les logiciels possédant une interface d'édition de

l'historique proposent parfois également une interface de navigation d'historique, qui possède dans sa liste les métacommandes de l'interface d'édition. C'est notamment le cas de SolidWorks et de Fustion360.

1.2.2 Fonctionnalités

Les commandes mises à disposition dans ces interfaces peuvent être implémentées à l'aide de deux types de fonctionnalités, l'annulation et la ré-exécution.

Annulation et ré-exécution séquentielle Dans le cas des commandes Undo et Redo des interfaces utilisateur de navigation, Undo est implémentée à partir d'une fonctionnalité d'annulation séquentielle [Ber94 ; JIJ12 ; Man96]. Cette fonctionnalité permet d'annuler, par rapport à un état d'éléments donné, la commande ayant généré cet état. Redo nécessite quant à elle une fonctionnalité de ré-exécution des commandes ayant été annulées. Redo ré-exécute alors une commande par rapport au même état d'éléments que son exécution initiale [Ber94 ; JIJ12 ; Man96]. L'annulation et la ré-exécution sont ici séquentielles.

Annulation et ré-exécution sélective La version séquentielle de l'annulation et de la ré-exécution sont un cas particulier de la version sélective de ces fonctionnalités, étendant leur champ d'application. L'annulation sélective SelectiveUndo s'applique à n'importe quelle commande active, et la ré-exécution sélective SelectiveRedo à n'importe quelle commande inactive [Ber94 ; MDL01].

L'utilisation de l'annulation et de la ré-exécution séquentielle et sélective permet d'implémenter les commandes Modify, Move, Remove et Insert du second modèle d'interface utilisateur. Remove est un SelectiveUndo de la commande visée. Insert peut être implémentée en utilisant Undo pour revenir à l'état précédent le point d'insertion dans l'historique, suivi de l'exécution de la commande à insérer, et un SelectiveRedo de toutes les commandes précédemment annulées. Move est une combinaison de Remove et d'Insert de la commande à déplacer. Modify est un Insert où l'on insert la version modifiée de la commandes en question, et l'on ne ré-exécute pas l'ancienne version.

À SelectiveRedo qui permet de ré-exécuter des commandes inactives s'ajoute SelectiveRepeat, permettant de ré-exécuter des commandes actives. Cette fonctionnalité est un autre cas de ré-exécution, que l'on peut retrouver dans certaines interfaces utilisateur [Mica]. Ces fonctionnalités de ré-exécution peuvent être utilisées pour créer des macros, séquences de commandes enregistrées que l'on peut décider d'exécuter à nouveau [Mye98].

Conséquences d'annulations et de ré-exécutions sélectives L'utilisation de fonctionnalités sélectives peut mener à des situations bloquantes. Certains choix sont à faire quant à la gestion de l'annulation sélective, où l'annulation d'une commande n'annule pas automatiquement les commandes suivantes. Cela signifie que les commandes qui dépendent de produits de la commande annulée se retrouvent dans une situation "paradoxale" [NC14; Ber94; BG; YAN19; PK94; YM19] [3]. Comme mentionné lors de la présentation du second modèle d'interface utilisateur, annuler une commande de création d'objet rendra paradoxale l'exécution d'une commande modifiant cet objet qui n'est plus créé au préalable [NC14]. Plusieurs stratégies ont été proposées. "Cascading Undo" propose d'annuler toute commande paradoxale récursivement, propageant ainsi l'annulation sélective initiale [CF]. Les commandes modifiant l'objet n'étant plus créées sont alors annulées à leur tour. Dans le modèle d'historique "Amulet", le choix est fait de garder le résultat initial de ces commandes [MK96]. Les commandes modifiant l'objet ont alors le même effet qu'initialement, ce qui réintroduit l'objet dans son état modifié. Causality laisse le choix de la gestion de ces paradoxes au logiciel l'utilisant, permettant d'adapter son choix à la situation [NC14].

Il en va de même pour la ré-exécution de commandes dans un contexte autre que celui de son exécution initiale.

1.2.3 Modèles d'historique

Je vais maintenant présenter certains modèles d'historiques proposant des fonctionnalités d'annulation et de ré-exécution séquentielle ou sélective.

Undo et gestion des commandes d'historique

Le modèle d'historique "Linear Undo" est le plus répandu pour implémenter une interface de navigation. Il permet d'annuler séquentiellement des commandes. Sa structure suit la structure d'une interface de navigation simple. Les commandes exécutées sont enregistrées dans une liste et un pointeur pointe vers la dernière commande active. Undo fait remonter le pointeur d'une commande, et Redo le fait reculer. Toute commande annulée est supprimée de l'historique lors de l'ajout d'une nouvelle commande dans la liste. Undo et Redo sont considérées comme des métacommandes et ne sont donc pas enregistrées dans la liste. Berlage illustre cette structure avec la figure 1.2.

Gordon et coll présentent un modèle similaire, "Retract Undo", qui supprime de la liste les commande lors de leur annulation [GLL84]. La perte automatique des

3. Le terme "conflit" est également utilisé pour mentioner des paradoxes dans la littérature

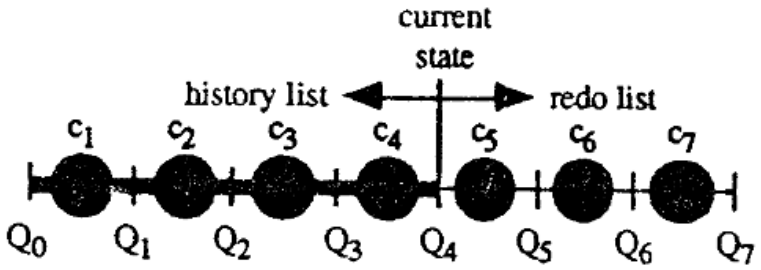

FIGURE 1.2 – Représentation d'un historique de commandes dit linéaire, par Berlage et coll [Ber94]. Les disques représentent les commandes exécutées.

commandes annulées de ces deux modèles limite leurs usages, à l'instar des interfaces de navigation. Certains modèles modifient leur fonctionnement pour conserver les commandes annulées.

Une première approche est d'adopter une structure en arbre, ce qui permet de conserver les commandes annulées dans une branche, et d'en créer une nouvelle à l'exécution d'une commande. Ce comportement est décrit par Gordon et coll au travers de leur modèle "Travel Undo" [GLL84]. Leur représentation de cet historique est fondée sur les états des données éditées, reliés entre eux par relation de causalité, la figure 1.3 en est un exemple.

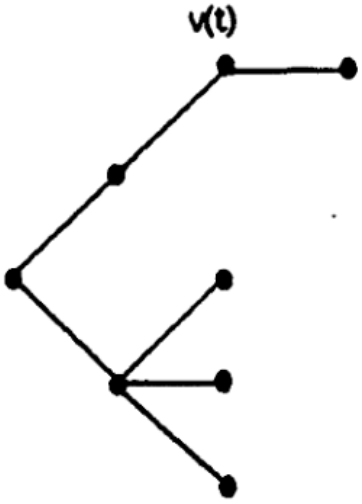

FIGURE 1.3 – Représentation d'un historique de commandes sous la forme d'un arbre, par Gordon et coll [GLL84]. Les disques représentent les états obtenus.

Une autre approche existe également, qui considère Undo et Redo comme des

commandes primitives, et les ajoute à la liste lors de leur exécution. Il n'y a plus ici de notion de déplacement vers un état précédent ou suivant ; il n'y a pas d'interprétation méta de leur exécution. On garde alors une structure linéaire persistante. Gordon et coll présentent ce fonctionnement avec le modèle "Recall Undo". Berlage et Genau représentent cette structure par une ligne dont certaines commandes sont des commandes d'annulation, illustrée par la figure 1.4.

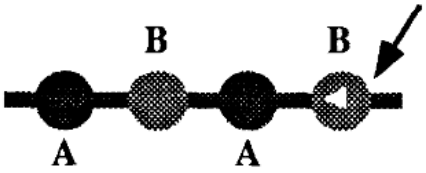

FIGURE 1.4 – Représentation d'un historique de commandes sous la forme d'une ligne, par Berlage et Genau [BG93a]. Les disques représentent les commandes exécutées. Le disque contenant le triangle représente l'exécution d'un Undo.

Deux approches se distinguent ici dans la gestion des commandes d'historique, l'une les considérant comme des métacommande, et l'autre comme des commandes primitives. Yang utilise les termes de modèle d'historique primitif et de modèle d'historique méta pour identifier ces deux types de modèles [YAN19]. L'Undo a pour effet de revenir en arrière dans la structure d'un modèle méta, là où il la fait avancer dans un modèle primitif. Dans un modèle méta, Redo annule l'Undo et permet ainsi d'avancer. Annuler un Undo dans un modèle primitif revient à annuler la dernière commande enregistrée, ce qui est le rôle de l'Undo. Undo peut donc s'auto-annuler. Yang et Gordon soulignent cette différence de commandes d'historique en fonction de leur gestion, les modèles méta proposant le couple Undo–Redo, tandis que seul Undo est proposé par les modèles primitifs. Abowd et Dix présentent cette différence comme une différence d'interprétation de l'annulation, l'une revenant en arrière pour reprendre le chemin souhaité, et l'autre continuant à avancer pour rejoindre ce chemin, et nomment respectivement ces deux approches la restitution régressive ("backward recovery") et la restitution progressive ("forward recovery") [AD92].

La différence d'effets d'Undo sur l'historique selon sa gestion lui donne des caractéristiques différentes. L'Undo d'un modèle méta, désigné par la suite par $Undo_M$, est décrit par Gordon et coll comme étant "dépilant" ("unstacking") [GLL84]. La liste de commandes peut être vue comme une pile, de laquelle Undo ôtera la commande sur le dessus. L'effet d'$Undo_M$ est cumulatif, deux exécutions consécutives ayant l'effet d'une annulation

de deux commandes. L'Undo d'un modèle primitif, désigné par la suite par $Undo_P$, est décrit par Gordon et coll comme étant "auto-applicatif" ("self applicable") [GLL84]. $Undo_P$ est annulable par l'exécution d'un autre $Undo_P$. Selon le type de gestion des commandes d'historique, deux Undo successifs cumuleront leurs effets ou s'annuleront.

SelectiveUndo et stratégies d'annulation

Certains modèles proposent la commande SelectiveUndo, et selon sa gestion des commandes d'historique, SelectiveRedo. Le modèle d'historique "Selective Undo" de Berlage [Ber94], "Selective Undo" de Prakash [PK92] et Amulet [MK96] proposent ces commandes. Le modèle US&R propose Skip, qui, combiné à $Undo_M$ et un Redo, permet d'annuler sélectivement des commandes [Vit84]. Il suffit pour cela d'annuler séquentiellement la commande à l'aide d'$Undo_M$, et de ré-exécuter cette séquence, en en ignorant la première à l'aide de Skip. Redo ici n'est pas purement séquentiel, car Skip a pour effet d'omettre des commandes, et un Redo après un Skip ne va potentiellement pas exécuter cette commande à partir du même état qu'initialement. Ce type de Redo, permettant d'exécuter à nouveau des commandes annulées sans être contraint par leur séquentialité est nommé SelectiveRedo par Berlage et coll [BG]. Cette stratégie d'annulation sélective, retournant à l'état précédent la commande à annuler et ré-exécutant les commandes la succédant, est privilégiée par les modèles méta. Cette commande étant considérée comme une métacommande par ces modèles, elle ne sera en général pas ajoutée à la structure de l'historique, mais ajoutera les commandes qui la composent, c'est à dire Undo et SelectiveRedo. Ces commandes sont à leur tour des métacommandes, donc Undo n'est pas non plus ajouté, mais déplace le curseur de la position courante vers un état passé. SelectiveRedo va ajouter à la structure la commande qu'elle ré-exécute, faisant avancer à nouveau le curseur. Dans les structures persistantes en forme d'arbre par exemple, SelectiveUndo aura ainsi pour effet de créer une nouvelle branche s'attachant à l'état précédant la commande à annuler, et contenant toutes les commandes ré-exécutées [Vit84 ; NC14]. Cette stratégie d'annulation est parfois nommée "Backtracking" [Lee86 ; MDL01]. J'utiliserai ici le terme de *reconstruction*, car sa caractéristique est d'être fondée sur la restitution d'un état passé et de reconstruire à partir ce celui-ci un nouvel état.

Les modèles primitifs utilisent généralement une autre approche, l'annulation par *inversion*. Le modèle "Selective Undo" de Berlage, "Selective Undo" de Prakash et Amulet ne retournent pas à un état passé, mais effectuent l'annulation à partir l'état courant. Pour cela, ils exécutent une fonction ayant l'effet inverse de la commande à annuler. La

définition de cette fonction dépend des modèles. Cette fonction peut être prédéfinie, ou calculée par un système d'inversion. Les trois modèles permettent au développeur de fournir une fonction inverse pour chaque commande créée. Amulet propose aussi une implémentation par défaut [MK96]. Floyd propose un système de correspondance entre commandes inverses [Flo67]. McCarthy quant à lui propose un système de calcul de l'inverse d'une fonction définie par machine de Turing, ce qui permettrait ainsi d'annuler n'importe quelle fonction par application de son inverse [McC56].

Contrairement à l'annulation par reconstruction, l'annulation par inversion permet d'interpréter Undo de plusieurs manières, comme le souligne Berlage [Ber94]. En effet, l'annulation par reconstruction a pour effet d'obtenir un état dans lequel la commande annulée n'a jamais existé. Undo peut cependant être vu comme une commande ayant l'effet inverse de l'exécution initiale. Une commande ayant changé la couleur d'un objet de rouge à vert aura alors pour effet de changer la couleur au rouge. Cette interprétation est le fonctionnement par défaut de l'Undo d'Amulet, et le fonctionnement proposé par Berlage [MK96 ; Ber94]. Ce type d'annulation sélective est nommée annulation sélective directe par Berlage [Ber94]. Mais l'annulation par inversion permet également d'obtenir le même résultat que l'annulation par reconstruction. Prakash et Knister présentent une manière de l'implémenter [PK94], et "REDUCE" de Sun possède cette même approche. À la commande à annuler est associée une fonction inverse, comme pour l'annulation sélective directe. Cette fonction est également exécutée sur l'état présent des éléments, ne nécessitant pas de la restitution d'un état passé. Mais de la même manière que l'annulation par reconstruction ré-exécute les commandes pour que leurs effets soient adaptés au nouvel état d'exécution, l'annulation par inversion de Prakash et Knister présentent un système de transposition de la fonction d'inversion pour prendre en compte les effets des commandes succédant la commande annulée. Par exemple, annuler une commande qui avait changé la couleur d'un objet n'aura pas d'effet si par la suite la couleur avait été affectée à une autre valeur. C'est pourquoi les effets de la fonction d'inversion sont comparés à ceux de toutes les commandes suivantes, et modifiés pour ne pas écraser les changements qui suivent. Ils nomment ce processus la "transposition" de la fonction d'inversion. Le résultat est le même que pour une annulation par reconstruction, mais le processus ne fait que manipuler des fonctions, les comparant et les modifiant. Une seule exécution est effectuée, celle de la fonction d'inversion résultant du processus de transposition.

Ces deux stratégies d'annulation ont été mentionnées par Leeman [Lee86]. La grande différence entre reconstruction et inversion est que là où la reconstruction restitue un état passé et ré-exécute des commandes, l'inversion n'exécute qu'une seule fonction,

obtenue en effectuant des "opérations" (comme la transposition par exemple) avec les commandes enregistrées.

Stratégies de restitution d'état

L'annulation par reconstruction est fondée sur la restitution d'états passés. Undo est un cas particulier de SelectiveUndo, où il n'y a que la partie restitution, sans ré-exécution de commandes. La restitution peut se faire de plusieurs manières. Archer et coll décrivent quatre stratégies de restitution, la ré-exécution intégrale ("Complete rerun"), le rechargement complet ("Full checkpoint"), partiel ("Partial checkpoint") et l'inversion ("Inverse command"). La ré-exécution intégrale ré-exécute l'intégralité des commandes de l'historique pour obtenir l'état à restituer, et part donc de l'état initial. Le modèle Script en est un exemple, où l'historique est une liste éditable de commandes, exécutée intégralement à chaque changement pour obtenir le résultat [ACS84]. Ce modèle ressemble fortement à une implémentation simple d'un historique pour les interfaces permettant l'édition de l'historique. La stratégie de rechargement complet permet d'éviter d'avoir à ré-exécuter l'intégralité des commandes en enregistrant complètement des états intermédiaires, ce qui permet de les recharger par la suite. Le rechargement partiel allège les états enregistrés en n'enregistrant qu'une partie d'intérêt qui a changé depuis son dernier enregistrement par exemple. Et finalement, le rechargement par inversion utilise l'état courant et lui applique l'inverse des commandes le séparant de l'état à restituer, remontant ainsi l'historique. Il existe d'autre stratégies [Yan92], comme l'enregistrement de différentiels représentant les effets d'une commande, ce qui permet de les appliquer à l'état présent pour obtenir celui qui le précède par exemple. La stratégie d'inversion est souvent employée dans les implémentations du modèle "Linear Undo", affectant à chaque commande une commande inverse [Qta].

Gestion des paradoxes

Undo est un cas particulier de SelectiveUndo, où la commande sélectionnée est la commande précédente par lien de causalité, et il en va de même pour Redo et SelectiveRedo, Redo étant la sélection d'une des commandes suivant directement l'état courant par lien de causalité. L'ordre d'exécution est alors conservé, et Undo restitue l'état précédent et Redo un état suivant (car dans le cas d'une structure en arbre, il peut y avoir plusieurs canditats pour Redo [GLL84]). Mais tout SelectiveUndo qui n'est pas un Undo ne conservera pas les liens de causalité d'origine, et il en va de même pour SelectiveRedo. La commande annulée ou exécutée l'est alors à partir d'un état différent

de l'état initial, et la différence entre ces états peut empêcher la commande de s'exécuter. Un exemple classique est l'exécution d'une commande de modification d'un objet qui n'existe pas dans le nouvel état. Annuler sélectivement et exécuter sélectivement une commande sont des actions qui sont des sources potentielles de situations paradoxales. SelectiveUndo a plusieurs manières possibles d'effectuer une annulation, et dans le cas de l'annulation par reconstruction, seule la phase de ré-exécution, déléguée à SelectiveRedo, ne conserve pas les liens de causalité. SelectiveUndo par inversion pourra avoir à gérer des paradoxes lors du calcul de la fonction inverse. Les annulations sélectives directes ne prennent pas en compte les commandes exécutées après la commande à annuler dans le calcul de la fonction inverse, donc seul la compatibilité entre la fonction inverse et l'état courant à partir duquel elle sera exécutée est à vérifier. Pour les annulations sélectives par inversion produisant le même résultat que par reconstruction comme l'annulation par transposition, il faudra en plus vérifier la compatibilité entre la fonction d'inversion et chaque commande exécutée après la commande à annuler.

Pour détecter de potentiels paradoxes, les relations entre les commandes enregistrées sont étudiées. Prakash et Knister mentionnent la notion de dépendance entre commandes [PK94]. L'exécution d'une commande se fait par rapport à un état, qui est le résultat de l'exécution de la séquence de commandes l'ayant généré. Toute commande n'utilisera pas forcément l'intégralité de l'état, et son exécution dépend de ce sous-état. Ainsi, elle est dépendante des dernières commandes à avoir modifié ce sous-état. Ces commandes sont à leur tour dépendantes des commandes ayant modifié le sous-état qu'elles utilisent, et ainsi de suite, ce qui crée un arbre de dépendances pour toute commande donnée. Une commande ré-exécutée ne peut devenir paradoxale que si son sous-état n'est plus le même que celui de l'exécution initiale, et dans le cadre de l'annulation sélective, seules les commandes dépendantes d'une commande annulée peuvent devenir paradoxales.

Bueno et coll mentionnent une autre notion de relation entre commandes, la commutativité [Bue+11]. Deux commandes adjacentes sont décrites comme étant commutatives si leur permutation ne change pas le résultat obtenu. Cela est vrai pour deux commandes indépendantes l'une de l'autre, mais également dans le cas de commandes relatives dont les effets sont permutables. Deux commandes ajoutant respectivement 1 et 5 à une même valeur sont par exemple commutatifs, sans pour autant être indépendants. Dans le contexte des paradoxes, cela signifie que la seconde commande du couple commutatif, même si dépendante de la première, ne deviendra pas pour autant paradoxale lors d'une annulation de la première.

En ce qui concernent les états, la suppression de données va déclencher des paradoxes pour toute commande dépendant de ces données. Ces paradoxes sont décrits comme

des *inconsistances* par Nancel et coll [NC14]. La commande ne peut pas s'exécuter, car il lui manque des données. Ils étendent la notion de paradoxe aux situations où à l'inverse d'un manque de données, des données supplémentaires sont présentes. Si une commande modifie la couleur d'un objet, et que cet objet est dupliqué, ré-exécuter cette commande est possible, mais aucune information n'est donnée quant à l'objet à traiter. Il se peut qu'il faille cibler la copie et non l'original. Ces situations paradoxales non bloquantes, où le nouvel état possède plus d'options d'exécution que l'état initial, sont nommées *ambiguïtés* par Nancel et coll [NC14].

Cass propose avec "Cascading Undo", d'annuler récursivement toute commande dépendante de la commande à annuler, propageant ainsi l'annulation [CF ; CFP06]. Cela change le résultat d'une annulation sélective, car paradoxales ou non, les commandes dépendantes ne seront pas ré-exécutées. Cela a pour effet d'éviter toute situation paradoxale. Ce comportement peut être utilisé comme stratégie de résolution d'une situation paradoxale.

Il est aussi possible de demander au logiciel de changer les paramètres de la commande pour pouvoir l'exécuter, ou tout simplement d'annuler l'opération d'annulation.

Édition collaborative

L'édition collaborative introduit la possibilité pour plusieurs utilisateurs d'éditer de manière plus ou moins indépendante un même document. La collaboration peut être synchrone ou asynchrone [FC00]. Dans le cas de la collaboration asynchrone, les modifications apportées par les utilisateurs ne sont pas systématiquement mises en commun en temps réel. Cette mise en commun se fait sur demande, et nécessite de fusionner les différentes versions obtenues, à la manière d'un système de contrôle de versions [BG93b]. Dans le cas de la collaboration synchrone, l'état du document édité est partagé entre les différents utilisateurs. La multiplicité de sources d'édition introduit la possibilité de cas d'édition concurrentielle, où deux ou plusieurs utilisateurs exécutent une commande simultanément, obtenant ainsi deux états différents, chacun ignorant la modification de l'autre. Les collecticiels utilisent différentes stratégies pour éviter ou résoudre ces conflits [EG89 ; Ste+87].

Verrouillage des accès Une manière d'éviter les conflits est de forcer la séquentialité des exécutions de commandes, au travers d'un verrou à aquérir au préalable par exemple [AD92 ; BG81 ; GM94]. Cela oblige chaque utilisateur à attendre que le verrou soit libéré. Ne verrouiller que la région du document à éditer permet de limiter ces attentes sans pour autant générer de conflit, rendant les commandes exécutées en

parallèle séquentialisable par la suite. Depart la séquentialité imposée à l'édition, l'historique est également séquentiel et tous les utilisateurs ont le même. Les modèles d'historique qui n'ont pas été développés pour la collaboration peuvent quand même être utilisés dans ce cas car l'édition est séquentielle. Ils peuvent être enrichis d'informations au sujet de l'auteur des commandes, ce qui permet d'identifier les commandes exécutées par un utilisateur en particulier. La capacité à annuler sélectivement une commande est ici cruciale, sans quoi il ne sera pas possible d'annuler les commandes par auteur sans avoir à annuler celles des autres [Lv+19]. Les modèles ne proposant que l'annulation séquentielle sont ici moins adaptés, et ne procureraient qu'un historique dit "global" où les actions d'annulation impacteraien le travail de tous les collaborateurs.

Conservation des divergences d'état Une autre approche consiste à résoudre les conflits plutôt que de les éviter. Berlage et coll proposent un historique permettant d'aider à gérer ces situation en les enregistrant sous la forme d'un embranchement [BG]. Il est ainsi possible de choisir la version à conserver, voire de lui appliquer la commande à l'origine de l'autre branche, à la manière d'un "rebase" sous Git [Gitb], ou de fusionner les résultats, à la manière d'un "merge" [Gita]. Un conflit bascule ainsi le mode de collaboration de synchrone à asynchrone à son apparition. L'édition n'est jamais bloquée, et la gestion du conflit permet de resynchroniser l'édition. Le modèle d'historique de Berlage et coll adapté à ce mode de collaboration permet les embranchements et la ré-exécution sélective [BG]. L'annulation sélective est également supportée et permet aux utilisateurs de n'annuler que leurs commandes si besoin est. Un moyen de détecter les conflits est alors nécessaire, par détection de dépendances entre commandes par exemple [EG89 ; Ste+87].

Résolution des conflits : OT et CRDT L'opération de résolution des conflits pour intégrer les changements de toutes les commandes conflictuelles pose la question de la ré-exécution de commandes à partir d'un autre état que celui utilisé à l'origine. Cette opération sera jugée réussie si l'intention de l'utilisateur ayant exécuté la commande initialement est conservée [Sun]. Deux approches à l'adaptation de la commande à ré-exécuter au nouvel état possèdent une littérature étendue : la transformation opérationnelle (OT) et les types de données répliquées commutatifs (CRDT) [Sun].

Supprimer ou insérer du texte d'un document décale le texte qui suit la portion manipulée. Exécutées dans le même ordre, les commandes s'exécutent à partir du même état, et ont par conséquent les mêmes effets. Si deux utilisateurs exécutent simultanément chacun une commande, ils obtiendront chacun un état de document

différent, ce qui mène à un conflit. OT et CRDT proposent des outils pour pouvoir faire converger à nouveau ces deux états. La transformation opérationnelle effectue des opérations sur la commande à exécuter et ses paramètres pour les adapter à l'état courant de chaque document [EG89 ; Sun]. L'approche CRDT ne transforme pas les opérations, mais ajoute dans les données manipulées des identifiants. Ces identifiants sont par exemple donnés aux lettres du texte et serviront de repères pour les commandes [Lit+22].

Le scénario d'utilisation de ces deux méthodes d'adaptation des opérations à un nouvel état est souvent le suivant : chaque utilisateur peut exécuter librement des commandes, les partager avec les collègues, et exécuter celles qu'il reçoit à son tour. Chaque utilisateur obtient à terme un état de document identique, même si les opérations n'ont pas été exécutées dans le même ordre. L'historique de commandes diffère alors entre chaque utilisateur [EG89]. La figure 1.5 illustre la propagation des commandes entre trois historiques dont l'ordre des commandes alors diffère.

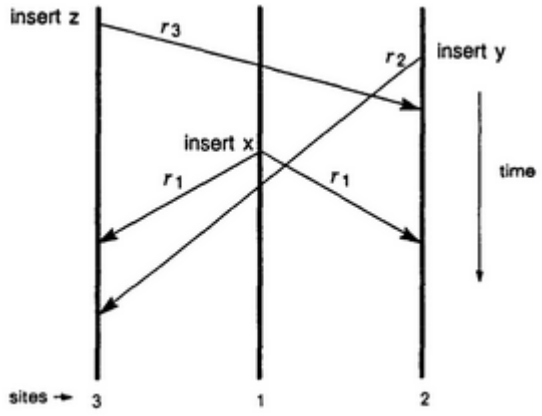

FIGURE 1.5 – Diagramme d'Ellis et coll représentant la propagation des commandes exécutées par trois utilisateurs entre historiques [EG89].

Opérations d'historique avec historiques divergents. Comme l'historique de chaque utilisateur peut diverger, il faut un moyen pour décrire l'opération d'annulation d'une commande indépendemment de l'historique à partir de laquelle elle a été effectuée. Pour cela, un identifiant unique est attribué à chaque commande, ce qui permet de la désigner quelque soit sa position dans les historiques [Che+14 ; CS01]. Aussi, il faut pouvoir enregistrer cette commande d'annulation. Sun et coll proposent de ne pas ajouter aux historiques les commandes d'annulation, mais de marquer les commandes concernées comme étant annulées à l'aide d'un booléen [Sun00]. Annuler une annulation revient à inverser le marqueur en question. Ces actions d'annulation n'ont cependant pas de notion

de temporalité, ce qui empêche leur navigation chronologique. La restitution d'états passés est également impactée par la divergence d'historiques. Un marqueur temporel peut servir pour désigner les commandes à annuler, mais les commandes d'historique n'ayant pas de marqueur temporel, celles-ci devront être désignées autrement. Une alternative à l'approche par OT marquant les commandes annulées est de marquer les données ayant subi l'annulation, approche s'inscrivant dans la méthode CRDT. Les données possèdent un entier représentant leur visibilité. Initialement à 1 lors de son insertion, il est décrémenté lors de sa suppression. Si la valeur est positive, le texte fait partie de l'état actuel du document, sinon il ne sert que de marqueur de position. Il est également incrémenté pour tout Undo d'une commande l'ayant supprimé et décrémenté pour tout Undo d'une commande l'ayant inséré [Lv+19]. Il est ainsi possible de savoir si ce segment est visible ou non et de construire l'état actuel du document. Ce modèle proposé par Lv et coll enregistre ces opérations d'annulation dans l'historique [Lv+19]. Celui-ci ayant une approche CRDT, l'opération utilise les identifiant des données impactées et peut être propagée aux autres utilisateurs tout en conservant le comportement désiré.

Les CRDT et l'OT ont principalement été appliquées pour l'édition de texte, leur utilisation dans d'autres contextes pouvant s'avérer complexe [CS01]. Nous nous concentrerons dans ce manuscrit sur les besoins de ces deux méthodes pour être implémentées, plus que sur la manière de les implémenter pour un type d'édition. Le but est d'étudier les interactions de CRDT et de OT avec les modèles d'historiques de commandes, et les contraintes qu'elles leurs imposent.

Description de régions

Les régions permettent d'exprimer le sous-état dont dépend la commande, ce qui permet de mieux identifier et de gérer les paradoxes et les conflits [Sun00 ; BG]. Différents types de description de région existent, fondées sur leur position dans l'état global ou sur des identifiants existants, comme le présente Sun dans le contexte de l'édition collaborative [Sun]. Dans le contexte de l'édition de texte, Yoon et coll présentent une description de région fondée sur la position du texte ciblé dans le document, avec un système de mise à jour dynamique de ces positions [YM19].

Délégation du choix du résultat

Les modèles d'historique peuvent suivre une gestion primitive ou méta des commandes d'historique, enregistrant ou non la trace d'exécution de ces commandes. Cela aura un

impact sur le comportement d'Undo, tantôt auto-applicatif, tantôt cumulatif. L'annulation proposée peut être séquentielle ou sélective, restreignant ou non l'annulation de commandes à l'exécution la plus récente. Cela définit les capacités d'Undo. L'implémentation de ces annulations peut varier, utilisant soit une approche par reconstruction, soit par inversion. Ces implémentations définissent le comportement de l'Undo qui peut être mené en ignorant les commandes à annuler par l'exécution directe d'une fonction inverse. L'annulation sélective peut mener à la génération de paradoxes, dont la gestion peut se faire de différentes manières. Il faudra alors décider d'une stratégie de résolution à adopter, ce qui influence également le résultat final de l'Undo. Et notamment pour gérer ces paradoxes, la description des régions manipulées par les différentes commandes peut être enregistrée dans la trace d'exécution.

Parmi ces différents comportement de l'annulation et de la ré-exécution et les différentes manières de gérer les paradoxes, les modèles d'historique actuels implémentent la plupart du temps une seule variante. Il peut être cependant utile de laisser le choix du comportement d'une annulation au logiciel d'édition ou à l'utilisateur, selon ses besoins. Pour cela, il faut qu'un même modèle puisse proposer ces différents comportements. Le modèle triadique de Yang est un exemple d'un modèle proposant une des fonctionnalités permettant au logiciel de composer des opérations d'annulation en fonction des besoins de l'utilisateur [YAN19]. Ce modèle a pour but d'être puissant tout en restant simple, et peut être vu comme une extension du modèle linéaire, y ajoutant l'opération de rotation. La liste des commandes annulées est considérée comme à part, et l'opération de rotation permet de déplacer la commande la plus ancienne de la liste à la première position, devant la commande ajoutée le plus récemment. Cela permet d'exécuter Redo sur n'importe quelle commande de la liste, reproduisant l'effet d'ignorer (skip) une ou plusieurs commandes lors de la ré-exécution de commandes annulées. Ce modèle ne présente cependant pas de moyens de conserver de manière pérenne une trace de l'édition.

1.2.4 Architecture des logiciels d'édition

Les fonctionnalités qu'un historique doit posséder fixent les exigences auxquelles doit répondre son modèle. Celui-ci décrit une structure pour y parvenir, et détermine ainsi les contraintes d'implémentation. Il peut être accompagné d'informations complémentaires concernant son implémentation, comme des algorithmes et des descriptions de structure de données adaptés à son implémentation. Ces informations sont importantes, car pour implémenter un modèle d'historique dans un logiciel, il faut savoir comment le modèle

interagit avec le logiciel pour pouvoir remplir ses fonctions.

Interface de l'historique

L'historique de commandes doit être informé des changements d'état des données pour pouvoir se construire. Les architectures logicielles séparant interface utilisateur, logique et données comme MVC, Flux ou Redux simplifient cette tâche. Suite à une action de l'utilisateur, l'interface utilisateur aura à exprimer la modification à apporter aux données sous la forme d'un objet lu par la partie algorithmique du logiciel comme les reducteurs de Redux ou les contrôleurs de MVC, qui mènera à bien la modification. Cet objet a différentes propriétés selon l'architecture, et peut être nommé "commande", "évènement ou "action" en fonction de l'architecture [Red23a ; BJ21 ; MK96]. L'historique peut alors enregistrer une copie de cet objet envoyé par l'interface utilisateur pour représenter le changement effectué. L'architecture du logiciel n'a ici pas à être modifiée pour que l'historique puisse être construit. Il faut cependant que l'historique contienne les informations nécessaires pour pouvoir implémenter ses fonctionnalités comme l'annulation et la ré-exécution des changements enregistrés. Certaines architectures intègrent un historique, comme Interacto et Amulet [BJ21 ; MK96]. Tous deux utilisent le patron de conception "Commande" pour permettre à l'historique d'enregistrer les informations de changements opérés. Chaque commande expose des méthodes. Dans Interacto, une commande possède "execution", "undo" et "redo". Dans Amulet, elle possède "DO" et "UNDO". Ces commandes permettent à l'historique d'exécuter la commande ainsi que d'annuler les effets de cette commande. C'est au logiciel de définir la manière d'annuler chaque commande. Le patron Commande est répandu dans les logiciels actuels, et des frameworks tels que Qt proposent des implémentations du modèle "Linear Undo" prêtes à l'usage [Qta]. Tout logiciel utilisant ce patron de conception pourra alors facilement intégrer un historique. Krita utilise la classe KUndo2Command pour représenter ses commandes [Kri22]. La classe UndoStep de Blender reprend aussi cette notion de commande, et laisse aux différents types de données le soin d'implémenter les fonctions d'undo et de redo [Bleb].

La commande d'exécution enregistrera les données nécessaires à l'exécution d'undo, comme pour une commande de suppression les données supprimées. Les deux logiciels enregistrent ces commandes sur une pile suivant le modèle "Linear Undo" [4].

4. Les versions avant 2.8 de Blender permettaient d'annuler les commandes relatives à un mode [Blee]. Ainsi, il était notamment possible d'annuler toutes les commandes modifiant le maillage d'un objet sans annuler les commandes déplaçant l'objet dans la scène ni les modifications de maillage d'autres objets. Depuis qu'il est possible d'éditer le maillage de plusieurs objets simultanément [Fou], cette fonctionnalité

Une autre approche plus générique à la création d'un historique est la création de copies de l'état des données éditées. Redux intègre un historique de commandes basé sur des copies complètes de l'état des données du logiciel. Toutes les données –modifiées ou non par la commande exécutée– sont enregistrées au lieu de n'enregistrer que l'état passé des données modifiées. Cela élimine l'étape de reconstruction de l'état précédent qui peut être coûteuse en temps, au prix de plus d'espace mémoire. L'approche par copie peut être utilisée localement, comme dans Krita où une copie complète d'un calque est effectuée lorsqu'il est modifié, permettant ainsi de restituer plus rapidement un état passé. Cette copie locale est nécessaire dans certains cas, comme dans le cas d'une commande de suppression d'objet. Des variations à la copie existent, comme l'utilisation de différentiels ou de patches [Pijb ; Pija]. Moins d'espace est utilisé, mais il faut pour obtenir l'état correspondant posséder l'état précédent.

Gestion des données

L'historique effectue un archivage de données nécessaires à l'implémentation de ses fonctionnalités, dont l'annulation. Dans le cas des interfaces d'historiques basées sur des commandes, c'est la commande qui contient ces données. Dans le cas de la création de copies complètes, c'est l'historique qui les gère directement. En plus d'être informé des changements effectués, l'historique peut donc également être informé de l'état du document. Wang et coll proposent de charger les objets du logiciel de la conservation de leur historique [WG91]. Pour suivre le principe d'encapsulation, chaque objet possède son historique et propose les méthodes undo, redo, skipback et skipforward. On obtient ainsi un historique hiérarchique.

Granularité

L'historique a besoin d'être informé des opérations effectuées lors de l'édition. Il impose au logiciel une séquentialisation et une discrétisation des opérations d'édition au travers de la notion de commandes. Ces commandes doivent être définies par le logiciel, définitions qui déterminent ainsi la granularité des opérations d'édition.

Les actions de l'utilisateur génèrent des évènements, interprétés par la suite en commandes [MK96 ; BJ21]. L'exécution d'une commande TracerCalque est ainsi déclenchée par un ensemble d'évènements (un évènement d'appui, des évènements de déplacement et un évènement de relèvement du stylet). Cette commande n'est qu'une étape de la tâche d'édition entreprise par l'utilisateur qui est par exemple le dessin d'un arbre. La

n'est plus disponible, du moins jusqu'à la version 4.0 alpha sortie lors de la rédaction du manuscrit.

commande se situe donc entre les évènements reçus par les périphériques d'entrée et la tâche que souhaite accomplir l'utilisateur. En fonction de la définition des commandes, celles-ci peuvent se trouver plus ou moins proches des évènements. On peut aller jusqu'à définir une commande pour chaque évènement possible. Un évènement ne décrit cependant pas toujours à lui seul une opération d'édition, et une opération déclenchée par le clic d'un bouton sera la conséquence d'un mouvement de souris vers le bouton, l'appui du bouton et enfin le relâchement de celui-ci, en réponse à quoi une opération d'édition sera effectuée. Comme les commandes enveloppent une opération d'édition, elles s'attachent généralement à décrire une action que l'utilisateur veut effectuer, plus que les évènements d'entrée qu'il utilise pour les exprimer. Les commandes sont plus proches des tâches de l'utilisateur et représentent des opérations plus symboliques, plus significative pour celui-ci, comme un tracé de pinceau. La palette de commandes crée ainsi un vocabulaire d'édition utilisé par l'utilisateur et le logiciel.

Les opérations d'édition que peuvent effectuer les commandes sont plus ou moins complexes, allant de l'insertion d'une lettre à un chercher-remplacer dans plusieurs documents. L'opération d'insertion d'une lettre peut être décrite comme étant une opération d'édition atomique, là où un chercher-remplacer est une composition de nombreuses autres opérations. Les commandes n'ont donc pas toutes le même niveau de granularité. Cela a un impact sur l'historique de commandes, comme l'annulation prend en général comme unité la commande. Annuler une fois annulera la dernière commande exécutée. Mais cette unité n'est pas toujours la plus adaptée pour l'utilisateur. C'est pourquoi certains historiques regroupent automatiquement les commandes selon un critère de séparation, et effectuent les annulations par groupe de commandes. C'est une pratique courante dans les éditeurs de texte, regroupant les commandes d'insertion de lettres [Qtb]. Les commandes insérant une espace servent en général de délimitation des groupes, en plus de la détection de pauses dans la saisie. Certains modèles proposent de regrouper les commandes par tâche entreprise par l'utilisateur, comme le dessin d'un arbre, ce qui lui permet de mieux s'y retrouver [RS97 ; LLM20]. À l'inverse, l'utilisateur peut vouloir annuler une partie d'une commande, comme une partie d'un chercher-remplacer. Pour cela, certains modèles permettent de définir des commandes composées de sous-commandes décrivant les étapes dont est composée la commande parente [MK96].

1.3 État de l'art : Besoins utilisateur et logiciel

Les historiques de commandes ont été décrits comme indispensables pour les utilisateurs [Fek96 ; Yan92].

1.3.1 Besoins utilisateur

La littérature IHM a exploré les besoins utilisateurs liés à l'édition numérique, regroupable en cinq catégories.

L'exploration d'alternatives Lors d'une tâche d'édition, l'utilisateur peut vouloir essayer différentes idées. Cette situation est commune aux tâches d'édition impliquant de la conception [Ter+04 ; MP96 ; TM02]. Les objectifs de la tâche peuvent être plus ou moins définis, et peuvent évoluer au cours de l'édition. Maher et Poon parlent de *co-évolution* du problème (les objectifs) et de la solution développée (l'artefact) [MP96 ; MP19]. L'exploration nécessite de pouvoir conserver d'une manière ou d'une autre plusieurs états différents de l'artefact, correspondant aux différentes alternatives.

Des outils dédiés ont été développés pour la gestion d'alternatives. Ils permettent de marquer un état comme étant une alternative, permettant ainsi d'y revenir, de l'éditer, d'en sélectionner une autre, ... On peut ainsi naviguer entre différentes alternatives et les comparer [Ter+04 ; Har+08 ; HZ22 ; KHM17]. L'outil Parallel Paths permet par exemple de dupliquer le projet, y compris l'historique, et de les éditer et visionner en parallèle [Ter+04]. Terry et coll soulignent l'importance de pouvoir expérimenter, les objectifs à atteindre lors de l'édition étant souvent mal définis. Ils mentionnent également l'intérêt d'avoir à manuellement marquer un état comme étant une variation, permettant ainsi de le différencier de tous les autres états plus ou moins importants générés lors de l'édition. L'outil de visualisation d'alternatives de Hossain et coll proposent, dans le cadre de la programmation du comportement de personnages dans un jeu vidéo, un moyen de comparer deux alternatives d'un arbre de décision par différentiels, montrant ce qu'il y a en plus dans l'un par rapport à l'autre [HZ22]. Ils proposent à partir de cette visualisation des commandes permettant de copier des parties de l'arbre d'une alternative à l'autre. Variolite propose de créer des alternatives d'une partie du code édité, permettant de les comparer et d'en changer sans affecter le reste du code [KHM17]. Kery et coll abordent le sujet de la gestion de variantes qui sont des résultats différents obtenus par la variation des arguments d'une commande effectuée. On peut ainsi tester différents arguments pour trouver ceux qui nous conviennent le mieux.

Les Système de contrôle de versions (SCV) sont des outils extérieurs aux contextes d'édition permettant d'enregistrer l'évolution d'un document. Ils permettent d'enregistrer manuellement des versions de ces documents pour une consultation ultérieure. Certains outils analogues sont proposés par les systèmes d'exploitation comme Mac OS intégrant la gestion de versions au niveau de son API à partir de High Sierra.

Kery et coll soulignent également l'utilisation de stratégies de copies de fichiers et de mise en commentaire des développeurs [5], préférant parfois ces stratégies aux SCV. Dans le cadre du dessin numérique, on peut citer la duplication de calques comme stratégie équivalente à la mise en commentaire. Un besoin d'enregistrements localisés à une partie du document et non au document entier, et un détachement de la chronologie pour se concentrer sur le contenu de l'enregistrement motiverait l'usage des stratégies manuelles d'enregistrement de données et non un SCV. La simplicité d'avoir tout "ici", à portée de main, lorsque l'on commente de l'ancien code rend l'expérimentation plus facile par rapport à la gestion par SCV, à court terme en tout cas [KHM17]. Ils mentionnent cependant le problème de ne plus s'y retrouver dans la quantité d'ancien code accumulé dans les commentaires, ou dans des copies du document, remplissant ainsi le dossier. La gestion locale des alternatives de leur outils Variolite a pour but de mieux gérer ces données, tout en les gardant très accessibles.

Des historiques de commandes ont également été développés pour répondre à ce besoin d'exploration, ainsi que des interfaces adaptées. L'outil de micro-versionnage de Mikami et coll propose ainsi dans le contexte d'éditeurs de code de recharger des états passés en fonction de leur position dans le fichier, sous la forme d'indicateurs de différences. Une barre à la fin des lignes concernées ou sous le mot en question par exemple exprime l'existence d'autres versions de cette région. Il est alors possible d'activer cette région et de voir les versions antérieures, notamment pour les recharger. Les historiques persistants en général permettent de garder ces alternatives et de les consulter. D'autres ont créé une interface adaptée à la gestion d'alternatives [MSI17].

Dans les logiciels actuels, certains historiques ont adopté des modèles persistants, comme par exemple Vim, Emacs et Photoshop [Vimb ; GNU ; Adob]. Photoshop propose également un système de Snapshots pour conserver un état d'intérêt [Adoa]. Blender propose d'éditer les paramètres de la dernière commande exécutée, permettant ainsi d'effectuer des tests et d'ajuster le résultat obtenu [Blea].

5. Par exemple, au moment d'écrire cette note, il y avait dans le fichier LaTeXde cette section 110 lignes de texte commentées, à portée de main pour être lues et utilisées à nouveau.

La consultation d'anciens états La conservation d'anciennes données n'est pas uniquement utile dans le cas de l'exploration de variantes. L'utilisateur a parfois besoin de stocker des données qui ne font plus partie de l'état courant de l'artefact. Et pour ce faire, on retrouve l'usage de stratégies mentionnées ci-dessus.

Yoon et coll ont étudié les habitudes d'utilisation de la mise en commentaire de développeurs [YM12]. Lorsqu'ils savaient que le code allait leur être utile ou qu'ils voulaient en désactiver quelques lignes, ils utilisaient cette technique pour garder le code à sa place, sans qu'il ne soit lu par le compilateur ou l'interpréteur, n'ayant ainsi pas d'influence sur le résultat. Pour temporairement activer ou désactiver des parties de code, d'autres stratégies on été mentionnées par les participants interrogés par Yoon et coll, dont l'usage de booléens pour contrôler un bloc d'instructions.

Bhatti et coll ont étudié la gestion de copies du document édité [Bha21]. Les utilisateurs ont tendance à créer une copie du document notamment avant ou après des modifications conséquentes ou incertaines, et en prévention à des pertes de données potentielles ou certaines. Cela leur permet ensuite de les consulter après qu'une erreur soit survenue, ou pour tester et explorer des alternatives, ou encore pour consulter d'anciennes données. Ils mentionnent également l'utilisation de versions pour garder une trace de l'évolution du projet, ce qui permet de mieux comprendre son processus de création et de le partager avec d'autres.

Les historiques permettent de restituer des états antérieurs à l'état courant. Comme mentionné dans le cas de l'exploration, il est nécessaire de conserver les informations, qu'elles aient été supprimées ou non.

La restauration d'un état passé Cette restauration peut être motivée par l'obtention de résultats non souhaités notamment. Il peut s'agir d'une erreur de la part de l'utilisateur [LB19]. L'erreur peut être notamment une action mal effectuée, comme une faute de frappe, ou involontaire, comme l'appui d'une touche par inadvertance. Il peut également s'agir d'une interférence d'interaction, où une commande est exécutée malencontreusement dû à un changement de l'état du logiciel juste avant que l'utilisateur ne déclenche son exécution [Sch+20]. On peut aussi mentionner les perturbations extérieures, telles une personne tierce ou un animal [6] qui viennent perturber une action menée par l'utilisateur, ou en effectuer une eux-mêmes de manière non sollicitée par l'utilisateur. Les historiques sont particulièrement adaptés au besoin de revenir à un état passé, processus parfois nommé *backtracking* [YM14 ; MDL01 ; Mye+15]. Undo, et dans le cas des historiques

6. Les chats par exemple. On remarquera l'existence de dispositifs de protection de clavier empêchant un animal de compagnie de faire la sieste dessus.

méta Redo, permettent alors de restituer l'état souhaité. Il est cependant important que l'historique soit persistant, sans quoi l'utilisateur pourra perdre l'accès à des états passés. Les historiques des logiciels d'édition actuels ont souvent une taille maximale, supprimant les commandes anciennes pour ne pas la dépasser. Certaines tâches sont cependant gourmandes en nombre de commandes à exécuter, comme le hachurage par exemple. Chaque ligne est une commande, remplissant rapidement l'historique, qui devra alors supprimer les commandes correspondant aux premières hachures, rendant ainsi impossible l'annulation de la tâche si le résultat ne convient pas à l'utilisateur.

Dans l'étude de Yoon et coll sur les stratégies de correction d'erreurs, 34% des corrections ont été faites manuellement (sans utiliser les commandes d'historique), et 9.5% étaient de nature sélective, et donc infaisable avec la majorité des historiques classiques actuels. Cela soutient l'utilité de l'annulation sélective. La forte présence d'annulation n'utilisant pas l'historique exprime entre autres un potentiel manque d'adaptation de celui-ci aux situations rencontrées. Il peut s'agir d'une interface utilisateur pas adaptée, ou d'un manque de fonctionnalités. Certains modèles d'historique proposent de donner à l'utilisateur plus de contrôle sur son contenu, pour mieux répondre à ses besoins [YAN19 ; Bue+11].

Une approche alternative est d'aider l'utilisateur à anticiper un problème. Les fortunettes en sont un exemple, exprimant visuellement à l'utilisateur les conséquences d'une action avant qu'elle ne soit effectuée [Cop+19].

Photoshop propose de restaurer une partie du calque à l'aide d'un pinceau permettant d'appliquer à un calque des données extraites d'un état passé. Les navigateurs web permettent d'annuler la fermeture d'un onglet et l'annulation des changements effectués dans la barre de navigation par exemple, ces deux systèmes d'annulation étant disjoints.

Un type d'erreur courante dans le domaine du dessin numérique est de dessiner sur le mauvais calque ("Wrong layer problem" [Dev]). Il faudrait alors pouvoir annuler la séquence de commandes effectuées sur le mauvais calque et les exécuter sur le bon. L'annulation sélective permet d'annuler les commandes en question, et la ré-exécution sélective permet de les appliquer à nouveau, à condition de pouvoir modifier ses paramètres pour changer le calque cible.

Consultation de commandes passées Garder une trace des travaux passées permet de se souvenir de ce que l'on était en train de faire et de comment on a procédé pour mener une tâche. Il est également pratique de pouvoir partager ces informations, ce pourquoi Photoshop propose d'enregistrer un journal des commandes exécutées [Adob]. Cela permet au graphiste de partager le journal avec son client pour qu'il sache ce qui a

été effectué.

Dans le contexte de l'édition collaborative, pouvoir savoir ce que les autres font est très utile, notamment pour synchroniser les tâches [KS12].

Réutilisation de commandes passées Parfois, un travail passe par une phase répétitive, où il peut être utile de pouvoir répéter la dernière, les dernières, ou n'importe quelle commande de l'historique. En plus de la ré-exécution sélective, Kurlander et coll ont mentionné l'utilisation d'un historique comme source de macros [KF00]. Cela permettrait de créer une macro après avoir exécuté une séquence de commandes, sans devoir nécessairement anticiper leur utilité et explicitement demander un enregistrement de celles-ci. À défaut de pouvoir réutiliser toute commande de l'historique, de nombreux logiciels proposent de répéter la dernière exécution [Vima ; Bled], et Gimp sauvegarde les paramètres des filtres appliqués pour les réutiliser par la suite [Gimb ; Gima]. Certains logiciels proposent également d'arrondir les tracés effectués, proposant ainsi une forme limitée d'édition du tracé effectué [Pro].

Qu'il s'agisse d'exploration, de consultation, de restauration ou de réutilisation, tous traduisent un besoin de réutiliser des informations du passé. On peut retrouver tout ces besoins dans une seule tâche d'édition, comme l'exemple de dessin numérique de la section 1.1.4. Lors de la phase de croquis, David va établir la direction du projet et ses éléments principaux. Il va explorer des idées, comparer les résultats et les combiner. Au cours de la phase d'ajout de détails, tout trait insatisfaisant sera annulé et retenté. Il dessine par mégarde les détails du soleil sur le calque de la montagne. Pour corriger cela, il lui faut pouvoir annuler les commandes effectuées sur le mauvais calque et les réutiliser sur le bon calque. Certaines tâches nécessitent de nombreuses commandes pour aboutir, comme les hachures. Pour pouvoir annuler cette tâche dans le cas où le résultat n'est pas satisfaisant, il faut pouvoir restituer l'état précédant ces changements. Les corrections de couleur peuvent nécessiter des ajustements et donc des annulations et de l'exploration. Il serait également utile de pouvoir modifier les paramètres des commandes exécutées. Cela permettrait de corriger le style des traits de pinceau, voire la taille et la forme des tracés. Certaines tâches peuvent être répétitives, comme l'application d'une série de filtres sur plusieurs calques. Pouvoir ré-exécuter les commandes en question sur d'autres calques à la manière d'une macro permettrait d'effectuer cette répétition efficacement.

Les informations nécessaires à ces tâches ont été générées par l'exécution de commandes au cours de la session et enregistrées par l'historique de commandes. Les historiques peuvent donc apporter des réponses à ces besoins.

1.3.2 Besoins logiciel

Besoins actuels

Outre les besoins utilisateur, la conservation par l'historique d'informations au sujet des évènements passés peut enrichir d'autres fonctionnalités du logiciel que celles directement liées à l'historique comme l'annulation et la ré-exécution de commandes.

Aide à l'utilisateur Outre l'usage des fonctionnalités pour proposer des commandes d'historique, la connaissance du travail effectué précédemment permet d'ajuster un agent de communication [RS97]. Elle permet de conseiller l'utilisateur sur la manière d'utiliser le logiciel pour que celui-ci obtienne le résultat escompté et de l'aider à améliorer son processus de travail. Les systèmes d'auto complétion peuvent également bénéficier de l'historique en en extrayant des informations leur permettant d'affiner leurs propositions [MJM12].

Support pour d'autres outils L'historique peut être utilisé par d'autres outils, comme un système de macros [KF92]. Il est ainsi possible de créer des macros à partir de commandes exécutées par le passé. L'historique permet également de conserver des version, comme le fait Photoshop [Adoa]. Il peut être une base pour développer les fonctionnalités d'un gestionnaire de versions. Dans le cas d'historiques enregistrables, les enregistrements peuvent aider à la récupération d'un état valide pour un logiciel qui aurait planté.

Collaboration Avec l'usage accru de logiciels web pour l'édition, le support de l'édition collaborative se répand [KS12 ; Lv+19 ; Ahm+11 ; GWH02 ; WDP99 ; Pal97]. Il faut donc que les historiques soient adaptés à ce cas d'usage, quelles que soient les fonctionnalités qu'ils proposent. Le projet PEPR eNSEMBLE est un témoin de la transition que sont en train de subir les logiciels [CNR]. Il s'attache à permettre à de nouveaux acteurs du monde numérique de proposer des solutions adaptées à des besoins et des contextes d'usage, dans un monde numérique de plus en plus fragmenté et cloisoné. C'est un enjeu de souveraineté et un enjeu sociétal qui dépasse la question de la collaboration, pour aller jusqu'à remettre en question les modes de fonctionnement des outils numériques actuels, dont la gestion et l'accès aux données au travers du principe d'interopérabilité [CNR].

Besoins à venir : L'Interopérabilité

Données et logiciels doivent être séparés ; un logiciel n'a pas l'exclusivité d'un ensemble de données [7]. On peut prendre l'architecture des systèmes de communication tels que l'E-Mail et l'IRC, qui possèdent en leur cœur un protocole. Implémenté par différents logiciels, il permettra à deux utilisateurs de s'envoyer des messages indépendamment du logiciel que chacun utilise.

Les logiciels d'édition possèdent la propriété de conserver dans des fichiers les données éditées, permettant alors à l'utilisateur de contrôler ces données. Les logiciels de communication n'ont pas toujours ce comportement, et utilisent pour certains le modèle centralisé de la gestion de données, où les logiciels clients ne font que lire les données conservées sur un serveur. Il en va de même pour le "cloud computing", où même les logiciels d'édition conservent les données sur le serveur. Une option de téléchargement de ces fichiers est cependant généralement présente.

L'interopérabilité est le principe de permettre à des produits ou services de travailler avec d'autres produits ou services [Doc19a ; Doc19b]. Ce principe est la réponse à cette perte de contrôle croissante de l'utilisateur sur ses données, perte due notamment à la migration des données vers des plateformes inaccessibles autrement que par le logiciel imposé, et le verrouillage des données par l'utilisation de formats dont les spécifications sont gardées secrètes. Le principe d'interopérabilité vient défier cette exclusivité d'accès aux données par un logiciel, et cela sous la forme de lois [Eur], rendant cette interopérabilité obligatoire. On parle alors d'"interopérabilité adverse" [Doc19a], changeant les règles du monde numérique et des modèles financiers des entreprises, en faisant avancer les droits de souveraineté des données. Ce sujet d'actualité est donc un enjeu majeur qui dessinera l'avenir du numérique, et par extension de nos sociétés [CNR].

L'interopérabilité demande des méthodes de communication interlogicielle. À l'instar des formats de fichier ouverts de logiciels et des formats d'échange (comme ODF, JPEG, PNG, FLAC, collada, ORA, ...), les logiciels doivent pouvoir partager entre eux les données des documents, à travers des protocoles et des API par exemple [8]. Mais cela va plus loin que les données de documents, comme dans le cas de la collaboration où ils partagent également des informations sur les actions effectuées par les différents utilisateurs.

7. Cette exclusivité d'accès à des données est la base du phénomène de "jardins privés" (walled gardens), cloisonnant les données des utilisateurs –et par extension les utilisateurs– dans des environnements logiciels [CNR].

8. Beaudouin-Lafon et coll vont jusqu'à supprimer les frontières que sont les logiciels en proposant un modèle d'interaction fondé sur le principe de substrats [Bea23]. L'étude de la notion d'historiques dans ce contexte est le sujet de la thèse d'Alexandre Battut [Bat20].

Les historiques ne sont pas en reste dans cette transition numérique. Ils possèdent des données nécessaires à leurs fonctionnalités et celles d'autres outils les utilisant, comme mentionné dans la section 1.3.2. Dès lors que l'historique n'est pas uniquement local et est supprimé lors de la fermeture du document, permettre à différents logiciels d'éditer un même document nécessite de pouvoir partager ces données d'historique. Collaborer nécessite en plus de pouvoir gérer les conflits d'édition. L'enjeu de l'interopérabilité doit donc être pris en compte dans le développement des historiques.

1.4 Unification des modèles pour répondre aux besoins utilisateur et logiciel

Les historiques de commandes peuvent apporter des réponses aux besoins des utilisateurs et des logiciels. Ces besoins ne sont cependant pas isolés les uns des autres, ce qui nécessite que les réponses soient intercompatibles. De plus, les développements récents décloisonnent les usages et même les logiciels, ce qui nécessite de pouvoir établir des protocoles de communication entre ces logiciels.

Cette unification des modèles soulève de nombreuses questions, car chaque modèle repose sur des structures différentes, les fonctionnalités proposées ne sont pas toutes implémentées de la même manière et le résultat obtenu peut varier de modèle en modèle. Un exemple de divergence est le type d'annulation sélective. L'annulation sélective par inversion possède un mode opératoire différent de celui par reconstruction, et en fonction de la manière d'implémenter l'inversion, les résultats seront identiques à ceux de l'annulation par reconstruction (inversion par transposition) ou non (inversion directe). Un autre exemple est le choix de la stratégie de résolution de paradoxes des différents modèles, qui aura un impact sur le résultat obtenu.

1.4.1 Le modèle Causality

Le modèle Causality de Nancel et coll [NC14] développe la question en définissant un modèle persistant permettant d'implémenter les différentes approches de l'annulation et de l'enregistrement de manière cohérente. Pour cela, il propose d'enregistrer les différents états ainsi que les changements générés par les exécutions de commandes, et décrit les liens de causalité à l'aide d'un vocabulaire adapté. Ces liens possèdent un type, qui exprime les relations entre état et changement. Chaque état impacté par un changement y sera relié par un lien "cible", *Target*, car il est la cible du changement. L'état obtenu post changement sera lié par un lien "résultat" *Result* à ce changement. Les

données d'entrées de la commande comme le tracé effectué seront liées au changement par un lien *Input*, et les états lus, tels que la couleur du pinceau, par un lien *Parameter*. Ce modèle décompose les données éditées en deux groupes, l'Artefact et le Contexte, le Contexte étant l'ensemble des outils d'édition comme le pinceau ou la palette de couleurs. Chaque groupe est décomposé en éléments, que l'on peut ainsi lier par causalité aux changements qui l'impactent. Chaque élément possède un type représentant la nature des données qu'il représente. La région précise de l'élément ayant été impactée par un changement peut aussi être décrite. Les commandes possèdent également un type décrivant l'opération qu'elles effectuent. Les liens Target, Result, Input et Parameter sont créés par l'exécution d'une commande, qui peut être paradoxale, ce pourquoi Causality possède un système flexible de gestion de paradoxes permettant de choisir la stratégie de résolution à utiliser.

Le but de ce modèle est de conserver le plus d'informations possible pour permettre de développer une palette de commandes riche de différentes fonctionnalités, répondant aux besoins de l'utilisateur. Le typage des Elements et des Commandes annote la nature des données et opérations manipulées. Il n'est pas strict et permet d'ajouter une couche d'abstraction par rapport aux types concrets utilisés par l'application. Cela peut être utilisé pour poser les fondations d'une communication interlogicielle, où chaque logiciel a sa correspondance entre type Causality et type concret. Causality permet l'annulation par reconstruction au travers de l'enregistrement de copies de l'état de l'Artefact et du Contexte, ainsi que l'enregistrement d'identifiants (labels) des commandes exécutées pour permettre leur ré-exécution. La gestion des régions et de la collaboration ne sont cependant pas décrits, et l'implémentation du modèle n'y est pas détaillée.

Mon but initial a été de proposer une implémentation de ce modèle, ce qui m'a mené à caractériser les modèles présentés dans la littérature pour pouvoir comprendre l'enjeu de l'unification des modèles et compléter les aspects du modèle Causality peu développés.

1.4.2 Caractéristiques d'un historique de commandes

La caractérisation des historiques existants m'a permi d'identifier les principales divergences entre modèles.

Enregistrement des commandes d'historique

Comme l'a décrit notamment Yang, certains historiques enregistrent leurs propres commandes, et d'autres non [YAN19]. Il les a nommés historiques *primitifs* ou *méta*,

respectivement. Les conséquences sur ses fonctionnalités ont été décrites de nombreuses fois et sous différents angles [GLL84; YAN19; AD92; MDL01]. Les modèles méta proposent le couple Undo–Redo, la commande Undo étant cumulative. Les modèles primitifs ne proposent généralement que la commande Undo, étant auto applicative. Undo peut donc avoir deux comportements, différenciés en Undo_{M} et Undo_{P} par la suite.

On remarque qu'une différenciation similaire est faite au niveau des interfaces graphiques, à travers les termes d'historique basé sur des états ou basé sur des actions de Heer [Hee+08]. Le modèle méta met plus l'accent sur les états obtenu, là où le modèle primitif se concentre sur l'exécution de commandes. On peut faire une analogie avec l'expression populaire "faire et défaire, c'est toujours travailler", où le modèle méta différencie l'action de faire et de défaire, vocabulaire mettant l'accent sur l'état de l'objet de notre travail, là où le modèle primitif considère tout comme du travail, mettant l'accent sur les actions effectuées. Cette différence d'interprétation a été soulignée par Abowd et Dix [AD92]. J'utiliserai par la suite deux représentations pour illustrer de manière cohérente ces deux approches. La représentation de l'approche méta met l'accent sur les états obtenus reliés par causalité, comme dans la figure 1.6, là où l'approche primitive se concentre sur les commandes exécutées, reliées chronologiquement, comme on peut le voir dans la figure 1.7.

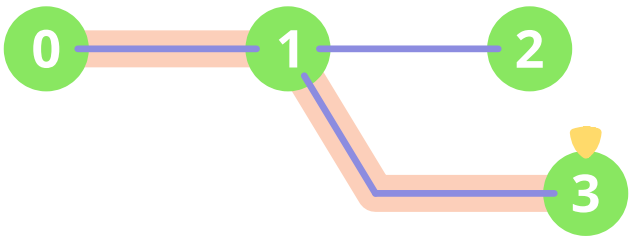

Figure 1.6 – Représentation d'une structure méta. Les états générés par les exécutions de commandes sont représentés par des disques verts. L'état courant est marqué d'un pointeur jaune. Les liens de causalité entre états sont représentés par des lignes bleues, marquées de la commande ayant généré ce changement d'état. Le chemin orange marque les liens menant à l'état courant.

Ces deux illustrations représentent la même session d'édition. À partir de l'état initial 0, une commande 1 est exécutée, générant l'état 1. Il en va de même pour l'exécution de

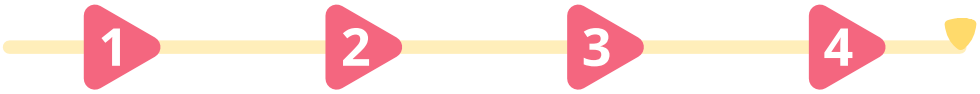

FIGURE 1.7 – Représentation d'une structure primitive. Les exécutions de commandes sont représentées par des triangles rouges. La chronologie est représentée par la ligne jaune. Le triangle jaune marque le présent.

commande 2 générant l'état 2. L'exécution de commande 3 est un Undo, restituant l'état 1, donc le modèle primitif l'enregistre comme exécution 3, là où le modèle méta revient sur l'état 1. S'en suit l'exécution de commande 4 générant l'état 3.

Il est possible de représenter un modèle méta à l'aide de commandes et un modèle primitif à l'aide d'états, ce que certains articles font [Ber94], mais le choix fait ici est plus en cohérence avec l'idée sous-jacente aux deux types d'Undo associés, peut être retrouvé dans la littérature [GLL84 ; BG93a] et reprend l'idée d'historique orienté état ou actions des interfaces d'interaction [Hee+08].

Les liens utilisés dans ces deux modèles suivent la chronologie des exécutions, à la différence près que, dans un modèle méta utilisant un curseur d'état courant, l'annulation est considérée comme un retour en arrière. Cela représente l'indépendance des nouvelles exécutions de commandes par rapport aux effets des commandes annulées. La structure donne ainsi plus d'informations sur les dépendances entre commandes. La représentation d'une annulation séquentielle est triviale et représentée dans la figure 1.6. L'annulation sélective est généralement représentée par une suite d'annulations séquentielles jusqu'à la commande à annuler la plus ancienne, puis l'exécution des commandes annulées à conserver, ceci générant une nouvelle branche (ce qui au passage correspond au fonctionnement de l'annulation par reconstruction). L'annulation sélective n'est pas représentée comme une seule action. Elle est composée de plusieurs modifications : un déplacement du curseur vers un état passé et la génération d'une séquence de commandes. L'ajout de métadonnées permet de regrouper ces opérations pour représenter l'opération parente qu'est l'annulation sélective.

Le modèle primitif représente quant à lui l'annulation comme une exécution de commande comme les autres, ce qui se prête à tout type de commandes d'annulation. L'absence de différenciation entre annulation et les autres commandes ne donne cependant

pas l'occasion d'enrichir la structure des informations de dépendance que possède le modèle méta. Cela peut se faire à l'aide de métadonnées reliant la commande d'annulation aux commandes qu'elle a annulées.

Certains modèles d'historiques dédiés à l'édition collaboative [Sun00 ; CS01] utilisant la transformation d'opérations (OT) marquent les commandes annulées d'une métadonnée au lieu de déplacer un curseur vers une commande passée. Comme elle n'enregistre pas de commande représentant l'annulation, cette approche est considérée comme méta, mais ne met pas l'accent sur les états comme le fait celle utilisant un curseur, plus répandue dans la littérature. Je la désignerai par le terme *structure méta par marquage*. Cette approche alternative reste linéaire mais ne conserve pas de trace du moment où une annulation a lieu, et à l'instar de l'approche méta plus répandue, elle ne regroupe pas les commandes impactées par une même opération d'annulation, à moins d'ajouter ces informations dans les métadonnées.

Types d'annulations

En termes de fonctionnalité, il existe deux types d'annulations, l'annulation séquentielle Undo, et l'annulation sélective SelectiveUndo. Undo est un cas particulier de SelectiveUndo possédant des garanties d'aboutissement de l'exécution de part sa nature séquentielle, que les autres cas de SelectiveUndo n'ont pas. Il en va de même pour Redo et SelectiveRedo. La version sélective de ces commandes permet cependant, au prix de la gestion de paradoxes, d'effectuer un plus grand nombre d'annulations, parfois indispensables [9]. Les figures 1.8 et 1.9 représentent ces deux types d'annulations.

9. Annuler ses dernières commandes dans un contexte collaboratif relève de l'annulation sélective, même si aux yeux de l'utilisateur elle paraît séquentielle

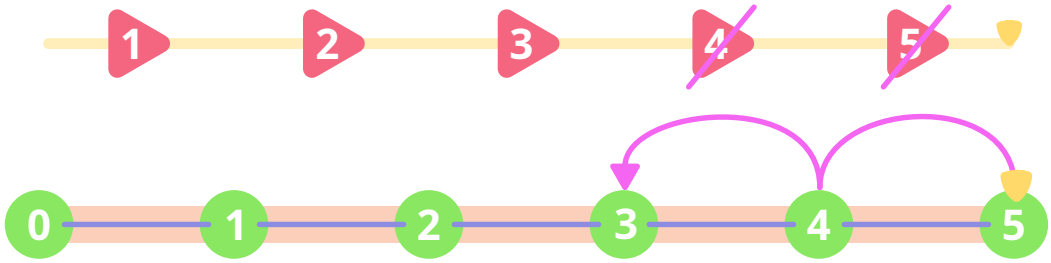

FIGURE 1.8 – Représentation de l'annulation séquentielle, qui restaure uniquement des états passés. Son effet a été illustré sur une représentation orientée actions et une orientée états, l'intention étant exprimable en fonction des états ou des commandes.

FIGURE 1.9 – Représentation de l'annulation sélective. L'annulation sélective annule les effets d'exécutions de n'importe quelle commande passée. Seul une représentation orientée actions a été utilisée ici pour illustrer son effet, l'intention étant exprimable en fonction des commandes.

Stratégies d'annulation

L'annulation peut se faire par reconstruction ou par inversion. L'annulation par inversion peut avoir des résultats différents selon son implémentation. L'annulation par reconstruction permet de décomposer le processus en des sous commandes Undo et SelectiveUndo, ce qui peut permettre de facilement l'adapter à d'autres besoins, comme l'insertion d'une commande en ajoutant une exécution de commande après l'Undo par exemple.

En ce qui concerne la restitution d'un état passé, elle peut se faire de plusieurs manières, dont la ré-exécution intégrale, le rechargement complet ou partiel, l'inversion et les différentiels. Le choix de la stratégie de restitution n'a cependant pas d'effet sur le résultat obtenu, son rôle étant de restituer à l'identique un état passé, donc ce choix n'a pas d'impact sur les fonctionnalités proposées.

Gestion des paradoxes, des éléments et des régions

Les paradoxes potentiels issus des opérations sélectives et de la collaboration peuvent être résolu de plusieurs manières. Annulation en cascade, modification d'un paramètre de la commande automatique ou manuel, abandon de l'exécution de la commande, ...

Pour préciser les données concernées par un lien de causalité dans l'historique ou un paradoxe à l'exécution d'une commande, il est possible de décomposer l'état général en un arbre d'éléments, et décrire les régions concernées. Plusieurs types de descriptions existent, dont celles utilisant la position globale, et celles utilisant des identifiants. Certains systèmes décrivent leur position relative à d'autres commandes, en exprimant leurs dépendances.

Persistance des données

Certains modèles d'historique suppriment automatiquement des données lors de l'exécution de commandes, par nécessité. D'autres ont une structure leur permettant de garder toutes les données de manière persistante. Cela permet à l'utilisateur de restituer un état ou une partie d'un état passé ou d'arguments passés d'une commande. Il peut ainsi expérimenter et ré-exécuter des commandes.

Gestion des conflits

La collaboration nécessite de gérer les conflits pouvant survenir lors de la synchronisation entre les collecticiels. L'édition peut être contrainte à rester séquentielle globalement à l'aide de verrous. En l'absence d'une telle contrainte, des conflits peuvent avoir lieu. L'historique peut les représenter au moyen de branches, ce qui rend le conflit non bloquant. Un système de transformation d'opérations (OT) peut être utilisé pour exécuter les commandes dès leur arrivée, tout en convergeant sur le résultat obtenu, quel que soit l'ordre d'exécution. Les données peuvent également contenir des identifiants permettant d'ajuster l'opération aux différents états et à terme également obtenir le même résultat (CRDT).

Taxonomie des modèles et nécessité de les unifier

Le tableau 1.1 reprend des modèles vus précédemment en précisant leurs caractéristiques.

Les modèles présentés ici se spécialisent dans certains domaines ou se concentrent sur certaines fonctionnalités. Les besoins des utilisateurs et les besoins logiciel actuels

Modèles	Gestion des commandes	Types d'annulation	Stratégies d'annulation	Persistance des données	Gestion des paradoxes	Description de régions	Gestion des conflits
US&R [Vit84]	Méta	Sélectif	Reconstruction	Oui	?	?	?
Script [ACS84]	Méta	Sélectif	Reconstruction	Enregistrement à part	?	?	?
Linear Undo [Qta]	Méta	Séquentiel	Reconstruction	Non	Non	Non	Non
Amulet [MK96]	Méta	Sélectif	*libre*, inversion directe par défaut	Non	?	?	?
Triadic [YAN19]	Méta	Sélectif	Reconstruction	Non	?	?	?
Travel Undo [GLL84]	Méta	Séquentiel	?	Oui	Non	Non	Non
Retract Undo [GLL84]	Méta	Séquentiel	?	Non	Non	Non	Non
Recall Undo [GLL84]	Primitif	Séquentiel	?	Oui	Non	Non	Non
Selective Undo Prakash [PK94]	Primitif	Sélectif	Inversion	Oui	Annulation ou choix utilisateur	Position	OT
Selective Undo Berlage [Ber94]	*libre*	Sélectif	Inversion directe	Oui	?	?	Branches
Cascading Undo [CF]	?	Sélectif	?	?	Annulation par cascade	Dépendances	?
Micro-versioning Mikami et coll [MSI17]	Primitif	Sélectif	Reconstruction	Oui	Choix utilisateur	Dépendances	?
Azurite [YM19]	Primitif	Sélectif	Reconstruction	Oui	Choix utilisateur	Position dynamique	?
ANYUNDO [CS01]	Méta (marquage)	Sélectif	Inversion	Oui	?	Dépendances	OT
ORGAU [Lv+19]	Primitif	Sélectif	Reconstruction [a]	Oui	?	Dépendances	CRDT
Causality [NC14]	Méta	Sélectif	Reconstruction	Oui	*libre*	*libre*	Non

a. Il s'agit ici de reconstruire l'état du document à partir des données annotées, ORGAU fonctionnant par CRDT

Tableau 1.1 – Comparaison entre les différents modèles d'historiques par caractéristique. Les points d'interrogation marquent des informations non trouvées pour le modèle en question, et "*libre*" signifie que le modèle n'impose pas de choix particulier, et peut être adapté à différentes situations.

et à venir montrent qu'il est cependant important de proposer l'ensemble de ces fonctionnalités.

La persistance de l'historique permet à l'utilisateur de restituer tout état passé, et ainsi d'explorer des variantes, de réutiliser des commandes passées, de corriger une erreur et de partager son processus d'édition. Il faut donc que le modèle soit persistant.

L'annulation sélective, la ré-exécution sélective et la gestion de paradoxes. L'annulation sélective est un sur-ensemble de l'annulation séquentielle et elle est nécessaire dans le contexte de l'édition collaborative pour permettre à chaque utilisateur d'annuler ses commandes de manière séquentielle. En plus de permettre l'annulation pseudo séquentielle par utilisateur, elle lui permet de librement annuler une commande de son choix, et ainsi d'annuler d'anciennes commandes indésirables sans perdre le travail effectué depuis. La ré-exécution de commandes permet à l'utilisateur de réutiliser le travail effectué. Cela lui donne l'occasion d'ajuster leurs paramètres pour obtenir le résultat souhaité. En combinaison avec l'annulation sélective, il est possible d'insérer une commande dans l'historique, d'en enlever ainsi que d'en modifier.

L'annulation sélective et la ré-exécution nécessitent de pouvoir gérer les potentiels paradoxes. La manière de gérer ces paradoxes influencera le résultat final obtenu. Il existe de nombreuses manières de résoudre les paradoxes. La stratégie de résolution à utiliser peut varier en fonction de la situation rencontrée. Le modèle doit donc permettre l'annulation sélective, la ré-exécution sélective et la gestion libre des paradoxes.

La gestion des conflits est nécessaire dans le contexte de l'édition collaborative. Le branchage permet de les rendre non bloquants et les opérations de transformations permettent de les résoudre. La gestion des conflits peut être gérés de plusieurs manières, en fonction de la stratégie de synchronisation des logiciels. Le modèle doit donc posséder un système de branchage et permettre de choisir la manière de résoudre les conflits librement, pour être adaptable aux stratégies des différents logiciels.

Les régions permettent de détailler les effets d'une commande, ce qui permet de mieux déterminer les dépendances entre commandes, et ainsi de détecter et résoudre les paradoxes et les conflits. Le langage de description de ces régions dépend de la nature des données éditées et de la manière de les éditer. Le modèle doit donc permettre de décrire des régions sans imposer de langage de description.

L'annulation par reconstruction ou inversion relève plus d'un choix d'implémentation, la première demandant d'enregistrer des données d'état et la seconde de définir des fonctions d'inversion. Selon la situation, il peut être plus simple d'effectuer les commandes à nouveau ou d'exécuter leur inverse. Les deux approches nécessitent d'enregistrer des données, les états pour la première, et les données supprimées par la commande pour la seconde. Un stockage est donc nécessaire quel que soit le choix de l'approche. L'annulation sélective directe est un cas particulier d'annulation qui est propre à l'approche par inversion et qui permet d'obtenir un comportement différent de la reconstruction et de l'inversion par transposition. Le choix de la stratégie de restitution n'a quant à lui aucun impact sur le résultat obtenu.

Le modèle doit donc permettre d'obtenir les différents comportements de l'annulation en proposant l'annulation par inversion ou également l'annulation par reconstruction. Il peut laisser libre le choix de la stratégie de restitution.

La manière de gérer les commandes d'historique a un impact sur sa structure. Les structures méta conservent plus d'informations concernant les liens de dépendance entre commandes. Les structures primitives représentent toutes les commandes de manière uniforme, qu'il s'agisse d'une annulation ou non, et quel que soit le type d'annulation. Les informations de dépendance sont utiles pour la gestion de paradoxes et de conflits, et l'uniformité de la représentation des commandes permet de traiter les annulations en un bloc quel que soit son type. Le modèle doit donc avoir une structure qui permette de conserver les propriétés des structures méta et celles des structures primitives.

Ces caractéristiques apportent des fonctionnalités dont l'utilisateur et le logiciel ont besoin, et les diverses manières de gérer conflits, paradoxes, régions, annulations et ré-exécutions ont été motivées et présentées comme utiles par les articles les introduisant [BG93a ; PK94 ; Ber94 ; CF06].

1.4.3 Exigences pour le modèle unifié

Le modèle unifié doit posséder ces différentes caractéristiques de manière intercompatible. On peut noter que certaines d'entre elles ont des points communs, telles la gestion des conflits et la gestion des paradoxes, toutes deux nécessitant des informations de dépendance entre commandes et de correction de celles-ci, par transformation opérationnelle, intervention de l'utilisateur ou autre. Ces deux fonctionnalités bénéficieront alors de l'ajout d'une gestion de régions précisant ces dépendances. Aussi, l'annulation sélective par reconstruction est composée d'une annulation séquentielle et de ré-exécutions,

et l'insertion, la modification et la suppression de commandes sont décomposables en opérations d'annulation sélective et de ré-exécutions. L'unification des fonctionnalités a donc un intérêt supplémentaire de mettre en commun des fonctions utiles pour différents usages, et de pouvoir les composer pour obtenir d'autres fonctionnalités.

Le modèle doit posséder certaines propriétés pour pouvoir avoir ces fonctionnalités :

1. **Universalité des éléments et des commandes :** toute donnée éditée et toute action d'édition peuvent être enregistrées. L'historique doit être utilisable quelle que soit la nature des données éditées et des commandes disponibles. Ces éléments et commandes doivent être composables et structurables pour permettre au développeur de contrôler la granularité de leur description, et ainsi enregistrer plus ou moins d'informations dans l'historique.
2. **Pérennité :** tout l'historique est conservé au cours de l'édition et d'une session d'édition à une autre. Les opérations manipulant les données de l'historique ne doivent pas être limitées par une suppression de données contrainte par la structure de celui-ci.
3. **Contrôle :** l'utilisateur décide de ce qui est conservé et supprimé définitivement de l'historique. Il peut marquer les états et changements importants et supprimer ce dont il n'a pas besoin. La structure d'historique est au service de l'utilisateur et sa gestion doit donc suivre les besoins de celui-ci et non l'inverse.
4. **Accessibilité / organisation des données :** toutes les données de l'historique sont accessibles chronologiquement et par les liens de causalité qu'elle a avec les autres. La conservation de ces relations organise la structure de l'historique et permet de le parcourir. L'utilisateur et le logiciel ont également accès à ces informations relationnelles.
5. **Altérabilité :** Les données de l'historique sont réutilisables, et il est possible de les modifier avant réutilisation. L'historique ne propose pas uniquement la lecture des données mais fourni des outils pour en faire usage.
6. **Cohérence :** L'historique possède des systèmes permettant d'identifier et de gérer les situations incohérentes (paradoxes et conflits), et de garantir la validité de sa structure à tout moment.

Causality possède les propriétés 2, 4, 5 et partiellement 1 (pas de gestion de granularité) et 6 (pas de gestion des conflits). Toutes les fonctionnalités n'ont cependant pas été intégrées à la structure.

Le but de ces travaux n'est pas de développer un modèle optimal pour une situation donnée. L'optimisation n'a donc pas été un critère premier, mais a guidé certains choix

qui n'impactaient pas les fonctionnalités du modèle. La gestion des données d'état a par exemple été modifiée par rapport à Causality qui a été pris comme point de départ. L'ajout des informations obtenues par la caractérisation des modèles existants a permi d'ajouter et de détailler les différentes fonctionnalités, et a mené au modèle ESCI présenté dans le chapitre suivant.

CHAPITRE 2

Modèle d'historiques de commandes ESCI

Ce chapitre présente le modèle conceptuel d'historiques ESCI. Ce modèle reprend de Causality les notions d'élément, de commande, d'input, de lien de causalité, ainsi que la gestion modulaire des paradoxes et l'inclusion d'une description régionale. Il introduit la notion de session, de vue, d'état et de changement. Les notions reprises de Causality et les nouvelles notions vont être détaillées dans la première sous-section. Dans la seconde sous-section j'illustrerai l'évolution de l'historique au cours d'une tâche d'édition. Dans la dernière sous-section je proposerai des commandes d'historique implémentant les différentes fonctionnalités des modèles de l'état de l'art à partir d'ESCI.

2.1 Le modèle ESCI

2.1.1 Aperçu de la structure

Le modèle d'historique de commandes ESCI a été conçu pour posséder les propriétés nécessaires à l'unification des fonctionnalités des historiques de la littérature. Il fait pour cela usage des concepts fondamentaux d'Elements, de Sessions, de Commands et d'Inputs, à l'origine du nom du modèle. L'universalité des données et des opérations d'édition est obtenue grâce à ces concepts servant d'interface générique de description de l'édition effectuée. Le logiciel utilise la notion d'*Elements* pour communiquer à l'historique des informations sur la structure des données éditées, et la notion de *Commands* pour les opérations disponibles. Cette interface générique Element–Command permet au logiciel de décrire la nature de l'édition permise par celui-ci. Ces notions sont composables, lui donnant ainsi le contrôle sur la granularité de cette description en décomposant les Elements en sous-Elements et les Commands en sous-Commands. Pour permettre à l'historique d'être informé des données d'entrée du projet telles une trajectoire de souris, une date, des données du presse papier ou un fichier lu, le logiciel décrit leurs propriétés au travers de la notion d'*Input*. Finalement, pour suivre le travail d'édition effectué dans un contexte d'édition, une Session lui est associée, qui contient des informations au sujet des Commands exécutées. Ces concepts forment le cadre dans lequel sera construit l'historique.

Pour assurer la pérennité des données de l'historique, la structure de données de celui-ci est persistante. Toute opération d'édition entraîne uniquement des ajouts de données à l'historique ; aucune suppression n'est effectuée. Cette structure découle des quatre concepts E–S–C–I. Lors de l'exécution d'une Command, plusieurs données sont enregistrées :

- Les modifications de données par la Command sont enregistrées sous la forme de *States* (états) représentant les nouveaux états des Elements.
- Les données externes sont enregistrées comme de nouveaux Inputs.
- Les données relatives à l'exécution de la Command comme les arguments qu'elle utilise sont enregistrées par un objet *Change* (changement) dans la Session en cours.
- Le Change est relié aux nouveaux Inputs et States par des liens de dépendance, nommés liens *TRIP*.

Les liens TRIP relient un Change aux Inputs et States en fonction de l'usage qu'il a fait de leurs données. Ils distinguent la lecture, l'écriture et l'enregistrement de nouveaux

Inputs aux moyens de quatre types de liens : Target, Result, Input et Parameter. Le lien Parameter relie le Change aux données qu'il a lues, le lien Input le relie aux données d'entrée qu'il a apportées, le lien Target le relie à un State à partir duquel il a effectué une écriture, et le lien Result le relie au State obtenu par cette écriture.

Le graphe que constituent les Changes, States et Inputs reliés entre eux chronologiquement et par dépendances respectivement par les Sessions et les liens TRIP forme l'historique.

L'universalité et la pérennité sont garanties par ces concepts. L'historique est parcourable par les liens chronologiques et de dépendance, et toute donnée peut être réutilisée ou supprimée à la demande de l'utilisateur. Les modifications se font par transaction, permettant de vérifier leur validité avant de les appliquer, garantissant ainsi la cohérence de la structure.

D'autres notions viennent s'ajouter à ces concepts fondamentaux pour ajouter des informations complémentaires (régions, métadonnées), pour préciser des détails de structuration et de manipulation de l'historique (Types, Versions, Records, References) ou pour gérer des situations particulières (réutilisation et paradoxes, synchronisation et conflits).

2.1.2 Composants fondamentaux

Je vais maintenant définir les différents concepts d'ESCI.

Element, ElementVersion et State

Un Element (E) représente une entité à laquelle on associe des états. Cela peut être un canevas, un calque, un pinceau, une couleur, une valeur d'opacité, de taille de pinceau, ... Son rôle est uniquement d'identifier cette entité. On peut comparer cela à une adresse de variable ou son nom, ou le concept d'Entité de l'architecture logicielle ECS. Un Element a un nom unique et un type (*ElementType*) décrivant la nature des données qu'il identifie. On peut ainsi représenter un projet de dessin numérique Projet_Paysage par un ensemble d'Elements comme Canevas0, Calque_Montagne, Calque_Ciel, Calque_Arrière_Plan, Pinceau_Calligraphie, Pinceau_Carré, Palette_Pastels et Palette_Printemps. Le choix de ce qui est représenté comme un Element est laissé à la discrétion du développeur, lui permettant de décider du niveau de granularité de la description de la structure des données.

Un ElementType possède à la manière d'une classe la liste des attributs (*Attributes*) que les données identifiées par l'Element ont à tout moment de l'édition et facultativement des liens d'héritage avec d'autres types.

FIGURE 2.1 – Exemples d'ElementTypes (en italique), de leurs Attributes (en gris) et d'Elements (en gras) pour un projet de dessin.

Les Attributes permettent de décrire les parties des données identifiées par l'Element, comme taille et opacité pour les pinceaux. Un ElementType Pinceau contiendra la liste de ces parties. L'ElementType Canevas peut contenir espace_de_couleur et dimensions. La figure 2.1 illustre les ElementTypes associés à des Attributes, et les Elements de ce type. Un canevas possède également une liste de calques dont la taille varie au cours de l'édition. Comme l'existence de ces parties n'est pas garantie, aucun Attribute n'est déclaré dans l'ElementType pour les représenter. À chaque Attribute déclaré peut être associé un ElementType pour décrire le type de données que cette partie contient.

Dans ESCI, les informations d'héritage sont utilisées pour marquer l'équivalence entre deux types. Un ElementType RGBA peut être marqué comme héritant du type RGB pour exprimer le fait que les données identifiées par l'Element de type RGBA peut être utilisé là où des données identifiées par un Element de type RGB est attendu.

Les ElementTypes permettent de prendre en considération la nature et la structure des données éditées. Ils décrivent les liens entre données au travers d'Attributes. La notions d'Element permet d'identifier l'ensemble des états relatifs à une entité au cours de l'édition, au sein de toutes les données éditées. Les différents états que prend un

calque au cours de l'édition seront identifiés par son Element. Ce calque pourra être supprimé du canevas, réinséré, déplacé au dessus ou en dessous d'un autre calque ou déplacé vers un autre canevas, c'est pourquoi cet identifiant est nécessaire pour le suivre parmi les autres données. Cela permet notamment par la suite de parcourir ses états, d'en restituer, et d'identifier pour chaque commande exécutée de quels Elements elles ont changé l'état. L'annulation régionale peut se faire par Element, en restituant l'un de ses états passés sans modifier l'état des autres.

La position et l'état de l'entité identifiée par un Element évoluent au cours de l'édition. C'est pourquoi elles sont enregistrées dans des ElementVersions.

EV **L'ElementVersion (EV)** est un bloc de données relatif à un Element. Une ElementVersion de Canevas0 contient les données de l'Element à un instant de l'édition. On peut comparer une ElementVersion aux données d'une variable ou d'un objet, où l'Element est son identifiant unique et l'ElementType de cet Element son type. Et de la même manière que l'on peut assigner à l'attribut d'un objet un autre objet, une ElementVersion peut aussi avoir une autre ElementVersion comme Attribute. Les ElementVersions sont ainsi organisées sous la forme d'une arborescence dans laquelle une ElementVersion dite parent peut être reliée à des ElementVersions dites enfant.

Ces liens sont identifiés par des Attributes de l'ElementVersion parent. Il s'agit entre autres d'Attributes déclarés dans l'ElementType associé à l'Element de l'ElementVersion. Cela n'est cependant pas une obligation, ce qui permet d'ajouter à une ElementVersion de Canevas0 des Attributes la reliant aux ElementVersions des calques enfants qui la composent au moment de la création de l'ElementVersion. Une version du canevas pourra avoir calque_0, calque_1 et calque_2, et une autre n'avoir que calque_0, après suppression de deux calques de la liste. Aussi, tous les Attributes déclarés dans l'ElementType d'une ElementVersion n'ont pas à être lié à une ElementVersion enfant. Les parties qu'ils décrivent seront alors enregistrées dans l'ElementVersion même.

La figure 2.2 représente un exemple d'arborescence d'ElementVersions pour un projet de dessin à un moment de l'édition.

Les ElementVersions permettent de segmenter par Element les données éditées, et de conserver les données passées. Les données des ElementVersions sont immuables. Elles permettent à l'historique de conserver de manière pérenne les informations relatives à l'état des données au cours de l'édition. La modification des données identifiées par un Element engendre la création d'une nouvelle ElementVersion. Une nouvelle ElementVersion est aussi créée lorsqu'une ElementVersion enfant est ajoutée, supprimée, ou remplacée par une ancienne ElementVersion ou une ElementVersion d'un autre

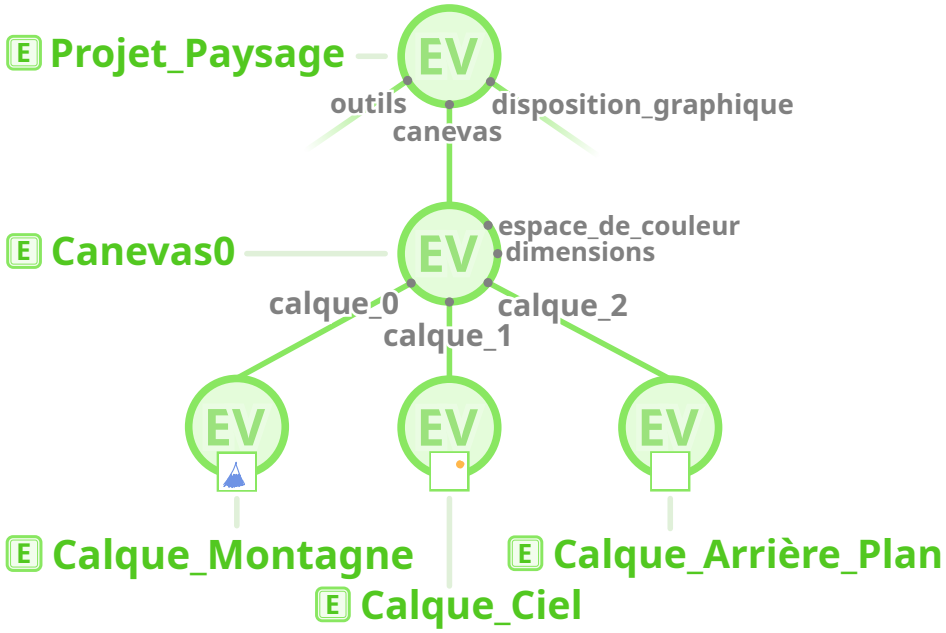

FIGURE 2.2 – Exemple d'arborescence d'ElementVersions d'un projet de dessin. La partie reliée à outils contenant notamment les pinceaux et les palettes, et celle reliée à disposition_graphique contenant par exemple la position et le niveau de zoom sur l'image, la positions des fenêtres et des barres d'outil, ont été exclues de l'illustration pour ne pas la surcharger.

Element. Si une ElementVersion enfant est remplacée par une autre suite à une modification de données, aucune ElementVersion ne sera créée pour le parent. Cela évite une réaction en chaîne où une ElementVersion serait créée pour tout parent d'une nouvelle ElementVersion. Cela signifie cependant que les liens enfant d'une ElementVersion la relie aux ElementVersions courantes des Elements enfants au moment de sa création. Ces liens ne seront plus à jour lors d'une modification de données d'un Element enfant, créant une nouvelle ElementVersion. Celle-ci sera cependant retrouvable au besoin en suivant l'évolution de l'Element enfant.

CAUSALITY ne possède pas d'ElementVersion, mais sépare les données éditées en deux : l'Artefact (les données à éditer : un canevas, un document texte, une scène 3D, ...) et le Context (les données nécessaires à l'édition de l'Artefact : la couleur active, la taille

et le style d'un pinceau ou du texte saisi, le calque sélectionné, la position actuelle dans un document, ...). De nouvelles versions de ces deux parties sont enregistrées lorsque leurs données sont éditées. Les éléments y sont des pointeurs vers la région de l'Artefact ou du Context correspondant.

Les ElementVersions sont une généralisation de cette séparation, segmentant les données en une arborescence où chaque nœud correspond à un Element. Les ElementVersions permettent d'enregistrer de plus petits blocs de données, dont la taille peut être réduite en le décomposant en sous-blocs à l'aide d'ElementVersions enfant, ce qui donne au développeur le contrôle sur la granularité de la segmentation des données.

Il est possible de recomposer tout état d'un Element à partir de l'ElementVersion qu'il y avait à ce moment, en y ajoutant l'ElementVersion de chaque enfant correspondant à ce moment. Cet arbre d'ElementVersions ainsi obtenu est un State (état) de l'Element.

 **Le State (St)** d'un Element est l'arbre d'ElementVersions qui l'a constitué à un moment donné de l'édition. Un State pointe vers l'une des ElementVersions de l'Element en question et vers toutes les ElementVersions qui le composait à ce moment. Le canevas contenant des calques, ses States contiennent également ceux des Elements calques. Comme chaque State est un arbre, il est possible d'en extraire des sous-états. Cet enchevêtrement de States est représenté dans la figure 2.3

Le but de ces trois notions est de conserver une trace de l'évolution des données. La notion d'Element rend générique les données gérées, les ElementVersions apportent la pérennité de l'enregistrement de leur évolution tout en limitant les redondances, et les States recomposent les données en la notion intuitive d'état de l'Element pour accéder et réutiliser les données. Comme seuls les Elements modifiés ont de nouvelles ElementVersions, le State obtenu après une édition pointe vers ces nouvelles versions, et pour les Elements qui n'ont pas été modifiés il pointe vers les mêmes ElementVersions que le State pré-changement. Ces States n'ont pas besoin d'être enregistrés et peuvent être recalculés au besoin à partir des liens de parenté des ElementVersions mis à jour à l'aide de l'évolution des Elements. Cette évolution est décrite par des informations de chronologie des changements effectués, présentés dans la section 2.1.2. Cela permet de diminuer la quantité d'informations à stoquer. De plus, ESCI n'impose pas de stratégie de stoquage des données associées aux ElementVersions. L'approche la plus simple est d'enregistrer une copie des données correspondantes dans l'ElementVersion, mais il est possible de n'enregistrer qu'un différentiel par rapport aux données de l'ElementVersion précédente, ou d'enregistrer les données dans une base de données séparée et n'enregistrer dans l'ElementVersion que la clef correspondant aux données

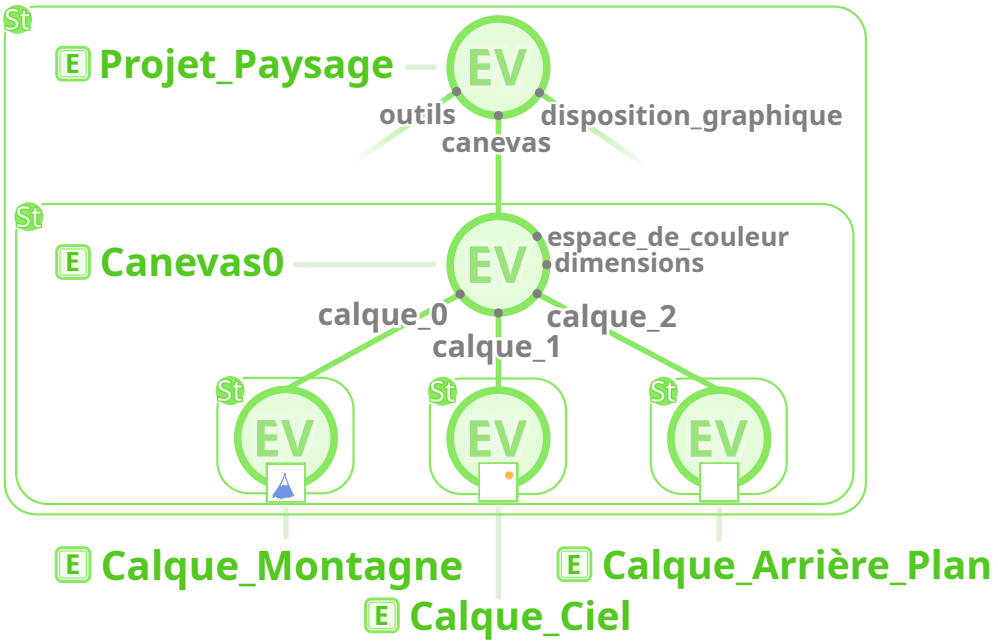

FIGURE 2.3 – Illustration des States. Les States sont représentés par un cadre contenant toutes les ElementVersions vers lesquelles il pointe, avec pour racine une ElementVersion d'un Element dont le State est un état.

correspondantes.

Pour résumer, nous avons jusqu'à présent vu les données enregistrées par l'historique relatives aux Elements édités. Les Elements ont des States par lesquels il sont passés au cours de l'édition, et chaque State est un arbre d'ElementVersions qui stockent les données constituant le State. La figure 2.4 résume schématiquement les relations entre ces composants.

Nous allons maintenant introduire la notion de chronologie et de changement d'état dans le modèle.

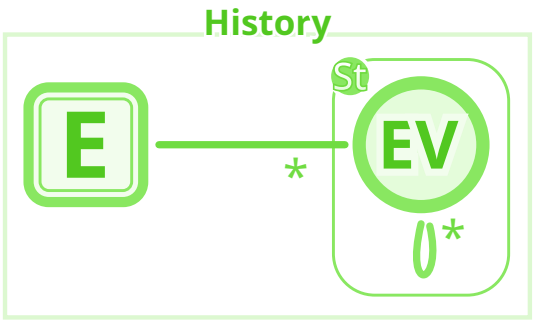

FIGURE 2.4 – Relations entre Element, State et ElementVersion. Un Element a plusieurs ElementVersions. Une ElementVersion contient des données d'état, ainsi que des liens vers des ElementVersions d'Elements enfant. Un State est composé d'un arbre d'ElementVersions.

Session et View

Toute édition est menée dans un contexte d'édition. Le projet à éditer est ouvert, édité, puis refermé. Je nomme cette période une session d'édition.

La Session (S) contient la séquence des exécutions de commandes d'une session d'édition. Ouvrir un projet avec un logiciel d'édition commencera une nouvelle Session. L'utilisateur effectue ensuite sa tâche d'édition, puis ferme le projet, fermant ainsi la Session. Lors de la création d'une Session, un identifiant unique lui est asssigné. Il est aussi possible de lui assigner un State provenant d'une autre Session comme état initial des données.

La Session est la frise chronologique des exécutions qui ont eu lieu. Au cours de l'édition, la vue (*View*) marque le présent dans cette frise, pointant vers les States courants des Elements.

La View (Vw) est le marqueur temporel du présent d'une Session, et représente donc les données actuelles que l'utilisateur voit lors de son édition. Elle est liée au State obtenu par la dernière exécution de commande de la Session. L'Element du State est alors la racine de la View. Ce State courant sera nommé l'état de la View. La View peut être déplacée vers n'importe quel State lors d'une exécution de commande de la Session, ce qui permet de revenir à un état généré par le passé et continuer son travail d'édition dessus. La création d'une Session génère une View, et si un State initial est assigné à la Session, la View y sera déplacée.

Comme la View n'est qu'un pointeur déplaçable vers un State, les seules données à enregistrer dans l'historique la concernant sont ses déplacements. Cela permet de retracer par la suite l'évolution du State courant présenté à l'utilisateur au cours de la Session. Pour enregistrer ses déplacements, un Element est utilisé. Lors de la création d'une Session, un Element représentant la View sera créé et associé à la nouvelle Session, et une nouvelle ElementVersion sera assignée à cet Element. Si un State initial est assigné à la Session, l'ElementVersion aura pour enfant l'ElementVersion racine du State donné. Sinon, l'ElementVersion n'aura pas d'enfant, ce qui représente une View vide. Il faudra alors déplacer la View vers un State existant ou créer un Element et une ElementVersion correspondante, puis déplacer la View dessus.

La relation de parenté entre ElementVersions est utilisée pour pointer vers un State. Déplacer la View vers un autre State revient alors à créer une nouvelle ElementVersion de l'Element la représentant et lui assigner l'ElementVersion racine du State comme enfant.

Toute exécution est ajoutée à la fin de la séquence d'exécutions enregistrée dans une Session. Ces exécutions modifient le State courant, et font ainsi progresser l'édition et avancer la View. La View est une vue dans l'historique d'édition, un présent, évoluant au cours d'une Session.

Les Sessions ordonnent les changements qui ont lieu. De plus, chaque utilisateur collaborant aura sa propre Session, ce qui permet d'enregistrer leur travail dans des séquences séparées, et ne les rend pas dépendant les uns des autres. Un système de synchronisation des Sessions ainsi que l'utilisation d'autres méthodes comme OT et CRDT seront sera présenté dans la section 2.1.2 dédiée à la collaboration. L'apport premier de la notion de Session à la structure de l'historique est l'ajout de la notion de chronologie des opération ayant lieu. Il est ainsi possible de parcourir l'historique de manière chronologique et de connaître l'ordre d'exécution, notamment pour rejouer une session d'édition ou ré-exécuter des séquences d'opérations.

La View introduit la notion d'état présent et d'évolution de celui-ci. Cette notion est centrale pour les contextes d'édition, comme elle représente les données actuellement éditées, affichées à l'utilisateur par l'interface utilisateur et manipulées par les commandes du contexte qui sont définies relativement à cet état courant. Il aurait été possible de ne pas avoir de notion de présent et de laisser les opérations d'édition parcourir et manipuler l'historique sous la forme d'un graphe, mais les commandes et les interfaces utilisateur actuelles sont développées autour de la notion de document, qui a un état que l'on modifie. Ce repère d'état présent est fourni par la View. ESCI n'empêche pas la modifications d'états qui ne font pas partie de l'état courant, ce qui permet d'explorer

d'autres manières d'éditer des données qui prennent en compte l'historique entier et non juste l'état présent. Ces opérations d'édition hors état courant restent cependant optionnelles et sont là pour permettre d'explorer différentes manières d'implémenter des fonctionnalités. Un exemple simple est l'annulation sélective par reconstruction. Il est possible de déplacer la View vers un State passé et d'y ré-exécuter les Commands à conserver. Une autre possibilité serait de ré-exécuter les Commands sur l'ancien State puis de déplacer la View sur le State résultant. À la manière d'une tâche d'arrière-plan, les changements ont lieu hors View. L'intérêt et le potentiel des deux approches doit encore être évalué. Pendant une Session, l'état courant des Elements est désigné par la View. Pour modifier l'état de ces Elements, le logiciel exécute une Command.

Command et Input

 **Une Command (C)** a pour but de manipuler l'état d'Elements. Pour un logiciel de peinture numérique, on peut avoir des Commands de tracé de trait, de création et suppression de calques, de redimensionnement du canevas, de changement de couleur du pinceau, ... Toute Command a un type décrivant l'opération d'édition qu'elle effectue, les types d'Arguments passés en entrée pour l'exécution et des informations d'héritage pour exprimer les équivalences entre Commands. Pour pouvoir exécuter une Command, il faut au préalable lui passer les Arguments à utiliser. Chaque Argument a un nom et une valeur associée. TracerCalque, une Command permettant de tracer un trait sur un calque, a par exemple besoin de connaître la trajectoire du trait en question (argument nommé tracé), le calque sur lequel dessiner ce trait (argument nommé calque), ainsi que d'autres informations de taille et d'opacité du tracé à effectuer (arguments nommés taille et opacité). Si l'utilisateur trace un trait sur sa tablette graphique par exemple, le logiciel va récupérer cette action contenant le trait effectué, et passe ce trait en tant qu'Argument à la Command créée au préalable. Le logiciel va également chercher les autres informations dans la View, telles que le calque sur lequel dessiner et les paramètres de pinceau, et passe également leur état en tant qu'Argument à la Command. Celle-ci est maintenant prête à être exécutée. Elle est alors passée à la View qui sert d'interface pour l'exécution, l'enregistrement des données générées telles les nouvelles ElementVersions dans l'historique, et la mise à jour du State courant vers lequel elle pointe.

Les Arguments peuvent soit venir de l'historique sous la forme de States ou d'opérations passées entre autres, soit de l'extérieur de l'historique comme le trait effectué, du texte collé depuis le presse papier, une date, la graine d'un générateur de nombres aléatoires,

... Ces données externes sont nommées Inputs.

Un Input (I) est un bloc de données externes à l'historique utilisé par une Command. Il est enregistré dans l'historique lors de l'exécution de cette Command, le rendant ainsi consultable et réutilisable. Comme les Elements et les Commands, les Inputs ont un type décrivant la nature des données qu'ils contiennent (Tracé, Date, Calque lorsqu'un calque est importé ou collé depuis un autre projet, ...) et des informations d'héritage. Les Inputs sont enregistrés sous la forme d'une **InputVersion** (IV) contenant ses données. Contrairement aux ElementVersions, les InputVersions n'ont pas d'enfants et sont des blocs de données indépendants. Les Inputs sont volontairement monolithiques car ils ne font que conserver des données d'entrée. Pour modifier leur contenu, il faut créer un Element (ou un arbre d'Elements pour détailler sa structure) à partir des données de l'Input et l'éditer.

Les notions de Command et d'Input permettent de représenter les informations relatives aux opérations d'édition exécutées et aux données externes insérées dans le projet. Elles permettent au logiciel de communiquer à l'historique les opérations et les données d'entrée de manière générique. Avec les Elements, les Commands et les Inputs permettent au logiciel de décrire l'édition.

De la même manière que les données des Elements sont enregistrées dans des ElementVersions et celles des Inputs dans des InputVersions, les données relatives à l'exécution de Commands sont enregistrées dans des CommandVersions.

CommandVersion, Change et TRIP

Une CommandVersion (CV) enregistre les Arguments utilisés lors de l'exécution de la Command associée, et est reliée aux Versions que la Command a créées. Elle représente les effets de cette Command sur les données enregistrées par l'historique, et est enregistrée dans la Session dans laquelle elle a été exécutée. Si l'on applique au Calque_Ciel un coup de pinceau à l'aide de la Command TracerCalque, les données du calque changeront, et une nouvelle ElementVersion sera créée pour enregistrer ces nouvelles données, comme on peut le voir dans la figure 2.5.

Les données de l'ElementVersion du Canevas0 n'ont pas été modifiées, donc aucune nouvelle ElementVersion n'est créée pour cet Element. Par contre, son State n'est plus le même, car un State enfant a changé. Il en va de même pour le projet.

Dans le cas où une Command exécute à son tour d'autres Commands lors de son exécution, on obtient alors un arbre de CommandVersions. Chaque CommandVersion de

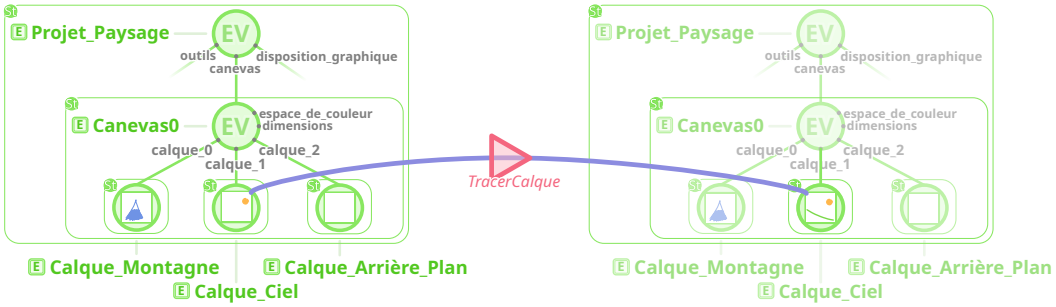

FIGURE 2.5 – Conséquences de l'exécution d'une Command. Le triangle rouge représente la CommandVersion créée pour cette Command de type TracerCalque. Une nouvelle ElementVersion est créée pour Calque_Ciel. Le State de ce calque en est de ce fait changé, ainsi que le State de tout Element parent. Seul les State des calques voisins n'ont pas changé. La ligne bleue représente le passage d'une ElementVersion à une autre dû à cette exécution de Command.

cette arbre correspond à une Command exécutée. On appelle les Commands exécutant d'autres Commands des *commandes composées*. Une Command implémentant l'annulation sélective par reconstruction devra après restitution d'un état passé ré-exécuter les Commands à conserver. Un arbre de CommandVersions représente l'ensemble des changements qui ont eu lieu suite à l'exécution de la Command à la racine de l'arbre. Ces arbres sont nommés des Changes (changements).

Un Change (Ch) représente l'ensemble des informations d'exécution et des effets des opérations qui ont eu lieu lors de l'exécution d'une Command. Un Change est un pointeur vers un arbre de CommandVersions.

Les CommandVersions enregistrent de manière pérenne les données d'exécution, et permettent de ré-exécuter les Commands à l'origine de la version. Les Changes permettent de parcourir ces données. Le logiciel a à sa disposition les notions de State et de Change pour parcourir l'historique et ré-utiliser les données.

Au cours de son exécution, une Command va manipuler des versions à des fins différentes. Certaines Versions sont lues, d'autres créées. Pour exprimer ces différents usages, différents types de liens sont utilisés pour relier la CommandVersion de cette Command avec les Versions utilisées. Ces liens sont nommés TRIP, acronyme pour *Targets*, *Results*, *Inputs*, *Parameters*.

Les liens TRIP sont les liens de causalité entre versions, reliant les ElementVersions et InputVersions utilisées ou créées par une commande à sa CommandVersion. On distingue les versions lues des données écrites, permettant ainsi d'avoir un flux de transfert d'informations qui a eu lieu lors de l'exécution. La figure 2.6 représente de manière chronologique l'exemple du tracé de trait des figures ci-dessus. Lors d'une Session, l'Element Calque_Ciel est dans un State correspondant à l'ElementVersion 0. La CommandVersion de la Command TracerCalque est liée au tracé effectué en entrée (Tracé dans la figure 2.6), à l'ElementVersion précédente du Calque_Ciel, ainsi qu'à la nouvelle ElementVersion résultant de l'exécution de la Command. Les deux Versions du calque sont reliées par la CommandVersion qui a effectué ce changement. Comme Causality [NC14], ESCI distingue quatre types de liens : les *Targets*, *Results*, *Inputs* et *Parameters*.

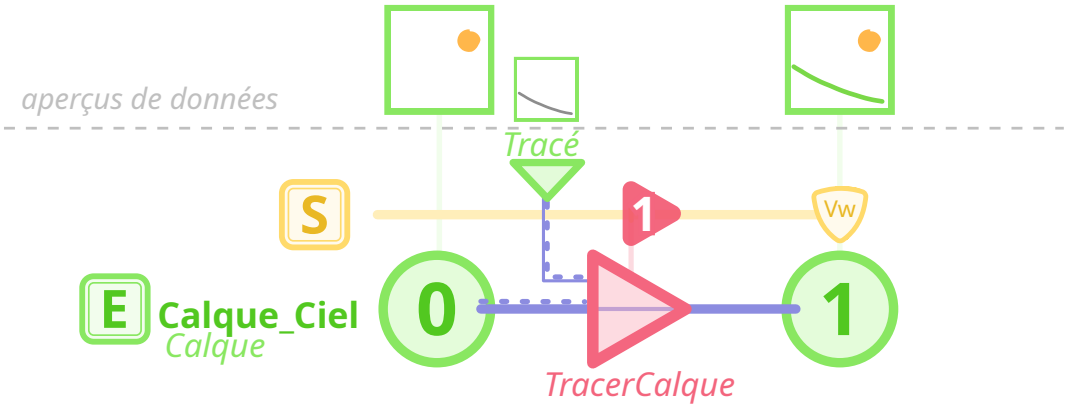

Figure 2.6 – Diagramme chronologique du tracé d'un trait sur le Calque_Ciel. Les liens TRIP sont bleus. Les Parameters sont les lignes en pointillés. L'Input est la ligne fine entre Input et la CommandVersion qui l'a introduit dans l'historique. La ligne pleine entre une ElementVersion et une CommandVersion est un Target, et cette même ligne passant de la CommandVersion à une ElementVersion est un Result.

Tout d'abord, il y a les liens de lecture, les Parameters. Ce sont les liens reliant les Versions lues à la CommandVersion qui les a lus. La Command a lu le tracé ainsi que l'ElementVersion 0 du calque. Ensuite, il y a l'Input, le lien entre Input et la CommandVersion qui l'a introduit dans l'historique. Le lien Target entre ElementVersion 0 et la CommandVersion représente le fait que la Command a changé l'état du calque. Le lien Result relie la CommandVersion à l'ElementVersion 1 qu'elle a créée. Le couple Target–Result marque les insertions, modifications et suppressions des ElementVersions

dans l'arbre d'Elements. Si la CommandVersion supprime le calque du canevas, il n'y aura qu'un lien Target. À l'inverse, si la CommandVersion introduit le calque dans le canevas, il n'y aura qu'un lien Result. On retrouve la ligne Target–Result dans les différentes illustrations de liens de causalité du chapitre 1, reliant entre eux States et Changes qui alternent au cours de l'édition.

Le TRIP permet d'apporter quelques précisions dans la description des liens de causalité :

— **Liens inter TRIP.** Les liens d'entrée (Target et Parameter) peuvent être reliés à des liens de sortie (Result). Dans l'illustration 2.6, le lien Target est lié au lien Result, ce qui permet de représenter le passage de l'ElementVersion 0 à l'ElementVersion 1 pour Calque_Ciel. Cela permet d'exprimer le flux d'informations au sein de la Command. On peut ainsi relier les Versions d'entrée ayant contribué à une Version de sortie.

— **Lien TRIP–origine.** Il est possible de relier un lien TRIP avec un Argument, comme par exemple le lien TRIP entre la Version initiale du calque Calque_Ciel et le nom de l'Argument par lequel il a été donné à la Command (ici, calque). On peut ainsi étudier ce que la Command fait des données passées en Arguments. Dans le cas où la Version n'a pas été passée en Argument mais a été recherchée dans l'historique par la Command lors de son exécution, le chemin (*Path*) suivi pour la retrouver sera enregistré.

— **Précision d'une *Region* concernée.** Le TRIP permet de préciser ce que l'on a utilisé dans une Version, à un niveau plus précis que la Version entière. Il est possible de spécifier un Attribute en particulier, comme la composante rouge d'une couleur, ou la position d'un objet dans une scène 3D. Il est également possible de spécifier une région (Region) spécifique, comme par exemple si l'on copie une section du calque Calque_Ciel vers Calque_Montagne. Une partie seulement du calque Calque_Ciel aura été lue, et une partie du calque Calque_Montagne aura été modifiée. ESCI laisse libre la manière de décrire ces Regions, permettant ainsi d'utiliser des langages de description existants [YM19]. On peut ainsi décrire plus finement ce qu'il s'est passé lors de l'exécution d'une Command.

Les liens TRIP et la gestion de régions apportent des informations relationnelles à l'historique. Ils relient les versions enregistrées dans l'historique entre elles et permettent ainsi de le parcourir en fonction des Changes qui ont eu lieu. Ils permettent aussi d'identifier des dépendances entre CommandVersions, ce qui est utile pour la gestion des paradoxes et des conflits.

En suivant les Sessions et les liens TRIP, il est possible de reconstruire tout State.

Le point de départ peut être un autre State déjà calculé ou un début de Session vide. À partir de là, on suit les liens TRIP de chaque Change de la Session jusqu'à atteindre le State désiré. Les liens Result informent de la création d'ElementVersions, ce qui permet de progressivement reconstruire le State.

Dans le cas où l'on a un State calculé qui précède le State désiré, on peut appliquer la même marche à partir de ce State. S'il succède le State à calculer, on n'a qu'à parcourir les liens TRIP dans le sens inverse, où l'on s'intéressera aux ElementVersions ciblées par une modification (lien Target).

Collaboration

Deux utilisateurs menant une tâche d'édition dans le même document auront deux Sessions différentes. Comme ils ont des Sessions différentes, leur View n'est pas la même, et ils avancent de manière indépendante. L'historique est cependant partagé, donc ils ont accès aux mêmes données, pouvant notamment consulter les States générés par l'autre, et de les utiliser comme Arguments par exemple. Ce scénario ressemble au partage de données à l'aide d'un SCV entre deux utilisateurs. Il est possible de fusionner des données entre Sessions. Mais comme les Commands exécutées dans une Session n'impactent par défaut que sa View, elle ne fera pas avancer de manière synchronisée les différentes Sessions. Le comportement par défaut d'ESCI convient à la collaboration asynchrone. Ce comportement assure l'enregistrabilité des travaux de chacun sans avoir à se soucier de potentiels conflits. Plusieurs options de synchronisation sont possibles pour faire de l'édition collaborative en temps réel, chacune adaptée à des méthodes de synchronisation différentes.

La synchronisation de Sessions convient en particulier aux scénarios de collaboration où la séquentialité des changements est garantie, à l'aide de verrous par exemple. Pour ce faire, il faut spécifier aux Sessions en question la partie de l'état de la View à synchroniser. Les Sessions prendront alors en compte les Changes impactant la partie spécifiée générés dans les autres Sessions synchronisées, faisant avancer leur View. Si jamais un Change entre en conflit avec un autre, les Sessions se désynchronisent, ce qui permet aux utilisateur de rester désynchronisé ou de résoudre le conflit et de les resynchroniser.

L'utilisation de méthodes OT ou CRDT est également possible nativement. Dans le cas de l'OT, la Command à l'origine du Change est transformée et ré-exécutée. L'outil de transformation bénéficie de l'ensemble des données de l'historique pour transformer la Command en fonction des Changes déjà effectués. Les informations de dépendance TRIP lui seront particulièrement utiles. L'historique possédant les Sessions de chaque

utilisateur, il lui est aussi possible de lire les informations de l'exécution originale. Dans le cas du CRDT, il n'y a qu'à ré-exécuter la Command dans la Session de chaque utilisateur.

Là où la redondance d'exécutions de Commands entre Sessions est limitée au travers du système de synchronisation de Sessions, dans le cas de l'OT et du CRDT les exécutions ne sont pas identiques entre Sessions, empêchant cette allègement. Ne pas enregistrer les Sessions des autres utilisateurs dans son historique est possible, mais prive de la possibilité de parcourir les données générées par les autres.

Il y a potentiellement d'autres manières d'utiliser ESCI dans un contexte collaboratif, et sera l'objet de potentiels projets futurs.

Résumé de la structure d'ESCI

Nous avons maintenant vu tous les composants principaux du modèle ESCI. Le schéma 2.7 résume ces différents composants ainsi que leurs interactions.

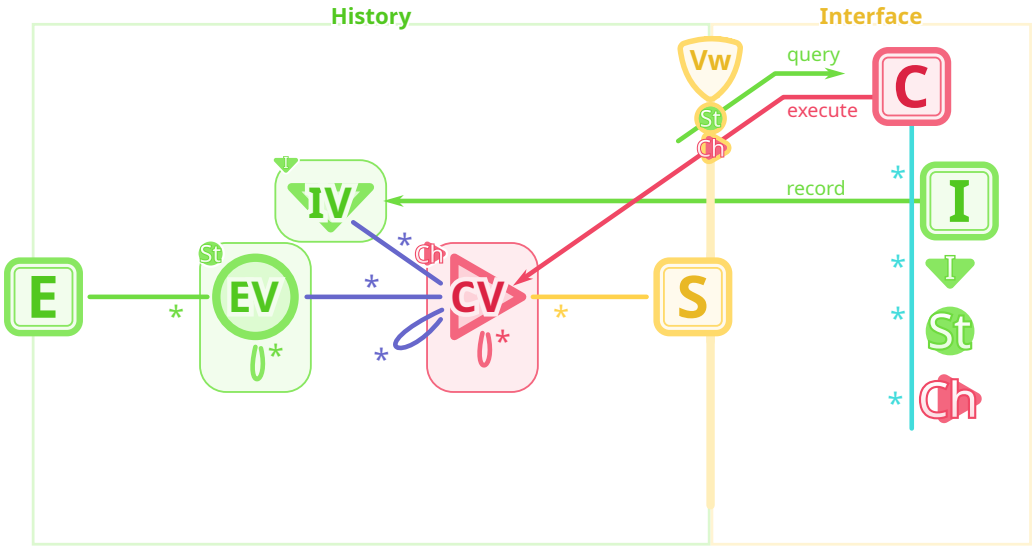

Figure 2.7 – schéma du modèle ESCI

Ils sont regroupés en deux parties : l'historique et l'interface d'ESCI. L'historique contient des Elements qui représentent les objets édités. Les Elements ont un ElementType et une collection d'ElementVersions, représentant ces Elements à différents moments de l'édition. Les States sont un arbre d'ElementVersions, et les ElementVersions contiennent les données.

L'historique possède également des Sessions qui enregistrent les opérations l'édition de ces Elements. Ces Sessions sont des collections de CommandVersions représentant les modifications des States des Elements. Chaque Change est un arbre de CommandVersions, et les CommandVersions contiennent les informations d'exécution de la Command en question.

L'historique possède également tous les Inputs utilisés par les CommandVersions. Les liens TRIP relient les CommandVersions aux autres Versions que la Command associée utilise lors de l'exécution.

Les Elements et les Sessions sont les conteneurs pour les informations d'état et de changement respectivement. Les versions enregistrent de manière pérenne les données, et les States et les Changes permettent de parcourir ces informations en les regroupant de manière cohérente.

Côté interface, les Commands sont exécutées par la View. Chaque Session possède une View qui permet au logiciel d'interagir avec l'historique. Elle pointe vers l'état courant des Elements, et permet de consulter les autres States, Changes et Inputs enregistrés. Les States, Changes et Inputs enregistrés sont ainsi nommés collectivement *Records* (enregistrements).

La Command possède un CommandType et des Arguments, qui peuvent être des Records ou un nouvel Input, qui sera enregistré dans l'historique. Une Command est exécutée par la View, enregistrée comme CommandVersion, et toutes les nouvelles Versions ainsi générées sont également enregistrées. Il en va de même pour les sous Commands que la Command a lancées au cours de son exécution.

Les Commands et les Inputs sont les conteneurs d'opérations et de données externes qui vont être intégrées à l'historique. Nous allons maintenant détailler un peu plus ce processus d'exécution de Command et d'enregistrement des Records en résultant.

2.1.3 Enregistrement et réutilisation de la trace d'édition

Enregistrement des Records

Une Command a accès lors de son exécution à ses Arguments ainsi qu'aux Records de l'historique. Pour effectuer une modification de l'état d'un Element, elle va créer de nouvelles ElementVersions. La figure 2.8 reprend l'illustration des conséquences de l'exécution d'une Command sur l'état du projet de la section précédente.

Dans ce scénario, la Command TracerCalque modifie le contenu d'un calque et crée donc une nouvelle ElementVersion pour représenter ces nouvelles données. Elle peut ainsi modifier l'arbre d'ElementVersions que sont les States en créant de nouvelles

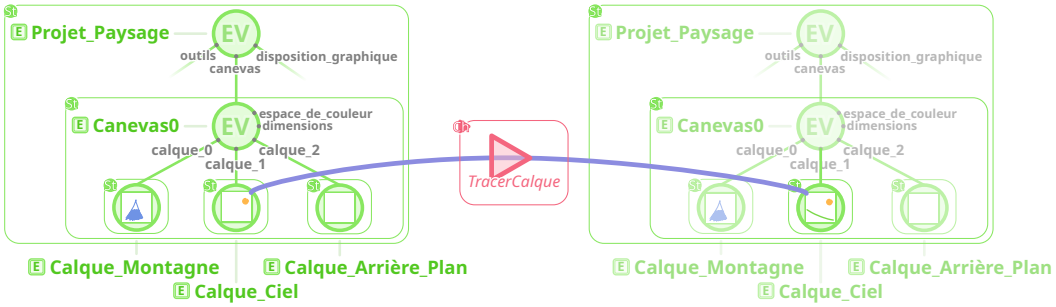

FIGURE 2.8 – Conséquences de l'exécution d'une Command. Le Change généré par la Command est représenté par le rectangle rouge, contenant toutes les CommandVersions la composant (ici, il y en a une).

ElementVersions qui se substitueront aux anciennes. Des enfants peuvent être ajoutés ou supprimés de cette ElementVersion. Ainsi, si un calque est supprimé, on créera une nouvelle ElementVersion du calque, de laquelle on supprimera l'ElementVersion en question. Il est également possible de recharger un State passé d'un calque en donnant à l'ElementVersion du canevas ce State comme enfant. Toutes ces ElementVersions seront reliées à la CommandVersion représentant cette exécution par des liens TRIP. La Command relie également à sa CommandVersion les Versions lues et les nouveaux Inputs enregistrés. Ces changements seront enregistrés dans l'historique une fois l'exécution terminée. Ces nouveaux arbre d'ElementVersions correspondent à de nouveaux States et la CommandVersion au nouveau Change. Lors de cette exécution, la Command peut exécuter d'autres Commands en préparant ses Arguments et la donnant à la View pour exécution. La CommandVersion en résultant sera liée en tant qu'enfant à celle de la Command parent et fera ainsi partie de son Change.

Il est possible de changer n'importe quel State enregistré dans l'historique, qu'il fasse partie de la View actuelle ou non. Ces changements font généralement avancer l'état de la View, comme dans toute scénario d'édition classique. Il est cependant possible de changer un State, ne faisant pas partie de la View, simplement pour créer un nouveau State, sans que la View n'en soit impactée. En d'autres termes, l'état de la View ne changera pas, ce qui signifie que la Command n'aura pas d'effet perceptible par l'utilisateur, les données qu'il voit n'ayant pas changées. Ces changements détachés de la View permettent par exemple de créer des States qu'il sera possible de réutiliser plus tard, même s'ils ne sont pas utilisés au moment de leur création.

Réutilisation des Records

Une fois enregistrés dans l'historique, ces Records peuvent être réutilisés.

On peut restituer l'état passé d'un Element. Cela permet de repartir de ces States et de travailler l'Element autrement, notamment dans le cadre de l'explorations d'idées ou d'erreurs effectuées. On peut travailler un Element d'une manière, puis revenir et le travailler autrement. Il est en suite possible de consulter les différentes alternatives et de choisir ce que l'on préfère en naviguant dans l'historique.

Les Inputs peuvent également être réutilisés comme les States en tant qu'Argument pour une nouvelle Command.

Pour les Changes, cela se traduit par la possibilité de ré-exécuter la Command associée. On peut par exemple répéter l'exécution de la dernière Command, ou de n'importe quelle séquence de Commands passées. Pour cela, on crée une Command à partir d'un Change consulté, et met à jour certains de ses Arguments pour adapter les entrées de la Command à l'état de la View. Par exemple, si l'on reprend une Command TracerCalque, c'est sur le State courant du calque ciblé que l'on veut effectuer le tracé. Il faut donc mettre à jour l'Argument calque au State courant du calque. Cette mise à jour des Records est conditionnée par l'existence d'un équivalent dans l'état de la View. Si le calque n'existe plus, il ne sera pas possible de le modifier.

Chaque Argument peut être enregistré comme étant soit dynamique, soit statique. Un Argument dynamique sera mis à jour lorsque l'on voudra exécuter à nouveau la Command l'utilisant. Un Argument statique gardera le Record donné initialement. Un exemple courant d'Argument dynamique sont les States que l'on veut modifier, ce qui permet lors de la réutilisation de la Command de modifier le State courant et non celui utilisé initialement.

La mise à jour des Arguments dynamiques est faite par la View juste avant d'exécuter la Command. Si cette mise à jour échoue, cela signifie qu'une donnée de l'historique requise n'est pas trouvable dans l'état courant. Pour marquer cela, un *Paradox* sera levé.

2.1.4 Paradoxes

Un Paradox représente une différence dans le fil de ré-exécution par rapport à l'exécution initiale, au niveau de requêtes de données dans l'historique. Chercher un State d'un Element dans l'état courant de la View peut échouer, celle-ci évoluant avec le temps. De plus, si l'utilisateur supprime de l'historique des States et Changes utilisés par d'autres Commands, ces dernières ne pourront être ré-exécutées en l'état et lèveront un Paradox lors d'une tentative de ré-exécution. Toute requête peut alors échouer.

La différence d'exécution peut être bloquante, comme l'absence de la View d'un Element à utiliser, ou non bloquante, comme l'existance de plusieurs copies du même Element utilisé initialement. Dans le premier cas on parle d'incohérence, et il sera nécessaire de la corriger pour avancer, car les données requises de l'historique n'ont pas été trouvées. Dans le second on parle d'ambiguïté, où plusieurs options s'offrent à nous. Si l'une d'elles peut être considérée comme le choix par défaut, l'exécution peut se poursuivre sans avoir à résoudre l'ambiguïté.

Lorsque l'on ré-exécute une Command, on met à jour chaque Argument dynamique en suivant le chemin donné. Ce chemin, nommé Path, permet de décrire un Record relativement à un autre ou à la View. Il est ainsi possible de désigner le State courant du canevas en donnant la liste des Attributes depuis la racine de la View par exemple. Pour chaque Argument dynamique pour lequel le chemin ne peut être résolu, il est précisé le Record se trouvant à la position attendue si il y en a un, ainsi que le ou les Records courant dans la View qui peuvent être utilisés à sa place. On a alors une liste de candidats pour la mise à jour. Cette liste est retournée au logiciel, pour qu'il décide de la manière de résoudre ce Paradox. Le système renverra à la View une Command de résolution d'erreur, qui modifiera les Arguments de la Command à l'origine du Paradox pour le résoudre, après quoi cette Command sera exécutée à nouveau.

Outre la mise à jour d'Arguments, des Paradoxes peuvent être levés lors du parcours de l'historique par une Command ré-exécutée.

La seule différence est qu'ils sont générés au cours de l'exécution de la Command et non avant.

Toute Command ayant à gérer une inconsistance lors de son exécution, à la suite de l'échec d'un parcours d'historique ou de l'exécution d'une Command paradoxale, poura soit résoudre le Paradox en résultant en choisissant des données (un Record ou un nouvel Input) à utiliser, soit s'interrompre et faire remonter le Paradox, devennant elle aussi paradoxale. L'interruption aura pour effet de restaurer la View à l'état précédent l'exécution, celle-ci ayant été abandonnée. Il sera alors possible de corriger les Paradoxes et de ré-exécuter la Command. Il est possible d'enregistrer les exécutions paradoxales, inconsistantes ou ambiguës, dans l'historique. Dans le cas d'inconsistances, comme la Command n'a pas pu aboutir, seuls les States qu'elle a pu génerer pré abandon seront parcourables, et elle ne fera pas avancer la View, la laissant au State pré-exécution.

2.1.5 Édition de l'historique

Le modèle ESCI permet au logiciel d'éditer les données de l'historique. Comme le mentionne Dix[Dix91], l'historique enregistre l'édition des données du logiciel. L'historique contient cependant lui même des données, et fait partie du logiciel. La question de l'éditabilité de ces données et de l'enregistrement de cette édition se pose alors. Le logiciel et l'utilisateur doivent conserver le contrôle sur les données contenues dans l'historique et doivent pouvoir en supprimer des données inutiles ou confidentielles. L'utilisateur peut ainsi supprimer la saisie par inadvertance d'un mot ou une faute de frappe. Pour cela, un mode d'interaction à part est proposé, permettant cette méta édition. Cette édition n'est cependant pas enregistrée comme une opération dans l'historique, car le but est de pouvoir supprimer des données de l'historique. Enregistrer cette suppression et la rendre annulable irait à l'encontre du principe de cette fonction. Si vraiment il est nécessaire d'enregistrer ces opérations, un second historique peut être utilisé pour les enregistrer.

Mode contrôle

Le mode contrôle permet d'ajouter, de modifier et de supprimer des Types, Elements, Sessions et Versions, ce qui permet d'insérer, de modifier et de retirer de l'historique des Changes et States. Ce mode d'édition des données de l'historique doit cependant produire une structure cohérente, où la chronologie et la causalité sont respectées. Voici quatre règles résumant cette conservation.

1. **La chronologie des évènements doit être respectée.** Un Change ne peut utiliser des données qui n'existaient pas encore lors de son exécution en termes de chronologie. Cela permet de conserver la chronologie de l'historique.
2. **Toute Version doit avoir été créée.** Toute Version doit être liée à une CommandVersion l'ayant créée, à l'exception des Commands exécutées par l'utilisateur. Cela permet de conserver les liens de causalité de l'historique.
3. **Toute Version a une description associée.** Les ElementVersions sont reliées à un Element, qui est relié à un ElementType, les InputVersions sont reliés à un InputType, et les CommandVersions sont reliés à une Session et un CommandType. Cela permet de conserver les informations de description de la structure de l'historique.
4. **Les Commands doivent être rejouables et les States et Inputs reconstructibles.** Les données associées aux States et aux Inputs doivent être recalculables en

rejouant la Command de la CommandVersion à laquelle il est relié. Cela permet de conserver l'évolution des données suivies par l'historique.

Dans le cadre de l'édition collaborative, le logiciel prend la responsabilité de reporter ces modifications sur les autres historiques si besoin est. Le modèle permet aussi d'ajouter des données supplémentaires aux différentes structures, fonctionnalité comparable aux métadonnées d'un fichier, dont la modification n'est pas non plus enregistrée.

Métadonnées

Chaque Type, Element, Session, Version et Record peut se voir attribué des données supplémentaires qui ne sont pas nécessaires à ESCI mais qui peuvent être utiles pour le logiciel ou les utilisateurs. Chaque structure peut se voir attribuer des métadonnées de trois types :

— **Un nom.** Il est possible d'attribuer à une structure un nom. Cela peut être un nom défini par l'utilisateur et servir d'alias d'affichage à la place de l'identifiant utilisé dans l'historique.

— **Des tags.** Un tag est un marqueur représentant une caractéristique ou une qualité d'une structure permettant de créer des groupes de structures partageant ces traits. Il est possible de différencier les Changes relevant d'une correction de formatage ou orthographique de celles d'une saisie de nouvelles informations ou celles d'une réorganisation de la structure du document. Cela permet aussi de regrouper ces Changes par phase d'édition, comme les phases de croquis, de détail et d'harmonisation de l'exemple. On retrouve ainsi dans l'historique ce à quoi les opérations qu'il contient ont servi, et qui permet de conserver une sémantique plus proche de l'intention de l'utilisateur que du fonctionnement du logiciel. Différents évènements peuvent également être marqués, comme le Change précédent une compilation ou un enregistrement du projet, servant également de point de repère. Il est également possible de marquer chaque Change d'un nom d'utilisateur. Cela peut être utile dans le cas où plusieurs utilisateurs travaillent sur un même poste à tour de rôle, et donc dans la même Session, pour différencier les Changes effectués par chacun.

— **Des annotations.** Il s'agit de données définies par le logiciel attachées aux structures, à la manière des métadonnées de fichiers multimédia. Elles permettent d'ajouter des informations complémentaires aux structures, comme des données d'aperçu pour les States de calques.

2.1.6 Conservation des propriétés

Les concepts fondamentaux d'ESCI donnent au modèle les six propriétés à posséder pour unifier les fonctionnalités des historiques de commandes.

1. **Universalité des éléments et des commandes : Elements, Sessions, Commands, Inputs.** ESCI ne suppose rien quant à la structure des données éditées et les opérations d'édition disponibles. Cette structure est décrite par le logiciel au travers d'Elements, la décomposant en parties. Pour représenter les opérations applicables aux données, le logiciel définit des Commands. Les Elements et les Commands sont composables, permettant au logiciel de choisir le niveau de détail de la description de la structure et des opérations.

   La notion d'Inputs permet au logiciel de décrire à l'historique les données d'entrée. Le logiciel décrit à ESCI au travers des Elements, des Commands et des Inputs la nature de l'édition. Les Sessions représentent le travail d'édition effectué. Enfin, il est possible d'utiliser un langage de description de régions pour décrire une région précise d'un Element.

2. **Pérennité : ElementVersions, CommandVersions, InputVersions.** Les données associées aux Elements évoluent au cours de l'édition par l'exécution de Commands. L'exécution de Commands ne fait qu'ajouter des données à l'historique. À tout instant on associe aux Elements de la structure une ElementVersion enregistrée dans l'historique. L'exécution d'une Command génère de nouvelles ElementVersions pour les Elements édités. Chaque exécution est représentée par une CommandVersion. Ces CommandVersions sont enregistrées séquentiellement dans une Session. Elles sont également reliées par des liens de dépendance TRIP aux autres Versions.

   La structure permet d'enregistrer des exécutions de Commands, mêmes paradoxales ou conflictuelles.

3. **Contrôle : Mode contrôle, Métadonnées.** Il est possible de supprimer des données jugées inutiles, et de marquer celles utiles à l'aide de métadonnées.

4. **Accessibilité / organisation des données : States, Changes, Sessions, TRIP.** Les Changes relient les données de l'historique par chronologie et dépendance grâce aux Sessions et aux liens TRIP. L'historique est parcourable à l'aide de ces liens.

5. **Altérabilité : Records.** Le logiciel peut définir des Commands faisant usage d'anciens States, ré-exécuter une Command à partir des Arguments d'un Change passé ou encore passer un ancien Input en Argument à une Command entre autres.

6. **Cohérence : tous les composants.** ESCI assure la cohérence de sa structure en utilisant le principe de transactions. Chaque Change généré par une exécution doit être valide pour être enregistré. Un Change valide contient tous les nouveaux Types, Elements, Sessions et Versions créés au cours de l'exécution, reliés correctement par des liens de parenté et TRIP, avec si besoin est des informations sur les Paradoxes rencontrés et si l'exécution a échoué ou non. ESCI possède des mécanismes de détection de paradoxes et de conflits, et laisse libre le choix de la stratégie de gestion de ceux-ci.

2.2 L'édition avec ESCI

2.2.1 Exemple d'édition

Nous allons maintenant reprendre les différentes notions présentées et les illustrer dans le contexte du déroulement d'une Session d'édition. Tout d'abord, nous allons reprendre notre exemple de tracé de trait sur le Calque_Ciel, et représenter les étapes de préparation et les conséquences de l'exécution d'une Command, ici TracerCalque. La figure 2.9 représente l'historique avant l'exécution d'une Command (1), la préparation de la Command (2), son exécution (3) et l'historique après exécution (4).

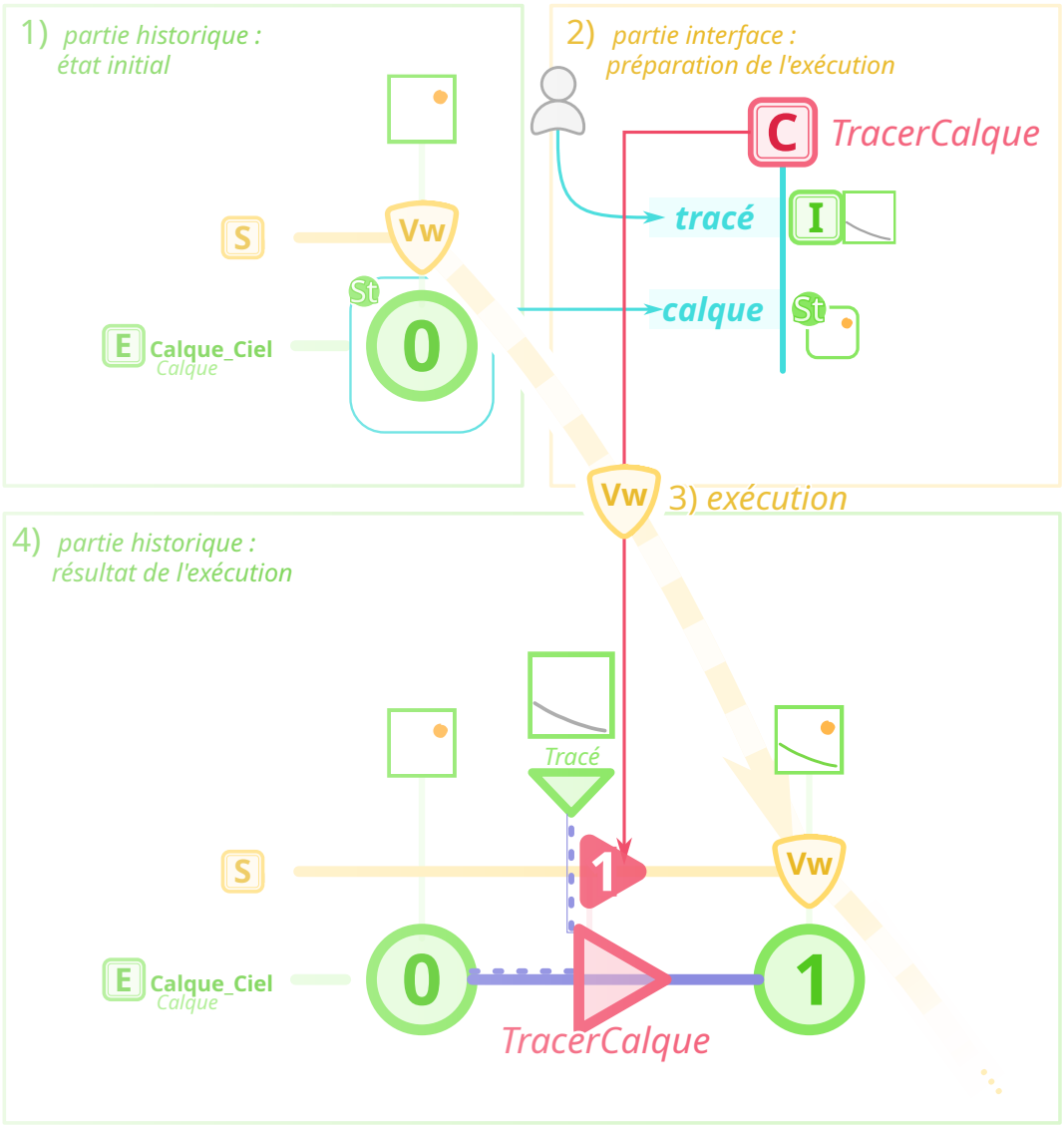

FIGURE 2.9 – Cycle d'exécution de Commands.
1) L'encadré représente l'état de l'historique avant l'exécution de la Command, montrant la Session et l'Element Calque_Ciel.
2) Le State de Calque_Ciel, constitué de son ElementVersion 0, est ensuite utilisé comme Argument de la Command TracerCalque. La Command prend également en Argument le trait effectué par l'utilisateur, en tant que nouvel Input.
3) Elle est alors prête à être exécutée, et est passée à la View pour qu'elle l'exécute.
4) Il en résulte l'enregistrement d'une nouvelle CommandVersion, InputVersion et ElementVersion dans l'historique, représentant le Change généré par la Command, le nouvel Input, et le nouveau State de l'Element Calque_Ciel.
Toutes ces Versions sont liées par liens TRIP représentant ce que la Command en a fait. La View se trouve maintenant à ce nouveau State, qui est le State courant de Calque_Ciel. Ces données sont à présent disponibles pour une nouvelle exécution, et ainsi de suite.

La Command fait évoluer l'état courant de la View au cours d'une Session d'édition, et l'historique enregistre ces données d'états et de changement. Si l'on représente un extrait de Session plus complet, on pourra y voir les liens entre Session, Elements, Commands et Inputs. La figure 2.10 représente une Session d'édition lors de laquelle le dessinateur David change la couleur active et dessine un soleil.

L'axe horizontal représente l'évolution des Elements au cours de la Session. La ligne d'un Element contient les ElementVersions correspondantes créées, ainsi que les CommandVersions qui l'ont impacté. En prenant également en compte les Elements enfants, on peut ainsi reconstruire ses States. Par exemple, le State de Canevas0 après le premier trait effectué (le Change numéro 2), est l'arbre d'ElementVersions composé de son ElementVersion 0, L'ElementVersion 1 de son enfant Calque_Montagne et l'ElementVersion 0 de son enfant Calque_Ciel. Même si aucun changement n'a affecté les données contenues par l'ElementVersion du canevas, tel que le titre ou ses dimensions, l'un de ses enfants a subi des changements. Canevas0 a donc changé de State car lui ou l'un de ses enfants (ou récursivement leurs enfants) a changé d'ElementVersion. La reconstitution du State d'un Element se fait ainsi en récupérant la dernière ElementVersion introduite pour chacun des enfants de l'Element en question et de lui même dans le contexte d'une Session. On remarque que les liens TRIP relient toutes les Versions entre elles, alternant entre ElementVersion et CommandVersion, et reliant le InputVersions aux CommandVersions. Le lien Target lie par exemple l'ElementVersion 0 du Calque_Montagne à la CommandVersion de type TracerCalque, qui est à son tour liée à l'ElementVersion 1 de Calque_Montagne qu'elle génère par un lien Result. Le lien Target est lié au lien Result pour ne former qu'une ligne reliant les deux ElementVersions, passant par la CommandVersion, représentant ainsi l'évolution de l'Element représenté. Les liens Parameter de lecture y figurent également, représentant les lectures de la part de la Command exécutée, ainsi que les liens Input marquant l'entrée de données externes dans l'historique.

L'utilisateur décide maintenant de restituer le State du Canevas0 à son état initial, sans soleil. L'effet de la restitution est représenté par la figure 2.11.

Il aurait pu ne restituer que le State du Calque_Montagne, ou de tout le Projet_Paysage, mais il a fait le choix de manipuler le Canevas0. La restitution génère une nouvelle ElementVersion pour le parent, ici Projet_Paysage, étant dans un nouvel état. Cette restitution romp la linéarité de l'évolution de ses enfants ; l'état du canevas n'est pas l'ensemble des ElementVersions courantes de lui et de ses enfants, mais des ElementVersions composant le State restitué. Il y a retour vers un State passé, reprenant un State passé de l'Element en question. L'utilisateur dessine ensuite un nuage. Il y a alors

"branchement" de l'évolution de l'Element, un State alternatif est créé. Il peut ensuite naviguer l'historique du Canevas0 pour consulter le State contenant le soleil, et décider de ce qu'il préfère, ou continuer à travailler sur ces deux alternatives, restituant celle qu'il souhaite travailler, puis décider qu'il préfère le soleil et continuer à travailler à partir de cette alternative. Il peut aussi combiner les deux en utilisant une Command qui prendra en Argument des données de l'alternative avec nuages et les ajoutera au soleil.

Reprenons notre exemple avant la restitution, avec le soleil de dessiné, pour éviter de surcharger les illustrations. L'utilisateur réalise après avoir dessiné le soleil, non pas qu'il ne lui plaît pas et qu'il veut un nuage à la place, mais qu'il a commis l'erreur de dessiner sur le mauvais calque. Dès le début, son but était de dessiner le soleil sur le Calque_Ciel, et non le Calque_Montagne. Certes, dans notre cas le soleil fût obtenu en deux changements et le niveau de détail du tracé n'est pas trop important ici. Mais imaginons qu'il s'agit là d'un dessin par exemple photoréaliste, avec une heure de travail sur le soleil, pour réaliser au bout de cette heure qu'il se trouve sur le calque de la montagne, mélangé avec le reste, inséparable. Aucune restitution ne peut le sauver de la perte du travail qu'il a effectué sur le mauvais calque, car ce n'est pas un état passé qu'il souhaite réutiliser, mais les changements qu'il a effectué, pour pouvoir les appliquer au calque souhaité. Il va donc réutiliser les Commands enregistrées, et les appliquer au bon calque cette fois-ci, comme le montre la frise 2.12

Après avoir restitué l'état du Calque_Montagne précédent le dessin du soleil (il aurait pu restituer celui du canevas entier, comme montré dans l'illustration précédente, mais mettons qu'il a entre deux modifié d'autres calques, et qu'il souhaite garder ces modifications), il applique la séquence de Commands ayant modifié le Calque_Montagne sur le Calque_Ciel à l'aide de la commande composée RéExécuterSéquence, qui prend les CommandVersions et exécute à nouveau les Commands qu'elles représentent, en changeant de calque visé. Cette Command ne modifie pas l'état d'Elements elle même, d'où l'absence de liens TRIP vers des ElementVersions. Il a ainsi évité de perdre son travail, et d'avoir à isoler et couper des parties d'un calque vers l'autre à l'aide de filtres, et d'effectuer des corrections manuelles.

2.2.2 Navigation, visualisation de l'historique et historiques individuels

Le travail effectué par l'utilisateur modifie progressivement les Elements éditées. On a alors deux manières de représenter cette édition. On peut la représenter par la Session, qui est une séquence de Changes, ou par Element édité, qui est une collection de States.

Ces représentations sont illustrées dans la figure 2.13 reprenant le scénario du soleil et du nuage (figure 2.11), où l'utilisateur ne se serait pas trompé de calque.

La Session représente un historique primitif, là où les Elements en représentent des méta. Ce ne sont pas plusieurs historiques séparés, mais un seul vu de différentes manières.

Trois remarques au sujet de ces historiques individuels :

1. Les States associés à l'historique de la Session sont ceux de l'Element à la racine de la View. Il s'agit ici de Projet_Paysage, qui est resté à la racine pendant toute la Session, sans subir de restitution. Les deux historiques ont donc une correspondance forte dans leur évolution.
2. La seconde remarque porte sur la propagation de l'action de restitution. Le Canevas0 a subi une restitution à un State passé, ce qui a créé une branche dans la structure d'historique. Et comme le State de l'Element est constitué de celui de ses enfants, une restitution de State appliquée à un Element a ainsi un effet sur tous les States enfants. Le Calque_Ciel subi donc également une restitution. Le Calque_Montagne n'ayant pas évolué depuis le State de restitution, celle-ci n'aura pas d'effets sur lui.
3. La dernière remarque porte sur la localité des historiques. La couleur Couleur a subi trois changements, et possède donc trois States. Elle n'a pas été sujette à la restitution, qui s'est appliquée à un Element ne faisant pas partie de son sous arbre d'enfants. Les changements qui se ont été appliqués au Calque_Ciel ne la concernent pas non plus. Il s'agit de l'évolution de son State, donc de son ElementVersion et des States de ses enfants. Elle a donc un historique bien plus court que les autres, ne prennant en compte que ce qui la concerne. À contrario, Projet_Paysage, étant la racine de la View, est concerné par tous les Changes ayant eu lieu, d'où ses nombreux States.

Tout State d'Element a été généré par un Change de la Session, donc les Elements et les Sessions sont intimement liés. Cela permet de naviguer en fonction de ce qu'il s'est passé, où de ce que l'on a obtenu au cours de l'édition, et de passer d'une structure à l'autre. Cette dualité est représentée dans la figure 2.7. Element est d'un côté, Session de l'autre, avec de leur côté les States et Changes qu'ils contiennent, reliés entre eux au moyen du TRIP. Elements et Sessions sont complémentaires, étroitement liés et dépendant l'un de l'autre.

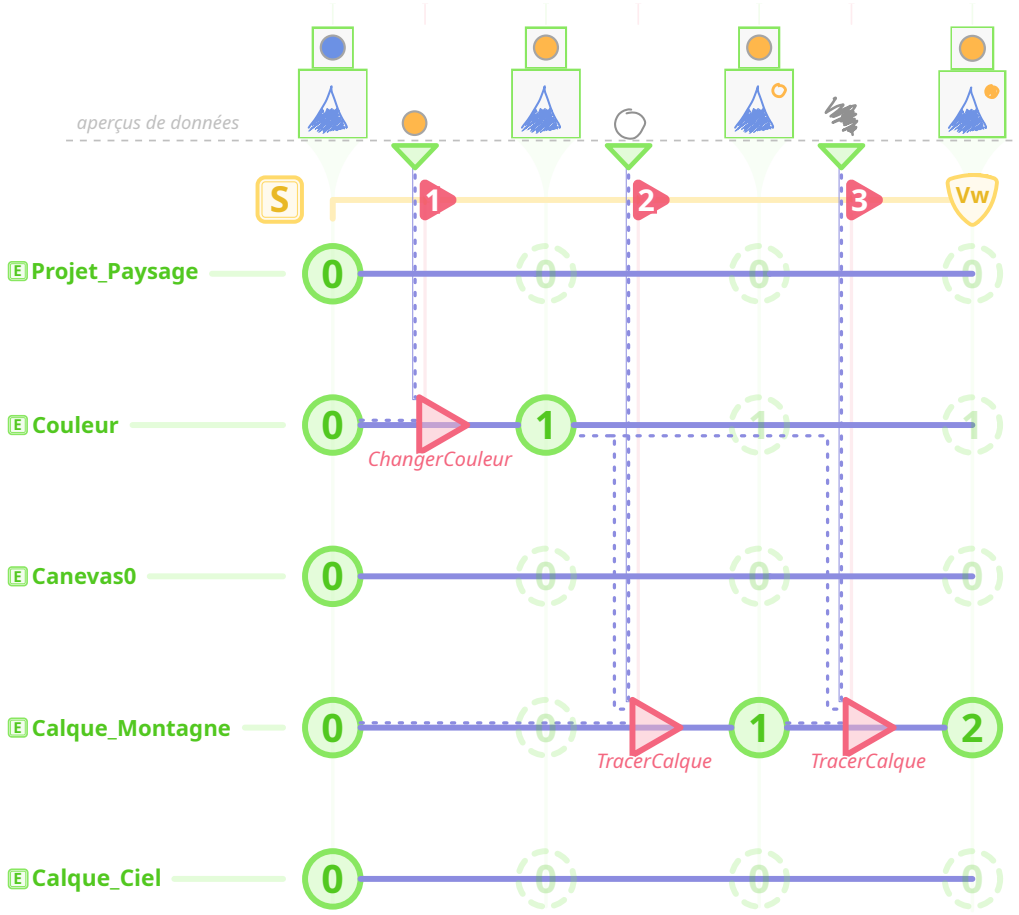

FIGURE 2.10 – Exemple de déroulement d'une Session d'édition. Les Elements utilisés dans cet exemple sont représentés à gauche. Des Commands ont été exécutées séquentiellement au cour d'une Session, générant des Changes représentés et numérotés sur la ligne de la Session. Ces Changes sont liés aux CommandVersions qui les composent. Les ElementVersions de chaque Element sont représentées sur une ligne par Element dans la frise, et liées aux CommandVersions les ayant générées. Les liens de parenté n'y sont pas représenté. Projet_Paysage est à la racine de la View, et possède Canevas0 et Couleur comme enfants. Canevas0 à son tour possède Calque_Montagne et Calque_Ciel comme enfants. Les états de Couleur et de Canevas0 sont illustrés en haut de la frise pour une meilleure visualisation. Le TRIP est représenté par les liens bleus entre Versions et la CommandVersion de la Command les ayant utilisées.

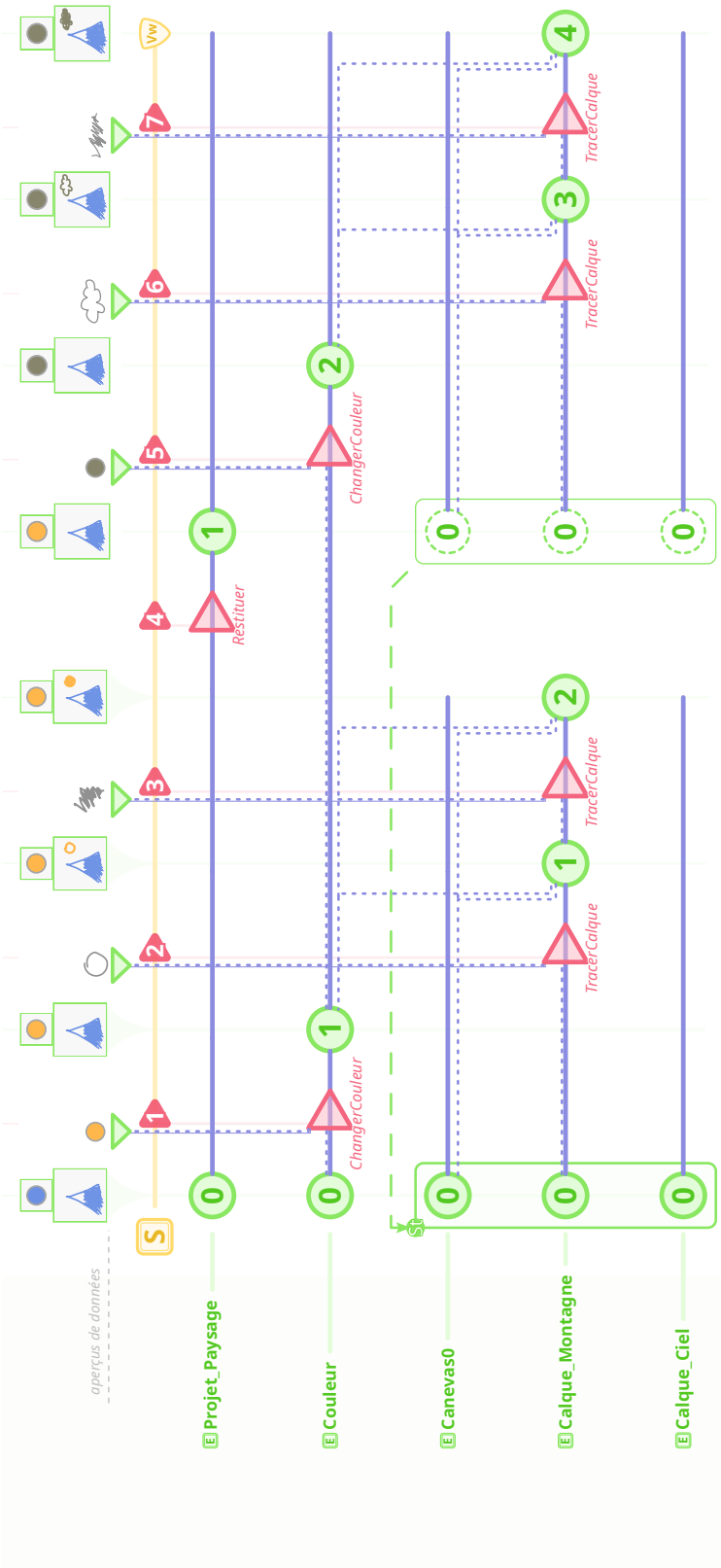

FIGURE 2.11 – Exemple de déroulement d'une Session d'édition avec restitution du State de l'Element Canevas0 à son State initial. Le State initial est représenté par l'encadré incluant toutes les ElementVersions qui le constituent. Celui-ci est restitué plus tard, restitution représentée par la flèche en pointillés et les ElementVersions en pointillés.

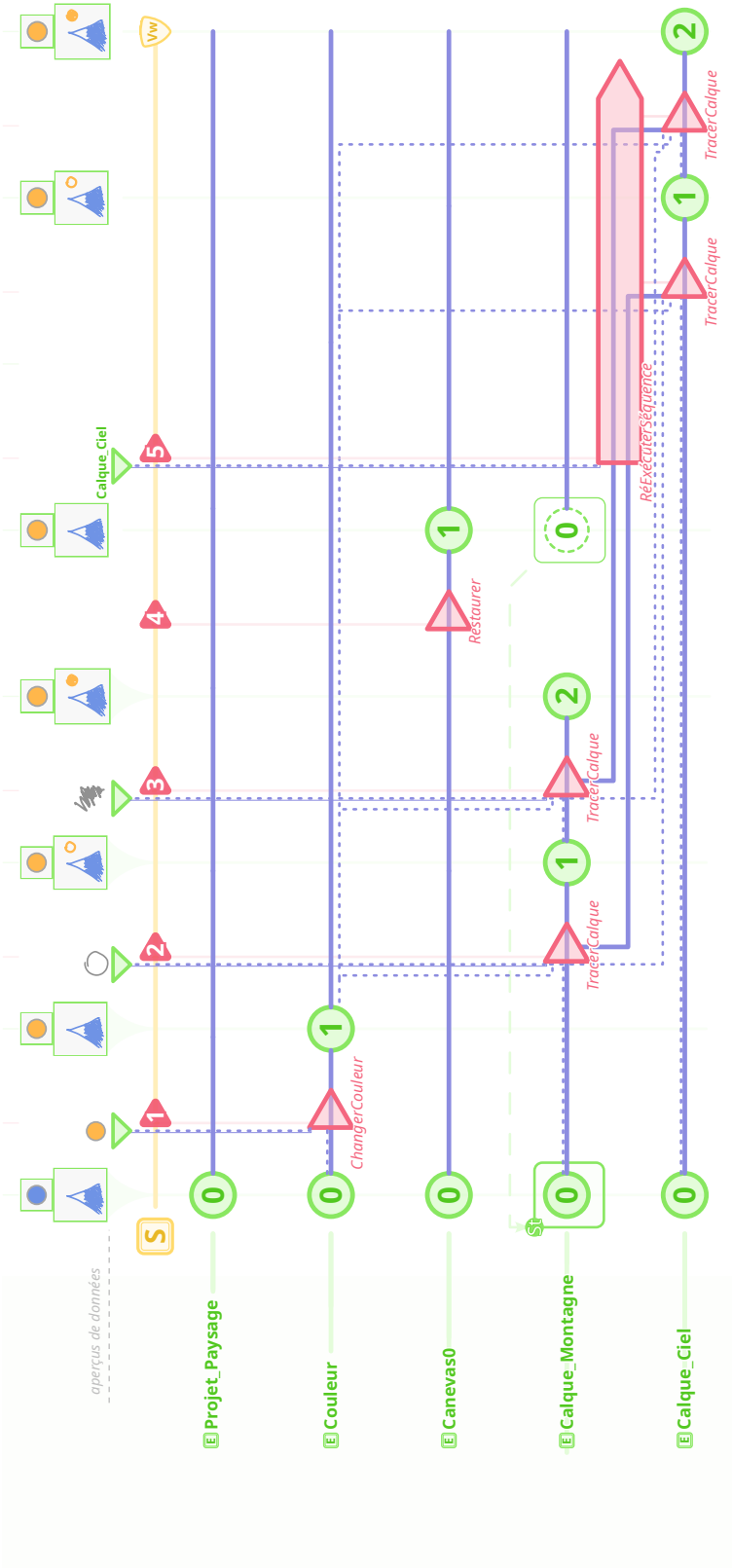

FIGURE 2.12 – Exemple de déroulement d'une Session d'édition avec restitution de State et réutilisation de Commands. La commande composée ré-exécutant les Commands associées aux deux CommandVersions de type TracerCalque de l'historique est étirée pour mieux voir les liens entre elle et les autres Versions.

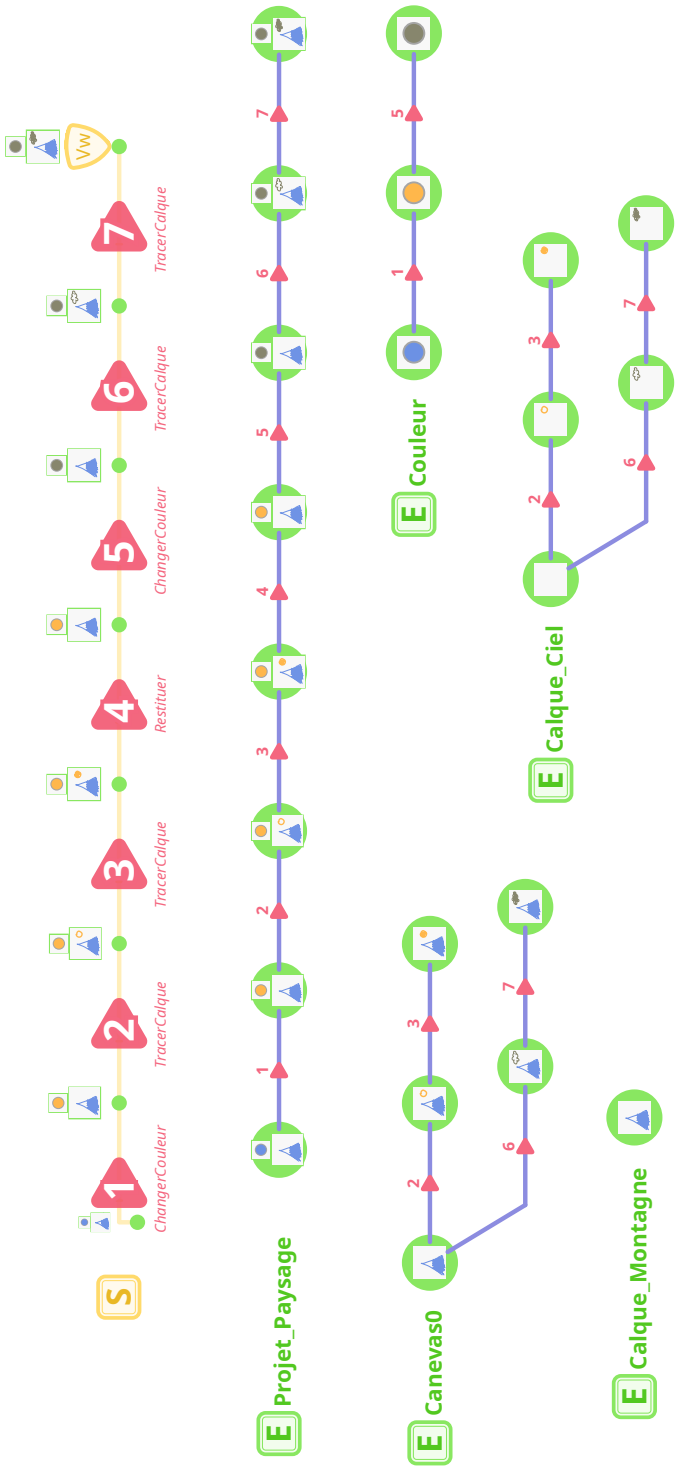

FIGURE 2.13 – Représentation de l'évolution d'une Session et des Elements édités. Les Changes sont numérotés, et les States sont illustrés. Les States des Elements sont représentés sous la forme de cercles verts pleins, et les Changes les ayant impactés sous la forme de triangles rouges pleins. Dans l'historique de la Session, les Changes sont ceux générés lors de cette Session, et les States sont ceux de l'Element à la racine de la View (ici Projet_Paysage).

2.3 Les opérations d'historique dans ESCI

Nous avons maintenant vu les concepts fondamentaux d'ESCI. Ces concepts sont les outils de base pour définir toute opération d'édition. Ces opérations sont définies par le système d'édition, sous la forme de Commands. ESCI ne définit pas de Commands liées au domaine d'édition, telles que TracerCalque ou ChangerCouleur. Nous allons cependant en prédéfinir quelques unes ayant trait à l'historique en soi, telles que Redo et Undo.

Nous allons d'abord présenter l'implémentation des fonctionnalités d'annulation, après quoi nous passerons aux commandes d'annulation et de ré-exécution.

Pour présenter ces fonctionnalités et commandes, nous allons reprendre le scénario précédent en ne conservant que l'historique local de la Session et celui du Projet_Paysage, illustré par la figure 2.14. L'historique local de la Session décrit le travail effectué lors de la Session d'édition, les changements qui y ont eu lieu. Celui de l'Element décrit son édition, ses états générés et manipulés à travers ces changements. On a ainsi une structure suivant le modèle primitif, et une suivant le modèle méta, comme présenté dans le chapitre 1.

Dans l'historique local de l'Element, on peut identifier une séquence de States reliant le State courant à la racine de son historique. Ici, on a la séquence $0, 1, 2, 3, 4, 7$. Ce chemin contient les changements qui n'ont pas été annulés, qui sont *actifs*. Je nommerai cette sous-partie de l'historique la *séquence de construction* du State de destination. Les changements ne faisant pas partie de cette séquence sont considérés inactifs, et ont été défaits par le passé. Cette notion de changements actifs ou non est liée au fait que dans les historiques méta, l'annulation déplace le curseur vers un état antérieur. Les historiques primitifs ne font pas cette distinction. L'historique de la Session n'a donc pas cette notion. On peut voir toutes ses commandes comme contribuant à l'état présent, et donc actives du point de vue de la Session.

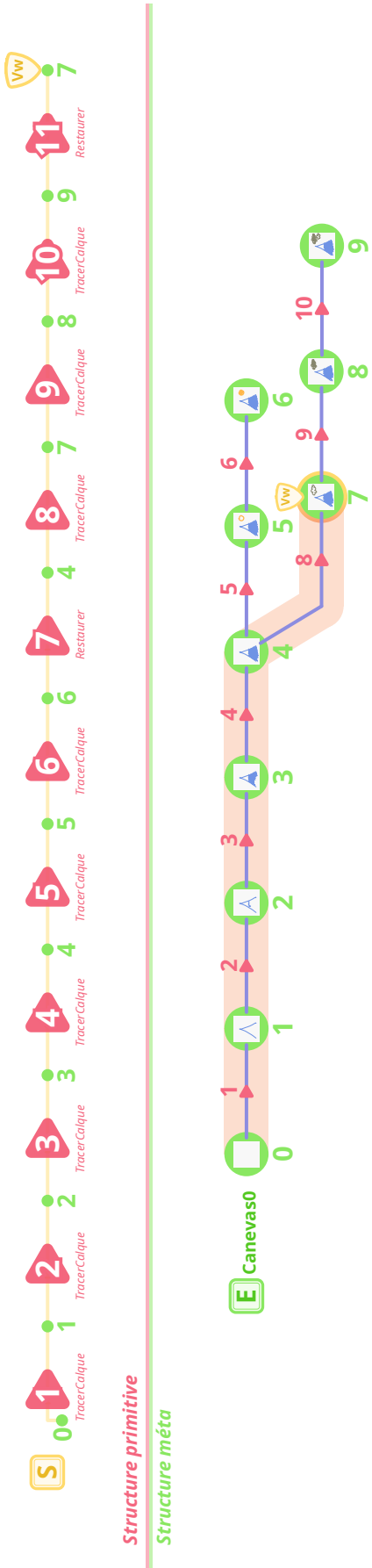

FIGURE 2.14 – Illustration d'une Session d'édition du point de vue de la Session et de l'Element édité. La ligne orange représente le chemin menant vers l'état courant.

2.3.1 Types et stratégies d'annulation

Comme nous l'avons vu précédemment, il existe deux fonctionnalités d'annulation, l'annulation séquentielle et l'annulation sélective. Chacune peut être implémentée de deux manières différentes, par reconstruction d'états ou par inversion de changements.

L'annulation par reconstruction est déjà implémentée par ESCI, le modèle permettant de restaurer tout State passé et de ré-exécuter des Commands. Une Command d'annulation par reconstruction est ainsi facilement implémentée. Une Command utilisant l'annulation par inversion requiert un algorithme d'inversion car l'historique ne possède pas d'informations sur la manière d'effectuer l'inversion. C'est pourquoi cette tâche est laissée à la discrétion du développeur du système. Il peut par exemple créer pour toute Command une Command inverse, comme les méthodes |undo()| du modèle Linear Undo [Qtc].

2.3.2 Commands d'historique

Annulation

Il existe deux types d'Undo, séquentiel ou sélectif, selon la manière de l'historique d'enregistrer cette commande. Si l'historique est primitif, et considère donc qu'Undo est une commande comme une autre, il l'enregistrera. Dans notre modèle, cela correspond à l'évolution de la Session, possédant tous les Changes ayant eu lieu. Undo dans ce contexte, que l'on appellera Undo_P, peut alors s'annuler lui-même. Si par contre l'historique est de type méta, et considère donc Undo comme une commande à part, il ne l'enregistrera pas, mais déplacera un curseur d'un état en arrière. Dans notre modèle, cela correspond à l'évolution de l'Element, avec la View qui se déplace d'un état en arrière lors de l'exécution d'un Undo impactant cet Element. Undo dans ce contexte, que l'on appellera Undo_M, ne peut s'annuler lui-même mais cumule ses effets, reculant de plus en plus. Le modèle présente donc les deux approches, et permet ainsi facilement d'implémenter les deux types d'Undo. Undo_P restituera le State généré par l'avant dernier Change de la Session, suivant la chronologie des exécutions. Undo_M restituera quant à lui le State précédent le State courant de l'Element, suivant les liens de causalité TRIP.

Dans notre exemple précédent, la Command annulée par chacun des deux Undo est encadrée dans la figure 2.15.

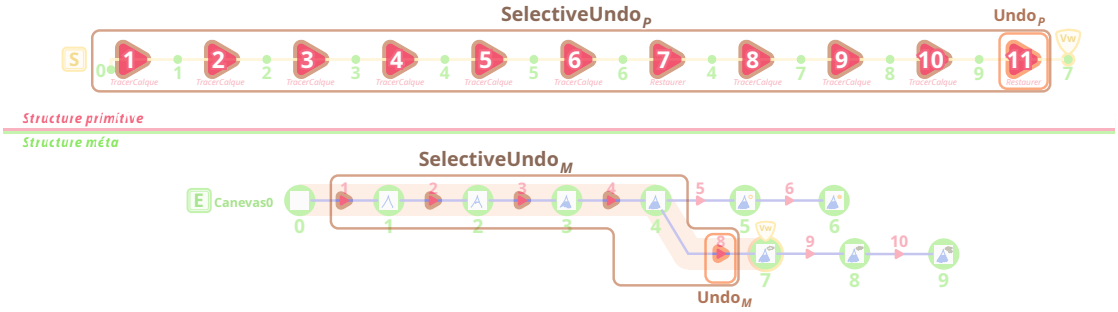

FIGURE 2.15 – Commands concernées par Undo_P et Undo_M.

Quant à l'annulation sélective directe, qui ne prend pas en compte les Paradoxes potentiels en appliquant uniquement l'inverse de la Command à annuler sur l'état courant, elle sera exécutée comme les autres, faisant avancer la Session. Comme elle ne correspond pas à une restitution d'un état passé suivi d'une exécution sélective, mais plus à une modification de l'état courant par l'inverse d'une Command, elle génèrera à partir du State courant un nouveau State.

En ce qui concerne SelectiveUndo, son exécution ajoutera un Change à la Session, et créera une nouvelle branche à l'Element. Cette branche est générée en restaurant un State antérieur, et ré-exécutant les Commands que l'on ne veut pas annuler.

Ré-exécution

Les commandes de ré-exécution sont Redo, SelectiveRedo, Repeat et SelectiveRepeat. La différence entre ces commandes sont les Commands qu'elles peuvent ré-exécuter. Pour Redo, il s'agit des Commands ayant généré un Change à partir du State courant. Repeat prendra la dernière Command exécutée. Selon la gestion des commandes d'historique, cette Command ne sera pas la même. On aura donc deux variantes, $\text{Repeat}_{\text{Ch}}$ et $\text{Repeat}_{\text{St}}$. Les versions sélectives de Redo et Repeat permettent respectivement de ré-exécuter des commandes inactives et actives. La figure 2.16 reprend la Session d'édition de la figure 2.14 précédente, en regroupant par Command de ré-exécution les changements qu'elles peuvent prendre.

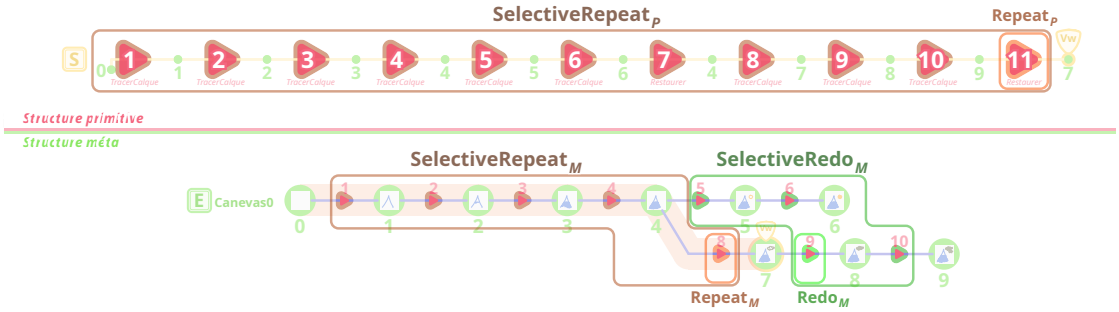

FIGURE 2.16 – Commands concernées par les différentes Commands de ré-exécution.

Insert, Modify, Move, Remove

À partir de ces commandes d'annulation et de ré-exécution, il est possible d'implémenter toutes sortes de Commands d'historique, dont par exemple celles des interfaces utilisateur d'édition d'historique. La seule différence par rapport à ces historiques, dont le modèle Script peut être une représentation, est que notre historique ne supprimera pas d'informations, mais créera une nouvelle branche reflétant les changements désirés.

Insert permet d'insérer une commande dans la séquence menant à l'état final, ce qui correspond à la séquence de construction d'un State dans ESCI. Pour insérer une Command, il suffit d'utiliser Undo pour revenir au State précédent la position de l'insertion, d'exécuter cette nouvelle Command, puis de Redo tous les Changes que l'on vient d'Undo.

Modify permet de modifier les arguments d'une commande. Le processus est le même que pour Insert, sauf qu'au lieu d'exécuter une nouvelle Command, on exécutera la même Command avec des Arguments modifiés.

Remove permet de supprimer une commande de la séquence. Il s'agit là d'un SelectiveUndo. Et pour Move, qui déplace une commande dans la séquence, il suffit de combiner un Remove et un Insert.

Ces Commands peuvent ainsi être combinées pour en créer de nouvelles.

CHAPITRE 3

Architecture logicielle ReTracer et implémentation

Le modèle ESCI définit une structure d'historique polyvalent permettant d'implémenter des commandes d'historique permettant les fonctionnalités des modèles de la littérature. Il précise les informations nécessaires au fonctionnement de l'historique, mais ne développe pas les questions qui peuvent survenir lors de son implémentation, telles

que des détails d'implémentation et la séparation des responsabilités. Pour compléter la définition du modèle avec les aspects architecturaux de son implémentation, je vais présenter dans ce chapitre ReTracer. ReTracer est une architecture logicielle qui permet d'implémenter un historique suivant le modèle ESCI.

Les deux premières sections porteront sur la description de l'architecture ReTracer et des algorithmes associés. La première développera la structure de l'historique, et la seconde son interface. La troisième partie abordera les choix d'implémentation faits pour la librairie ReTracer-rs (modules complémentaires, gestion de la View, ...). Cette librairie suit l'architecture logicielle ReTracer et permet aux développeurs d'intégrer à leur logiciel d'édition un historique qui suit le modèle ESCI. J'utilise le suffixe "-rs" pour différencier la librairie ReTracer en Rust (dont "rs" est l'abréviation) de l'architecture ReTracer dont elle est une implémentation. Pour illustrer l'utilisation d'un historique ReTracer, un exemple d'utilisation de ReTracer-rs sera donné dans le contexte d'un programme simple de compteurs.

3.1 L'architecture logicielle de l'historique ReTracer

L'architecture ReTracer possède une partie dédiée à la conservation de l'historique et une autre à l'interaction entre le logiciel et l'historique. L'enregistrement des données nécessaires pour un historique ESCI a ainsi pu être étudié séparément de l'étude de l'utilisation de cet historique.

La partie historique est composée de Types, d'Elements, de Sessions et de Versions. Elle reprend les notions d'ESCI en ajoutant des détails d'implémentation, en particulier pour les liens TRIP.

3.1.1 Types

Pour décrire la nature des Elements, Inputs et Commands, des types sont enregistrés dans l'historique. Un type possède un nom unique qui l'identifie et des liens vers d'autres types dont il hérite. Ces liens peuvent pointer vers un ElementType, CommandType ou InputType, permettant par exemple à un InputType de pointer vers un ElementType pour exprimer que les données de ces Inputs sont équivalentes à celles des Elements du type indiqué. Le but est de désigner la nature de la donnée indépendamment de la manière avec laquelle elle est gérée dans l'historique. Un Input contenant un calque à lire pourra être utilisé à la place des données d'un State d'un Element calque, et à l'inverse, les données d'un State d'un Element contenant un tracé pourront être utilisées

à la place d'un Input en contenant un.

Outre ces informations communes aux Types, les ElementTypes possèdent le nom des Attributes que leurs Elements ont à tout moment de l'édition. Les Attributes qui peuvent être ajoutés et supprimés n'y figurent pas. À chaque Attribute peut être associé un ElementType pour signifier à l'historique la nature des données qui y sont attendues. Les CommandTypes possèdent quant à eux le nom des Arguments que les Commands auront, chacune associée au Type des données attendus. Le système de typage existe à titre indicatif, pour permettre d'organiser les données et d'effectuer des propositions dans le cas de paradoxes entre autres, en listant les States d'Elements dont le type correspond par exemple. Comme le typage est plus une indication qu'une nécessité, le système reste assez simple et flexible. Il pourrait cependant être enrichi en étudiant les usages possibles des States, Inputs et Changes pour voir comment contraindre le système au lieu de permettre n'importe quelle substitution. Ce système contraint le logiciel à déclarer la nature de toute donnée, mais il crée ainsi une couche d'abstraction par rapport aux types concrets propres au logiciel. Deux logiciels différents peuvent alors utiliser le même historique tant qu'ils partagent les mêmes Types.

Le diagramme 3.1 récapitule ces structures.

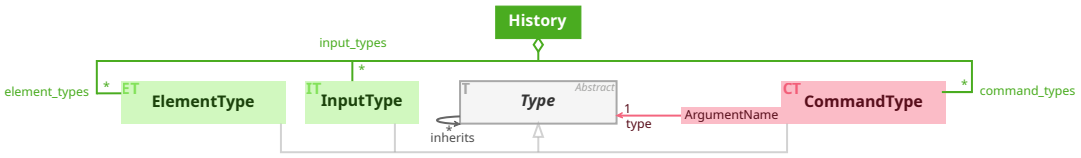

Figure 3.1 – Diagramme de classe des Types enregistrés dans l'historique. Les structures relatives aux données manipulées sont en vert, et celles représentant les changements effectués sont en rouge.

3.1.2 Elements et Sessions

L'historique contient pour chaque Element et Session une structure correspondante, qui contient un identifiant unique. Les Elements sont reliés à leur ElementType et aux Elements qui ont été créés par une copie d'un State de cet Element. Ils contiennent aussi l'ensemble de leurs ElementVersions. L'intérêt de relier l'Element à ses copies est de pouvoir les identifier rapidement et de pouvoir les proposer comme alternative dans le cadre de paradoxes par exemple.

Les Sessions possèdent des informations de synchronisation et la liste des CommandVersions y ayant été créées. Les informations de synchronisation sont une liste de triplets (Path, Timestamp, Timestamp) représentant la partie de la View synchronisée et l'intervalle temporel pendant lequel la partie est synchronisée. Un Path représente la position d'une ElementVersion dans un State, c'est-à-dire le chemin entre l'ElementVersion racine du State et l'ElementVersion en question, à la manière d'un chemin dans un système de fichiers. Ce chemin est effectué en empruntant les liens de parenté identifiés par les Attributes. Le State pointé par le Path doit être le même à travers les Sessions à synchroniser pour que la synchronisation puisse avoir lieu, ce qui signifie qu'une synchronisation de l'état du canevas entre plusieurs Sessions nécessite qu'il soit dans le même état au départ.

3.1.3 Versions

Toute Version est liée par des liens TRIP à la CommandVersion de la Command l'ayant générée et vice-versa. Ces liens sont bidirectionnels pour permettre une navigation par liens de causalité.

En plus de ces liens, les ElementVersions sont reliées aux ElementVersions enfant qu'elles ont en Attribute et à l'ElementVersion parent au moment de leur création. Ces Attributes sont à minima ceux déclarés dans l'ElementType de l'Element associé. D'autres peuvent être ajoutés, ce qui permet de représenter des listes de taille dynamique comme la liste des calques d'un canevas.

Les CommandVersions contiennent les Arguments et le CommandType de la Command exécutée et sont liées aux CommandVersions des Commands enfant exécutées par celle-ci, et au parent qui les a exécutées. Elles possèdent également un horodatage marquant le début de leur exécution, l'état de l'exécution (en cours, terminée avec succès ou échouée) et les informations décrivant son fil d'exécution. L'horodatage a pour but de marquer le moment où l'exécution a eu lieu. L'état de l'exécution permet de savoir si la CommandVersion enregistrée a abouti et fait avancer l'état de la View ou s'il s'agit d'un échec d'exécution enregistré malgré tout, mais qui ne contribue pas à l'avancement de l'état du projet, l'exécution ayant échoué. Les informations décrivant son fil d'exécution sont les liens TRIP générés lors de celle-ci, les Paradoxes rencontrés et les informations de correction des Paradoxes utilisées (les *Overrides*), présentés dans la section suivante.

Le diagramme 3.2 résume les liens entre classes enregistrées dans l'historique.

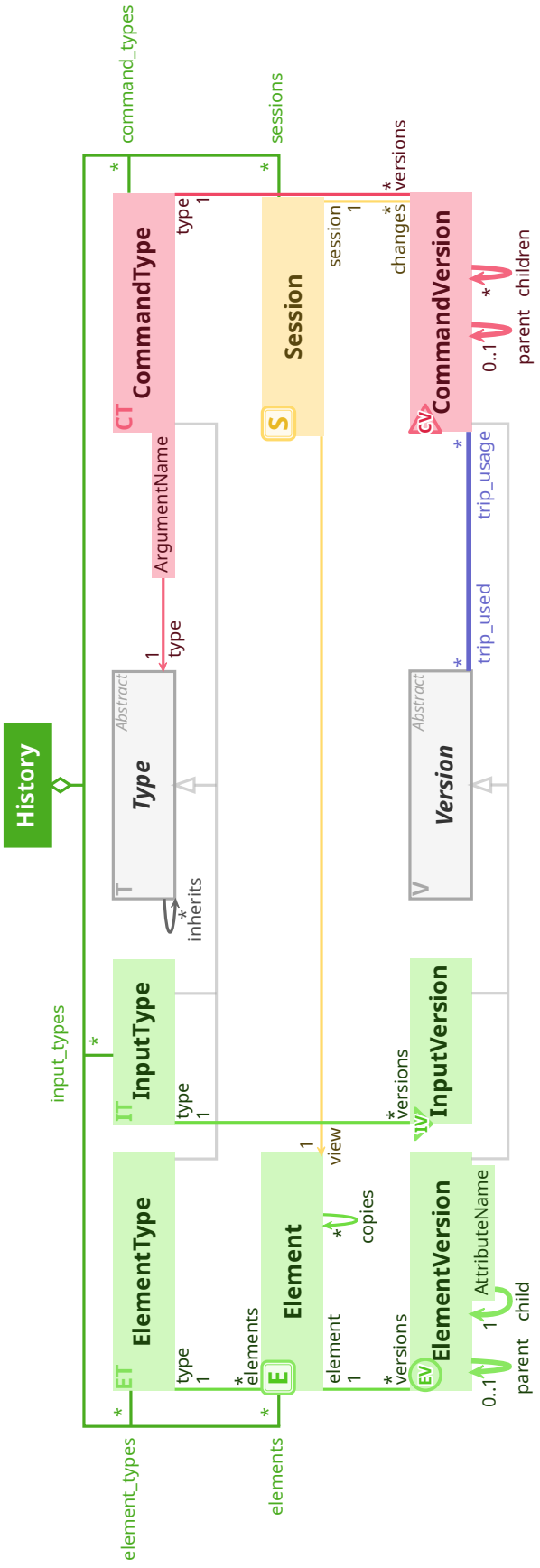

Figure 3.2 – Diagramme des classes enregistrés dans l'historique ReTracer. Les liens TRIP sont représentés en bleu, et la Session en jaune, exprimant respectivement les liens de causalité et de chronologie.

3.1.4 Liens TRIP

Informations de liaison

Les informations relatives aux liens TRIP sont enregistrées dans la CommandVersion de l'exécution les ayant générés. Le tableau contenant ces liens représente les effets de l'exécution sur les données enregistrées dans l'historique. Chaque lien TRIP pointe vers la Version dont il décrit la relation avec la CommandVersion. En retour, la Version pointe vers ce lien, ce qui rend les liens TRIP bidirectionnels.

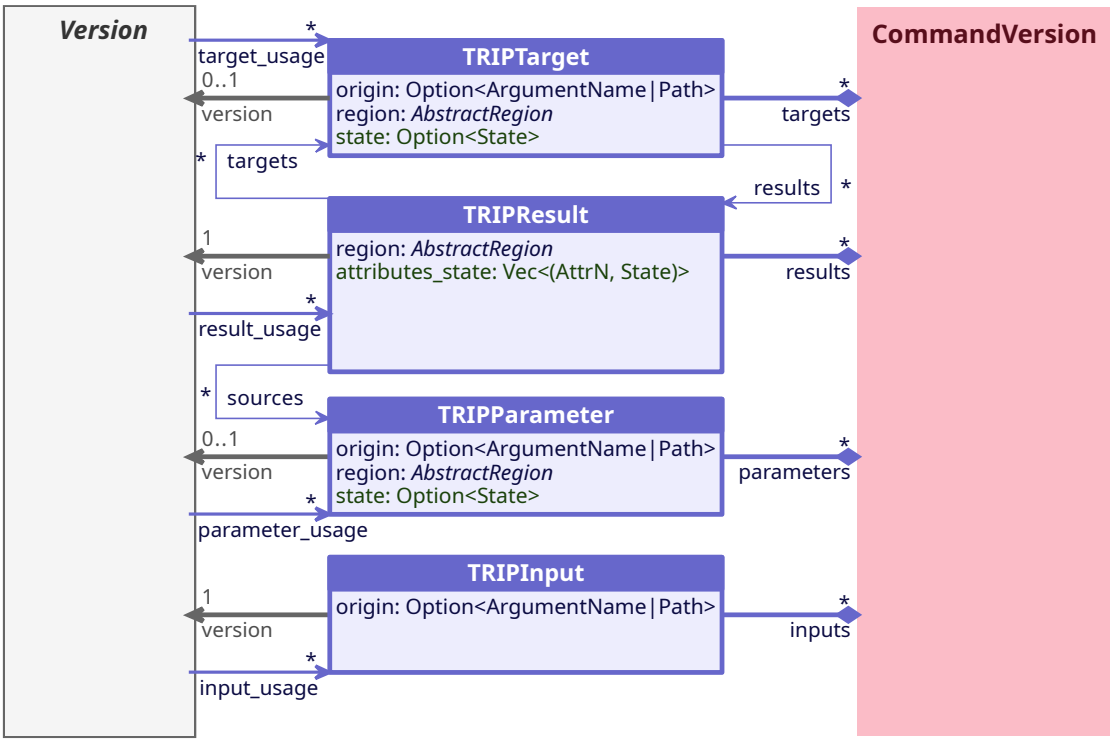

Figure 3.3 – Détail des structures pour enregistrer les liens TRIP

La figure 3.3 représente les informations contenues dans les liens TRIP enregistrés dans les CommandVersions, reliant des Versions à une CommandVersion.

origin : Si la Version provient d'un Argument, le nom de l'Argument, et si elle provient de l'état de la View, sa position dans la View. Cela permet d'enregistrer la manière par laquelle la CommandVersion a accédé à cette Version.

region : Il est possible de décrire la région concernée par une modification ou une lecture, région qui sera enregistrée dans les liens Target, Result et Parameter correspondants. Le choix du langage de description de la région est libre.

targets et results : Pour représenter le passage d'une ElementVersion à une autre, les liens Result pointant vers les ElementVersions résultantes sont reliés aux Targets pointant vers les ElementVersions d'origine, et inversement.

sources : De plus, les Results sont reliés aux Parameters lus pour créer la nouvelle ElementVersion, ce qui permet d'exprimer les influences entre données. Changer le contenu de l'un de ces paramètres aura pour effet d'obtenir un autre résultat.

state : Lorsque la Version utilisée est une ElementVersion, les liens Target et Parameter ont besoin de connaître le State depuis lequel elle est utilisée. Cela permet de parcourir les états de l'Element par liens de causalité, en passant de son état pré-changement à son nouvel état post-changement et inversement.

attributes_state : Pour restituer l'état d'un Element enfant, il faut spécifier le State en question, car le lien seul vers une ElementVersion ne suffit pas, une ElementVersion pouvant faire partie de plusieurs States différents.

Les Targets et Parameters peuvent échouer, c'est à dire que le State à lire n'existe pas. C'est pourquoi ils peuvent ne pas avoir de Version associée. Ils seront quand même enregistrés à des fins de comparaison avec des ré-exécutions ultérieures.

Exécution et Paradoxes

Au cours de l'exécution de la Command, toute lecture et modification de données est enregistrée dans la CommandVersion sous la forme de liens TRIP. Le tableau des liens TRIP retrace ainsi chronologiquement les opérations effectuées lors de l'exécution.

Ces liens s'appliquent le plus souvent à des ElementVersions et InputVersions, mais il se peut que la Version visée soit une CommandVersion, comme dans le cas d'une ré-exécution : une Command peut effectuer une ré-exécution en prenant la CommandVersion de l'exécution initiale comme Target, et en ré-exécutant la Command, ce qui crée une nouvelle CommandVersion liée par Result.

Lors de la ré-exécution, il faut pouvoir détecter les Paradoxes, qu'il s'agisse d'ambiguïtés ou d'inconsistances. Pour cela, tout lien TRIP enregistré dans le tableau est comparé à celui de l'exécution initiale. S'il y a une différence, l'ambiguïté est enregistrée dans un Paradox lié à ce lien TRIP. Lorsque la Command tente de lire ou de modifier une donnée qui n'est pas présente dans l'état courant, l'inconsistance est enregistrée dans un Paradox, et l'exécution est interrompue si nécessaire.

Un Paradox est relié au lien TRIP initial et possède la liste des données candidates qui peuvent être utilisées à la place. Il est aussi relié au lien TRIP paradoxal dans le cas d'une ambiguïté, et contient le Path menant à la donnée absente dans le cas d'une

inconsistance. Cela permet de marquer la présence d'alternatives au fil d'exécution suivi.

Une exécution qui échoue peut malgré tout être enregistrée dans l'historique, auquel cas la CommandVersion conservera ces informations de paradoxes. Il est possible de désigner les candidats à utiliser pour résoudre les Paradoxes ou d'utiliser d'autres données non proposées, puis de ré-exécuter la Command avec. Cela permet de diriger le déroulement de l'exécution, et ainsi de résoudre des inconsistances puis de choisir des chemins alternatifs où il y a des ambiguïtés. Le choix est fait librement par le logiciel, qui peut prendre sa décision en suivant une stratégie de résolution de paradoxes de la littérature par exemple. Ce système de décision ne s'applique pas uniquement aux liens paradoxaux, mais à tout lien d'une CommandVersion, ce qui permet soit de forcer des choix pour un lien en amont, et peut permettre d'éviter la génération de liens paradoxaux par la suite, soit juste d'obtenir un résultat d'exécution différent. Ces choix sont nommés "Overrides", car ils prévalent sur celui fait par défaut lors de l'exécution, s'il y en a un.

3.1.5 States

Un State enregistre deux couples d'informations :

- **at** est un Path qui donne la position du State dans la View.
- **from** pointe vers le Change ayant généré ce State, et contient aussi l'index du State et un horodatage.
  - **root** pointe vers l'ElementVersion racine du State.
  - **tree** pointe vers toutes les ElementVersions enfant du State, reconstruisant ainsi l'arbre d'état.

Le couple (at, from) décrit de manière unique le State en reprenant les informations nécessaires à son identification, et (root, tree) contient les liens vers les ElementVersions le constituant.

L'horodatage utilisé dans le premier couple est utile dans les situations de synchronisation de Sessions. Comme le Change n'est pas copié dans toutes les Sessions synchronisées, il n'y a pas de Change sur lequel pointer dans chaque Session. Pointer vers le Change de la Session source n'est pas non plus une solution, comme les synchronisations peuvent être partielles, ce qui signifie que les States obtenus pour chaque Session ne sont pas forcément identiques. À la place, "from" pointera vers le dernier Change de la Session utilisée, et l'horodatage sera celui du dernier Change à exécuter. Cela permet de signifier qu'il faut ajouter après le Change pointé de la Session utilisée les Changes des autres Sessions synchronisées s'étant exécutés entre ce Change et l'horodatage.

Le second couple peut être omis pour limiter la quantité de données enregistrées. Il pourra être recalculé par la suite en suivant dans le sens inverse les liens TRIP permettant de constituer le State.

Constitution de States

Les nouvelles ElementVersions pointées par les Results forment progressivement un nouveau State. Elles remplacent les anciennes ElementVersions reliées par Target. Ce système de différentiel ne marque que les changements qui ont eu lieu. Les ElementVersions n'ayant pas changé feront partie du nouveau State. Au lieu de cumuler toutes les nouvelles ElementVersions générées lors de l'exécution en un seul State final, il est possible de créer des States intermédiaires. Pour cela, un groupe de pointeurs vers les Results contribuant à un State est créé et enregistré dans le couple (root, tree). La CommandVersion enregistre la liste des groupes de pointeurs, qui permettra de constituer les States successifs. Cela permet non seulement au logiciel d'enregistrer des états intermédiaires pour une utilisation ultérieure, mais cela permet aussi d'effectuer des modifications de l'état de la View pour préparer l'exécution d'une Command enfant qui s'exécutera à partir de ce nouvel état. Celui-ci retournera une View qu'il aura également pu faire avancer.

Toute modification d'ElementVersion contribue au prochain State. Mais il se peut qu'après avoir modifié un Element, par exemple un calque, on modifie l'un de ses parents, le canevas. Toute modification ne modifiant pas ce lien de parenté est acceptée. Cependant, supprimer le calque du canevas rendra la nouvelle ElementVersion représentant le calque modifié orpheline. Le logiciel peut alors enregistrer le State obtenu après modification du calque, puis enregistrer le State suivant résultant de la suppression du calque du canevas. Cela permet de créer des alternatives dans le cadre de l'exploration d'idées entre autres.

Cette liste de States décrit l'évolution de l'état de la View au cours de l'exécution, qui se termine par le State final dans lequel sera la View après cette exécution. Les CommandVersions possèdent en plus de cette liste de States la possibilité de créer des listes en parallèle, commençant à un State donné, et constituant d'autres States détachés de la View. Cela permet à la Command de modifier n'importe quel State de l'historique sans être limité à la View. Cette liberté permet de dépasser le mode d'édition classique où l'édition se fait par rapport à un état courant (ici pointé par la View). Ce nouveau mode d'édition prend en compte l'historique complet du projet et offre ainsi de nouvelles opportunités à explorer. L'enregistrement de ces States détachés est identique à celui de

la liste de States de la View, mais chaque liste est précédée d'un State de départ.

3.1.6 Édition de l'historique

Métadonnées

À chaque Type, Element, Session, State Input ou Change peut être associé des métadonnées. Les CommandVersions contiennent les métadonnées des States qu'elles génèrent. Le but de ces métadonnées est de permettre au logiciel d'annoter et de catégoriser le contenu de l'historique librement. Ces métadonnées sont éditables par le logiciel et leur évolution n'est pas enregistrée. Il n'y a donc pas de lien TRIP créé pour cela.

Mode contrôle d'historique

L'historique peut être édité par le logiciel au moyen du mode contrôle d'historique, opération qui n'est pas non plus enregistrée. Le logiciel peut ainsi supprimer des données que l'utilisateur juge inutile ou sensibles. Quelques opérations conservant la validité de l'historique ont été identifiées.

Des CommandVersions ne créant et ne lisant aucune donnée peuvent être simplement supprimées, comme celles d'une commande d'enregistrement. Sinon, les données créées peuvent être à leur tour supprimées, à condition qu'elles ne soient pas utilisées par d'autres CommandVersions, ou alors ces CommandVersions peuvent à leur tour être supprimées, et ainsi de suite. Cette stratégie de suppression en cascade permet de supprimer complètement une partie de l'historique comme une branche. Une autre approche consiste à remplacer une séquence de CommandVersions par une seule nouvelle CommandVersion qui rassemble tous les liens TRIP de la séquence. Cela permet de supprimer les changements dans un intervalle de temps, tout en reliant l'historique d'avant et d'après cet intervalle. Ces changements peuvent être remplacés pour simplifier l'historique, ou pour des raisons confidentielles. J'appellerai ce genre de remplacement une ellipse. Un CommandType devra être associé à cette ellipse, ainsi qu'un moyen de recalculer les données qu'elle génère. Pour cela, le logiciel peut déclarer un CommandType "Ellipse" dont la CommandVersion ne fera qu'enregistrer les données générées par les CommandVersions qu'elle remplace dans la base de données, qui les restituera au besoin. Outre la suppression en cascade et la création d'ellipses, il est aussi possible de supprimer une CommandVersion, puis de ré-exécuter toutes les Commands qui ont suivi, en remplaçant les CommandVersions initiales par les nouvelles générées. Les systèmes de gestion de l'exécution et de détection de Paradoxes sont utilisables ici

aussi. Cette approche ressemble à une restitution d'un état passé suivi d'une ré-exécution des Commands, sauf qu'ici, les exécutions initiales sont écrasées par les nouvelles. Les modifications et insertions de Versions se font en respectant ces mêmes règles.

Là où les exécutions de Commands ne font qu'ajouter des Versions à l'historique, l'édition de l'historique peut en modifier, en supprimer et en insérer dans l'historique. Pour éviter tout problème, lorsqu'une opération d'édition d'historique est en cours, aucune autre modification n'est autorisée par d'autres Sessions (celles de collaborateurs par exemple), pour lesquels l'historique est alors en lecture seule. Contrairement à l'édition de l'historique, l'exécution de Commands et l'enregistrement de Records provenant d'autres historiques peuvent se faire en parallèle tant qu'elles ont lieu dans des Sessions différentes et ne manipulent pas des parties synchronisées s'il y en a. Cela permet de limiter le verrouillage de l'historique aux seuls cas non contrôlables pas l'historique.

3.1.7 Names, UIDs, References et Records

Pour identifier des parties de l'historique à des fins de parcours, d'édition ou pour partager des informations entre collaborateurs, ReTracer possède des identifiants. Tout d'abord, l'historique même a un identifiant unique généré lors de sa création. Les Types sont identifiables de manière unique par leur nom et la sorte de type dont il s'agit (ElementType, InputType ou CommandType). Les Elements et Sessions ont tous un identifiant unique qui leur est attribué lors de leur génération.

References

Les Versions sont enregistrées dans des tableaux et ont donc un index associé. Ces indexes sont utilisables au sein de l'historique, mais pour un usage extérieur, il est préférable d'utiliser des notions insensibles à la position des Versions dans ces tableaux, celle-ci pouvant varier entre historiques de collaborateurs en fonction de l'Input ou de l'ElementVersion enregistré en premier par exemple. Pour cela, on utilise des References qui identifient de manière unique un State, un Input ou un Change. Une ChangeReference est composée de l'identifiant de l'historique, de celui de la Session dans laquelle le Change a eu lieu et de l'horodatage de la CommandVersion racine. Une InputReference est composée d'une ChangeReference désignant la CommandVersion ayant généré l'Input ainsi que l'index de celui-ci, comme plusieurs Inputs peuvent être enregistrés lors d'une même exécution. Il en va de même avec les StateReferences, composées d'une ChangeReference et de l'index du State en question dans les listes de

States enregistrées par la CommandVersion.

Records

Pour partager des States, Changes et Inputs à travers des historiques, on peut utiliser les Records, qui contiennent une copie des Versions en question. Il est ainsi possible d'exporter des données vers d'autres historiques, qu'ils portent sur le même projet ou non. Pour cela, chaque Record possède une liste de dépendances à satisfaire pour pouvoir enregistrer son contenu. Il s'agit là des identifiants des Types, des Elements, des Sessions et d'autres Versions dont celles-ci dépend.

3.1.8 Base de données

L'enregistrement des données associées aux Versions est effectué séparément de la structure de l'historique, dans une base de données créée par le logiciel. Cela permet au logiciel de choisir la manière de les enregistrer en fonction de leur nature, et d'optimiser leur conservation pour les usages prévus. La base de données est contrôlée par l'historique, qui y enregistre les données au cours des exécutions de Commands, en utilisant les References comme clef. L'interface entre historique et base de données est composée d'une méthode d'insertion, de suppression et de lecture des données.

La conservation des données est laissée à la discrétion de la base de données, qui peut tout enregistrer sous la forme de snapshots ou enregistrer des différentiels par exemple. Elle peut aussi lire l'historique pour reconstituer des données qu'elle aura supprimées pour gagner en espace mémoire. Cette reconstitution peut se faire de différentes manières, en partant d'un snapshot passé et en rejouant les Commands suivantes, ou en utilisant les données de l'état présent et en exécutant des fonctions inversant les effets des Commands exécutées depuis. Le but de la séparation est, à l'instar du système de Types, d'isoler les données spécifiques au logiciel de la structure de l'historique. Cette structure peut ainsi être partagée entre différents logiciels tout en gardant séparée la partie spécifique à chacun d'eux. Aussi, chaque logiciel peut décider lui même de la manière de conserver et de reconstituer les données éditées dont le format est spécifique au logiciel, sans que cela n'ait d'impact sur l'historique.

3.2 L'architecture de l'interface de ReTracer

L'interface permet de parcourir l'historique, d'exécuter des Commands et d'éditer l'historique.

3.2.1 Queries

Pour parcourir l'historique, les Types, les Elements, les Sessions, les States, les Changes et les Inputs sont utilisés. Les liens chronologiques permettent de passer d'un Change à un autre et les liens de causalité permettent de passer d'un State à un autre. Les Changes sont enregistrés séquentiellement dans les Sessions. Les States, les Changes [1] et les Inputs sont liés aux Changes qui les ont générés. Ce réseau de structures est utilisé pour se déplacer dans l'historique.

La lecture de ces structures est libre, par contre l'ajout de nouvelles structures est réservé aux Commands en cours d'exécution, et la modification des structures existantes est réservées au mode contrôle d'historique.

Pour gérer les droits d'accès aux données de l'historique, l'interface propose une structure qui enveloppe les Types, Elements, Sessions et Versions, et ne permet d'effectuer que les opérations permises en fonction du contexte. Cette structure est nommée Query, et présente au logiciel les méthodes de parcours tout en vérifiant les droits qu'il a selon la situation donnée. À la manière d'une requête pour une base de données, Query enregistre le chemin à effectuer pour obtenir les données souhaitées. La récupération n'est effectuée que lorsque le logiciel exprime le but de la navigation à travers une méthode de lancement de requête. Un objet Query peut être généré par l'historique ou la View directement, et permet de lire, de demander la modification d'une structure ou d'en créer de nouvelles.

3.2.2 Exécuter des Commands

Patron de conception Command

Une Command est exécutée à partir d'une View, qui représente l'état actuel de la Session en cours. La View possède une méthode "run(commande)" prenant une Command en paramètre. ReTracer utilise le patron de conception Commande, où Command est une interface que des classes du logiciel peuvent implémenter. Elle possède des méthodes d'accès aux Arguments de la commande, son CommandType, ses métadonnées et une méthode pour l'exécuter. Le but d'utiliser ce patron de conception est de simplifier l'utilisation de ReTracer à la place d'un historique existant, en reprenant l'interface existante et en l'enrichissant.

La méthode "run" prend en paramètre la View qui donne accès à l'historique, aux Arguments mis à jour au préalable, et permet d'enregistrer les changements. Si la mise

1. Des Changes sont générés par d'autres Changes dans le cadre d'une ré-exécution par exemple, comme décrit dans la section précédente.

à jour des Arguments échoue, l'exécution se terminera prématurément et renverra les informations d'erreur de mise à jour.

Exécution de Commands

Au cours de l'exécution, la Command peut enregistrer de nouveaux Inputs, de nouveaux Types ou de nouveaux Elements. Les Types et Elements peuvent être enregistrés à partir de la View en donnant les informations les constituant. Pour les Inputs, une interface est mise à disposition, permettant de donner son type, ses métadonnées et les données qu'il contient à l'historique, qui les enregistrera dans la base de données.

La Command peut également lire ou modifier l'état d'un Element en le récupérant à l'aide d'une Query. Si elle souhaite lire les dimensions du canevas, elle peut demander à la View "view.query('/canevas/dimensions').read()", utilisant un Path pour désigner les données à lire dans la View et demandant un accès en lecture. Les données correspondantes lui seront alors retournées. Ce système permet à l'historique d'être informé des usages de ses données, ce qui lui permet de les tracer automatiquement, sans demander au logiciel de le faire. Ici, il ajoutera à la CommandVersion de la Command en cours un lien Parameter vers le State lu pour marquer cette lecture. Pour modifier les dimensions, la Command utilisera la méthode "change(nouvelles_donnees)" en lui donnant les nouvelles données à enregistrer dans la base de données, ce qui générera une nouvelle Version. Un lien Target vers l'ElementVersion ciblée sera créé, ainsi qu'un Result vers la nouvelle ElementVersion générée. Tout enregistrement de lien Parameter ou Target déclenche la détection de Paradoxes , qui seront également enregistrés dans la CommandVersion. Une fois toutes les modifications effectuées pour obtenir un nouveau State, "view.record_state()" enregistrera dans la CommandVersion la création d'un nouveau State cumulant les dernières modifications effectuées. Dans le cas discuté précédemment où une ElementVersion est modifiée avant d'être supprimée de l'état présent par l'un de ses parents, elle ne sera pas enregistrée, cette modification se retrouvant orpheline. la méthode peut alors renvoyer un message alertant la Command de cet évènement. La Command peut également enregistrer de nouveaux Inputs avec "record_input(input)", et exécuter d'autres Commands à l'aide de la même méthode "run" qui a permis de l'exécuter.

Paradoxes et la gestion des Changes

Lorsque l'exécution d'une Command se termine, la View reprend la main pour vérifier l'intégrité des modifications. Si l'exécution a abouti et si aucun Paradox n'est

survenu, elle enregistrera automatiquement ce nouveau changement dans l'historique. On peut comparer cela aux systèmes de transaction des bases de données. Le cas échéant, elle retourne à l'appelant les données de changement dans une structure Change, ce qui permet à la Command parente ou à l'utilisateur à l'origine de l'exécution de résoudre le problème et de ré-exécuter la Command avec les corrections nécessaires. Ces corrections sont faites à l'aide d'Overrides. La structure Change contenant les modifications à enregistrer dans l'historique permet de parcourir les liens TRIP qu'elle possède, et de leur associer un Override. "change.trip()" permet d'obtenir les liens, et "trip_link.override(state_a_utiliser)" permet d'associer un Override à un lien.

L'échec de l'exécution n'empêche pas l'enregistrement du Change dans l'historique, mais contrairement à une exécution qui aboutit, il faut le demander explicitement à la View "view.record_change(change)". L'absence d'enregistrement automatique dans ce cas permet d'abandonner les changements qui ont échoués avant qu'ils soient enregistrés.

L'appelant peut aussi demander une ré-exécution, que le Change soit enregistré ou non. Cette ré-exécution prendra en compte les Overrides et retournera à son tour un Change si un problème survient lors de l'exécution.

De cette manière, il est possible de gérer les Paradoxes librement, de même que les autres erreurs d'exécution. Il est bien sûr possible de définir des fonctions implémentant des stratégies de résolution de Paradoxes de la littérature et de leur donner le Change en question pour qu'ils le corrigent à l'aide d'Overrides. Dans le cadre de l'édition collaborative, ces Changes peuvent être échangés avec d'autre collaborateurs pour décider de la manière de les résoudre si nécessaire. Leur enregistrement dans l'historique entraînera la propagation du changement à travers les autres historiques. La propagation des changements entre historiques est effectué en partageant les ChangeRecords contenant l'ensemble des Versions composant le changement à partager. Cette propagation peut être automatisée à l'aide d'un observateur des exécutions effectuées. Chaque logiciel recevant les ChangeRecords propagés pourra simplement les enregistrer dans son historique pour rester à jour avec autres. Dans le cas de l'édition collaborative en temps réel, il pourra aussi utiliser un système de synchronisation de sa Session avec les changements des autres, comme l'utilisation de la fonctionnalité de synchronisation des Sessions d'ESCI ou en utilisant des stratégies comme OT ou CRDT pour ré-exécuter la Command ayant généré le Change dans leur propre Session.

3.2.3 Mode contrôle d'historique

L'historique est lui même modifiable, à condition de conserver la validité de sa structure. Une opération d'édition d'historique est entreprise à l'aide de la méthode "enter_control_mode()" de la View ou de l'historique. Elle renvoit un accès à l'historique même permettant de modifier son contenu. Les modifications sont vérifiées avant d'être enregistrées lors de la sortie du mode d'édition.

3.3 La librairie ReTracer-rs

Cette architecture a été développée sous la forme d'une librairie en Rust, utilisable dans les projets de logiciels d'édition. Le but de cette librairie est d'être utilisable dans des contextes variés, ce qui permettra d'évaluer le modèle dans des cas d'usage différents. La possibilité de l'exporter en WASM permettra de l'utiliser dans des projets web, c'est pourquoi j'utiliserai du Typescript dans les extraits de code de cette section.

3.3.1 Exemple d'utilisation

Pour utiliser la librairie, il faut définir des commandes implémentant l'interface Command. L'extrait de code 3.1 est un exemple d'une commande permettant d'incrémenter un compteur reçu en Argument.

La commande a une méthode pour donner son CommandTypeName et ses Arguments, et une méthode d'exécution lors de laquelle elle récupère la valeur courante du compteur, l'incrémente puis la lui affecte en tant que nouvelle valeur. Elle termine par l'enregistrement des modifications en tant que nouvel état.

L'exécution de cette Command a pour effet d'enregistrer une nouvelle ElementVersion correspondant à ce nouvel état, et une nouvelle CommandVersion pour représenter le changement qui a eu lieu, relié par lien TRIP aux données modifiées de l'état précédent en lecture et modification (Parameter et Target), et aux nouvelles données formant le nouvel état résultant de cette modification (Result).

Pour initialiser l'historique, une Command qui crée les Elements et States initiaux est définie. Elle peut aussi enregistrer les Types. Son Type doit cependant être enregistré avant exécution.

La Command d'initialisation définie dans l'extrait 3.2 enregistre d'abord l'ElementType "Counter" et le CommandType "Increment". Elle continue en créant un Element de type "Counter", crée un nouveau State pour celui-ci, lui associe la valeur 0 et enregistre ce nouveau State. Cela va créer une ElementVersion représentant l'état initial de l'Element.

```
class Increment extends Command {

  args: Arguments;

  commandType() {
    return "Increment";
  }

  arguments() {
    return this.args;
  }

  run(view: View) {
    var value = view.argument("counter").read();
    value += 1;
    view.argument("counter").change(value);
    view.recordState();
  }
}
```

Listing 3.1 – Exemple de définition d'une commande d'incrémentation

La Command termine en déplaçant la View vers ce nouveau State, qui devient ainsi l'état courant de la View. "moveTo" permet de déplacer la View vers n'importe quel State de l'historique, permettant ainsi de le restituer. Ce déplacement doit également être enregistré, d'où le "recordState" final.

Au lancement du logiciel, l'historique est soit chargé à l'aide d'une méthode "load" lisant un fichier contenant les données de l'historique, soit créé à l'aide de "new" comme illustré dans l'extrait de code 3.3. Au chargement de l'historique, la base de données contenant les données de l'édition est passée en paramètre. Dans le cas de la création d'un historique, une nouvelle base de données vide sera donnée.

Avant d'exécuter la commande d'initialisation à partir de la Session créée, son CommandType est enregistré. À partir de là, la View peut être utilisée pour lire des données pour l'interface utilisateur et peut exécuter des Commands pour les modifier.

3.3.2 Choix d'implémentation

Le langage Rust a été choisi pour l'implémentation de ReTracer pour plusieurs raisons. Tout d'abord, la librairie doit être utilisable par des logiciels différents, pour pouvoir étudier l'utilisation du modèle dans différents contextes. Les librairies Rust

```
class InitProject extends Command {

  commandType() {
    return "InitProject";
  }

  arguments() {
    return Arguments.new();
  }

  run(view: View) {
    view.recordElementType({
      name: "Counter",
      inherits: [],
      attributes: [],
    })

    view.recordCommandType({
      name: "Increment",
      inherits: [],
      arguments: [("counter", "Counter")],
    });

    var counterId = view.newElement("Counter");
    var counterState = view.newState(counterId);
    counterState.write(0);
    counterState.recordState();
    view.moveTo(counterState);
    view.recordState();
  }
}
```

Listing 3.2 – Exemple de définition d'une commande d'initialisation de l'historique

peuvent être utilisées par d'autres langages à travers l'interface de fonctions externes (FFI) du langage C, qui est très utilisée. De plus, il est possible de les compiler en WASM, ce qui permet de les utiliser dans des projets web. Cela sera utile pour mener des études des usages de l'historique par des utilisateurs, et en particulier pour observer comment leur processus d'édition peut bénéficier de cet historique. Les outils web simplifient la mise en place de l'environnement d'expérience à distance, d'où son importance dans le choix du langage d'implémentation de la librairie. Rust possède des outils de gestion de

```
var history = retracer.new(Database.new());
var view = history.newSession();

view.recordCommandType({
  name: "InitProject",
  inherits: [],
  arguments: [],
});

view.run(InitProject.new());
```

Listing 3.3 – Exemple d'initialisation de l'historique

la compilation en WASM qui nécessitent moins de mise en place que ceux que j'ai trouvé pour C et C++, ce qui permet de passer plus de temps sur le développement du projet que son intégration. De plus, une des pistes d'étude est l'intégration de l'historique aux langages ou aux systèmes d'exploitation par exemple, et Rust se prête plus à cet exercice que Typescript à ma connaissance.

La librairie est disponible sur le dépôt git `https://gitlab.inria.fr/pschmid/retracer`, ou dans le registre de librairies Rust `https://crates.io/crates/retracer`. Son développement est encore en cours au moment de la rédaction du manuscrit, mais cette section décrit son fonctionnement une fois complétée. La documentation de la librairie apporte plus de détails quant aux structures qui la composent et des exemples d'utilisation.

3.3.3 Outils prédéfinis

La librairie suit la segmentation de ReTracer en deux parties : une partie historique enregistrant la structure et une partie interface se chargeant de l'interaction entre historique et logiciel. Ces deux parties suivent les définitions de ReTracer, en conservant néanmoins les définitions de la sérialisation de l'historique, l'interface pour la base de données et le mode contrôle de l'historique dans des parties à part, ces parties ayant été ajoutées progressivement, construites autour des deux parties centrales. Une sixième partie contient des modules prêts à l'emploi, qui donnent des exemples d'implémentation des fonctionnalités discutées dans cette thèse, et montrent plus amplement les capacités du modèle ESCI et de ReTracer.

Des Commands d'historique telles qu'Undo_M, Redo et Undo_P sont prédéfinies. Undo_M et Undo_P prennent en Argument le nombre de Changes à annuler, et Redo prend le nombre de Changes à rejouer. Différentes versions de SelectiveUndo et SelectiveRedo sont aussi disponibles, prenant en paramètre des ChangeReferences vers les changements à annuler ou ré-exécuter. Comme ces Commands sont amenées à exécuter d'autres Commands, elles prennent également une command_factory en Argument pour obtenir à partir d'un CommandTypeName un objet Command correspondant. Des Commands avancées annulant partiellement des Changes sont aussi proposées, ce qui permet de contrôler la granularité de l'annulation et ne pas forcément prendre les Changes comme l'unité d'annulation. D'autres Commands sont également fournies, comme des Commands permettant de dupliquer l'état courant d'un Element et de fusionner deux états d'Elements à des fins d'exploration d'idées, des Commands d'aide à l'initialisation de l'historique et une implémentation d'une Command Ellipse.

Des fonctions de résolution de Paradoxes prenant un Change en entrée et résolvant les Paradoxes en suivant une stratégie de la littérature sont également disponibles, utilisables par une Command pour résoudre les Paradoxes de ses enfants par exemple.

La base de données doit être fournie par le développeur, car contrairement à l'historique qui ne connaît pas les spécificités des données éditées, il peut choisir une stratégie de conservation des données adaptée à leur nature. Quelques exemples simples de structures de données sont prédéfinies, comme une HashMap par exemple. La HashMap lie aux StateReferences un snapshot entier des données. Une autre approche qui limite les duplicatas est aussi proposée. Si deux états se trouvent posséder les mêmes données, une seule copie sera conservée et les References des deux états pointeront vers cette même copie. Une implémentation de la stratégie empruntée par la plupart des historiques actuels est également proposée. Les Commands doivent alors posséder en plus des autres méthodes une méthode "undo" inversant leurs effets et n'enregistrant que l'état présent et une liste de données nécessaires aux méthodes "undo". Pour toute Command ne définissant pas de méthode "undo", l'implémentation propose un comportement par défaut qui enregistre l'état entier, auquel cas "undo" ne sera qu'un rechargement de ce snapshot.

La sérialisation de l'historique permet de l'enregistrer dans un fichier, de le recharger ultérieurement et de le partager avec d'autres utilisateurs. Le format utilisé n'est pas imposé, et peut être textuel ou binaire. La sérialisation de la base de données est laissée

au logiciel.

La synchronisation des historiques se fait à l'aide d'observateurs déclenchés à l'exécutions d'une Command. Quelques observateurs sont donnés en exemple pour montrer comment un système OT ou CRDT peut y être intégré. Comme le fait Sun avec le framework REDUCE [Sun00] et Beaudouin-Lafon et coll [BK93], le système OT ou CRDT gérant la synchronisation des logiciels est séparé de l'historique et interagit avec.

Des Commands de synchronisation utilisant la fonctionnalité de synchronisation des Sessions sont aussi présentées.

Le langage de description de régions doit également être adapté au type de données éditées, et ReTracer n'impose pas de contrainte quant à son choix. Quelques exemples simples sont donnés pour un canevas de dessin numérique et un document texte.

D'autres outils d'aide à l'utilisation de ReTracer sont aussi fournis, comme des macros ou des structures d'enveloppe facilitant la manipulation de données.

CHAPITRE 4

Discussion : Validation du modèle ESCI et de l'architecture ReTracer

Le modèle d'historique de commandes ESCI permet d'unifier les fonctionnalités des modèles de la littérature, et l'architecture logicielle ReTracer permet d'implémenter un historique suivant ESCI. La thèse a principalement porté sur leur création, et le développement de la librairie ReTracer-rs a démarré la phase de validation expérimentale d'ESCI et de ReTracer. Une première version de la librairie est en cours de finalisation, et permettra de mettre en place des expériences pour continuer à développer le projet. Le développement d'un logiciel de dessin matriciel est également en cours, et sera un

premier support pour valider différents aspects d'ESCI et de ReTracer.

Bien qu'il soit difficile de valider l'utilité et l'utilisabilité d'ESCI et de ReTracer à ce stade, leur spécification permet de commencer une première validation théorique de leur utilisabilité. La première section de ce chapitre portera sur l'intégration de ReTracer dans un logiciel de développement, et la seconde présentera les apports de ReTracer au travail d'édition effectué par l'utilisateur. La troisième section s'attardera sur la nécessité de développer de nouvelles interfaces utilisateur pour interagir avec des historiques de commandes avancés comme ReTracer-rs, ainsi que des pistes de développement.

4.1 Modularité et intégration d'un historique ReTracer à un logiciel d'édition

L'intégration de ReTracer dans un logiciel dépend de l'architecture de ce dernier. Pour pouvoir s'adapter à cette architecture, les historiques ReTracer possèdent une structure modulaire, rendue possible par la flexibilité du modèle ESCI.

4.1.1 Modularité d'ESCI et de ReTracer

Le modèle ESCI possède les six propriétés permettant de combiner les caractéristiques des historiques de la littérature en un modèle. Ce modèle est alors capable de proposer les fonctionnalités de ces modèles de manière unifiée. Un historique suivant le modèle ESCI peut ainsi implémenter l'annulation (Undo) et la ré-exécution (Redo et Repeat) séquentielle, sélective et régionale, qu'elles soient méta ou primitives, et que l'on travaille seul, que l'on soit dans un contexte collaboratif asynchrone ou temps réel, entre instances du même logiciel ou de logiciels différents. Et au lieu de proposer une implémentation de ces fonctionnalités, le modèle fourni des outils permettant d'exécuter des commandes, de parcourir l'historique et de réutiliser ses données. Les fonctionnalités prennent la forme de commandes usant de ces outils. Ajouter une fonctionnalité revient à définir une nouvelle commande, qui peut utiliser les outils de l'historique ainsi que les fonctionnalités déjà définies. Cela permet d'adapter les fonctionnalités de l'historique aux cas d'usage du logiciel.

ESCI n'impose pas de contraintes quant à la nature des données éditées, et permet de gérer librement les paradoxes et la synchronisation des historiques. La gestion des données du logiciel, la définition des commandes, la description de régions, la résolution de paradoxes et la synchronisation des historiques se fait indépendamment

de l'historique. ReTracer propose une interface de communication avec l'historique pour chacune de ces tâches. Le but de cette modularité est de séparer ces responsabilités de l'historique pour pouvoir choisir la stratégie la plus adaptée à une situation sans impacter le fonctionnement de l'historique. Le logiciel peut ainsi choisir une base de données adaptée aux données éditées, et si besoin est en changer, sans avoir à changer l'historique ou la gestion des autres tâches. Cela permet en plus de créer une banque de modules prédéfinis. Séparer la gestion de ces tâches de l'historique et la déléguer au logiciel permet de lui rendre la main, ce qui permet de limiter les contraintes imposées à l'architecture du logiciel pour pouvoir intégrer l'historique.

4.1.2 Intégration d'un historique ReTracer

Bien que l'intégration de ReTracer à des architectures variées n'a pas encore été étudiée en profondeur, on peut déduire des besoins d'un historique et du fonctionnement de ReTracer certaines manières de procéder, ainsi que les potentielles contraintes que l'on peut rencontrer.

Les historiques ont besoin d'être informés des opérations effectuées lors de l'édition. Ils imposent au logiciel une séquentialisation et une discrétisation des opérations d'édition, généralement sous la forme de commandes.

Remplacement d'un historique par ReTracer

Les logiciels existants possèdant déjà cette notion de commandes, l'historique ReTracer peut les réutiliser. Quelques ajouts sont à effectuer pour que l'historique soit fonctionnel. Chaque commande a un type qu'il faudra renseigner à ReTracer au travers des CommandTypes. Cette notion existe souvent déjà dans les logiciels, au travers d'un type énumérant tous les types de commandes existants. À minima, il faudra aussi renseigner un ElementType et créer un Element qui représentera l'ensemble des données éditées. Il en va de même pour les données d'entrée, pour lesquelles sera renseigné un InputType. La librairie propose une fonction à cet effet dans les modules complémentaires. Quelques lignes de code doivent être adaptées, comme l'initialisation de l'historique et les appels aux commandes qui doivent se faire à l'aide de la View. Ces modifications n'apportent pas de nouveau concept et ne font qu'adapter les structures existantes à ReTracer. La seule modification majeure est l'utilisation de la View dans les commandes comme interface pour la lecture et la modification de données. À minima, elles liront leurs arguments et l'état du projet depuis la View au début, et y enregistreront chaque donnée d'entrée à leur réception et

les modifications effectuées à la fin de l'exécution. Il est à noter que certains logiciels centralisent déjà partiellement ou entièrement leurs sources de données, au moyen de la notion de Context par exemple [Reaa ; Blec]. Une base de données prédéfinie conservant le système de recalcul des données de l'historique existant peut être choisie, comme l'approche classique d'enregistrer les commandes exécutées et d'utiliser leur méthode d'annulation. Finalement, les appels à "undo" seront convertis en une exécution de la commande Undo_{M} prédéfinie. Au prix de ces quelques adaptations, le logiciel peut bénéficier des nombreuses fonctionnalités de ReTracer, comme les différents types d'annulations et de ré-exécution, le parcours de l'historique persistant et sérialisable, la collaboration synchrone ou non et le partage de données de l'historique. Il possède alors aussi des notions lui permettant d'ajouter plus de granularité à l'édition, en décrivant et combinant des Elements entre eux, et idem pour les Commands. Cette description de la structure des données et des opérations viendra enrichir le parcours de l'historique et la réutilisation de ses données, dont l'utilité sera développée dans la prochaine section.

La modularité de ReTracer permet également de changer la manière de gérer les données sans que cela n'impacte le reste, de rendre le logiciel collaboratif sans avoir à changer l'historique et les systèmes qui en dépendent, de choisir la manière de gérer un paradoxe au moment où il survient, et d'ajouter d'autres commandes d'historique sans qu'elles n'entrent en conflit avec les commandes existantes.

Intégration de ReTracer à d'autres architectures logicielle

Si le logiciel ne possède pas déjà de commandes, il est possible d'adapter ReTracer à des notions déjà existantes plutôt que de convertir les opérations existantes en commandes.

Distinction des rôles d'une commande. Les historiques sont souvent implémentés sous la forme d'une liste de commandes exécutées [Qtc]. La commande a alors plusieurs rôles à remplir. Elle identifie une opération d'édition, permet de l'exécuter ainsi que de l'annuler et de la ré-exécuter en général. Pour cela, la commande doit conserver les données nécessaires à cette annulation, comme du texte supprimé par l'exécution de la commande, et celles permettant de la ré-exécuter, c'est-à-dire ses arguments. L'historique est composé de données d'identification des opérations, d'une opération, de données servant à l'exécution de ces opérations et de données concernant la restitution d'états. ReTracer distingue ces différents rôles, suivant le principe de séparation des responsabilités. Les CommandTypes identifient une opération de manière abstraite, l'interface Command permet d'exécuter l'opération concrète correspondante, les Arguments

contiennent les données servant à l'exécution de l'opération, et la base de données conserve les données relatives à la restitution d'états. Chaque composant a un rôle. Les CommandVersions et les ElementVersions composant l'historique ont pour rôle de décrire l'évolution de l'édition au moyen de liens chronologiques et causaux, et relient les composants entre eux. Ces distinctions donnent à l'historique une plus grande flexibilité. Une commande classique peut être recomposée à partir de ces composants, mais ils peuvent également s'adapter à des architectures n'utilisant pas de commandes.

Alternatives aux commandes. Certaines architectures séparent le logiciel en un composant responsable de l'interaction avec l'utilisateur, un composant gérant les opérations à effectuer, et un composant gérant la conservation des données. Souvent, l'interface utilisateur envoie un évènement au composant gérant les opérations, qui manipulera les données du troisième composant. Le rôle exact de chaque composant, le nombre d'instance de chacun de ces composants, leur manière de communiquer et le vocabulaire utilisé pour les désigner, varie entre architecture, mais on peut citer comme exemple Redux, Flux, MVC et MVP.

Si l'on prend le vocabulaire de Redux, une Vue [1] présente à l'utilisateur l'état du logiciel. Pour déclencher une modification de cet état, elle crée un évènement qui génère une Action. Cette Action possède un type identifiant la modification à effectuer, et une charge utile contenant les données nécessaires à l'exécution de la modification. Elle est envoyée à un Store qui est composé de deux parties : les Reducers et le State. Un Reducer n'est autre qu'une fonction qui effectue une modification, et le State est l'ensemble des données du logiciel. Les Reducers prennent l'Action en paramètre, peuvent modifier le State, et appeler à leur tour d'autres Reducers. Une fois la modification terminée, la Vue est mise à jour. Une des caractéristiques de l'architecture Redux est qu'il n'y a qu'une seule Vue et qu'un seul Store. Flux permet par exemple d'avoir plusieurs Stores. Le composant Modèle de MVC et MVP est un équivalent du State. Ces modèles possèdent également la notion de Vue. La gestion des opérations par le troisième composant (Controler, Presenter ou Reducers) et la communication entre chaque composant diffèrent fortement entre modèles.

Un historique ReTracer peut être intégré à un logiciel suivant ces architectures en faisant usage des concepts existants. Dans le cas de Redux, les Actions identifient une opération et contiennent les données servant à son exécution, ce qui correspond respectivement aux CommandTypes et aux Arguments. Les Reducers modifient les

1. View en anglais, mais le terme français sera utilisé pour ne pas le confondre avec la View d'une Session ReTracer

données du logiciel à partir des Actions et peuvent appeler d'autres Reducers, et suivent ainsi le comportement des Commands. Enfin, le State contient les données du logiciel que les Reducers modifient, à la manière de la base de données associée à l'historique [2]. Les concepts de ReTracer permettent d'adapter l'historique à ces notions qui répartissent les rôles différemment que lors de l'utilisation de commandes. L'Action ne possède pas elle même l'opération, et celle-ci est décomposée en plusieurs opérations s'appelant en cascade. ReTracer permet ainsi de s'intégrer à une architecture Redux et proposer toutes ses fonctionnalités tout en conservant les concepts de l'architecture. Les informations dont elle a besoin sont extractable des notions pré-existantes.

La place de l'historique dans les architectures. L'historique ReTracer a besoin d'être informé des opérations à exécuter, à des fins de ré-exécution notamment. Dans le cas de Redux et Flux, il peut s'interposer entre la Vue et les Reducers pour enregistrer les Actions. L'historique ReTracer a également besoin d'être informé du déroulement de l'exécution pour créer les liens chronologiques ainsi que de causalité et détecter les paradoxes. Pour cela, il peut s'interposer entre les Reducers et le State pour enregistrer les lectures et écritures sous la forme de liens TRIP. Enfin, l'historique ReTracer a besoin de pouvoir influer sur le déroulement de l'exécution pour appliquer des Overrides principalement. En interceptant les échanges entre Reducers et State, il peut fournir aux Reducers des données modifiées en fonction des Overrides enregistrés. Pour obtenir les données nécessaires, ReTracer peut théoriquement s'interposer entre chacun des trois composants, sans avoir à modifier leur fonctionnement. En pratique, les implémentations des différentes architectures varient, et la prise en compte de ces variations doit se faire au cas par cas.

Avec Redux, il n'y a qu'une seule View et un seul Store. Flux permet d'avoir plusieurs Stores. Dans le cadre d'un logiciel d'édition, on peut imaginer que chaque Store représente un projet ouvert, ou des parties d'un même projet. Dans le premier cas, un historique ReTracer pourra être attribué pour chaque Store. Dans le second cas, la notion d'Element pourra être utilisée pour représenter les différentes parties du projet gérées par chaque Store. Les architectures MVC et MVP peuvent aussi être utilisées à cet effet, tout comme les Microservices. La structure hiérarchique des Elements peut être utilisée pour s'adapter à une architecture PAC aussi. Il est possible d'associer à chaque instance –qu'il s'agisse de Stores, de triplets MVC, d'agents PAC ou de services–

2. Pour être précis, dans le cas de Redux, les Reducers sont des fonctions pures qui renvoient un nouvel état, sans modifier l'état courant. La librairie Redux.js conserve les anciens états et possède ainsi une fonctionnalité simple de restitution d'état. Ce comportement est reproductible avec ReTracer en utilisant une base de données suivant ce principe d'immuabilité des données enregistrées.

un historique différent, ou d'utiliser un même historique pour toutes les instances, en fonction des besoins. Mais comme l'historique séquentialise les opérations, utiliser un même historique pour chaque instance rompt leur indépendance. Des travaux futurs approfondissant la question de l'intégration d'historiques ReTracer à ces architectures permettront de vérifier et de préciser ces suppositions ainsi que d'apporter des solutions plus concrètes.

Certaines architectures comme Interacto [BJ21] et Amulet [MK96] définissent une place pour l'historique, en précisant comment le reste du logiciel interagit avec celui-ci. Cette interaction est en général basée sur la notion de commandes, ce à quoi peut être adapté un historique ReTracer. Il peut être cependant intéressant de voir si ReTracer peut faire évoluer ces architectures.

Interaction de ReTracer avec d'autres librairies et outils. D'autres parties du logiciel telles que le système de macros ou un système d'assistance [RS96] peuvent également interagir avec l'historique. Ils ont pour cela accès à la View de ReTracer pour parcourir l'historique et y lire des données. Le logiciel peut aussi les autoriser à exécuter des commandes.

ReTracer permet d'éditer l'historique et notamment d'en supprimer des données sensibles ou privées. La conservation des données dans une base de données hors de l'historique permet de le garder neutre. Et cette séparation peut faciliter le développement d'outils permettant le respect de règlements comme le RGPD.

Les librairies et les frameworks proposant des outils pour structurer un logiciel comme Redux.js ou React demandent parfois d'utiliser des méthodes spécifiques pour lire et modifier les données [Red21 ; Reab]. Cela est nécessaire pour qu'ils puissent contrôler ces lectures et écritures. Il en va de même pour ReTracer, qui doit être au courant des exécutions lancées et de leurs effets. Cela soulève la question d'une potentielle intégration de ReTracer à ces librairies et frameworks pour factoriser cette tâche d'écoute des lectures et écritures. Redux.js propose déjà un moyen d'intercepter les Actions entrantes au moyens de Middlewares, et permet aussi de contrôler le Store au moyens d'un Store Enhancer, ce qui permettrait d'y insérer l'historique [Red23b]. Le partage de l'interception des lectures et écritures est encore à étudier.

Intégration de ReTracer à l'environnement du système d'exploitation

La présence croissante de l'édition collaborative au quotidien contraint les logiciels à s'y adapter. Pour cela, deux instances d'un logiciel doivent être capable de communiquer entre elles. De plus, la question de l'interopérabilité est d'actualité et est en passe de

devenir une obligation légale [Eur]. Cela élargit la communication interlogicielle à des instances de logiciels différents. Cette communication doit être basée sur un protocol décrivant l'édition de manière générique, indépendamment des détails d'implémentation de chaque logiciel. Le système de typage abstrait des opérations d'édition et des données éditées de ReTracer pourrait être utilisé à cet effet. Chaque logiciel définirait sa commande correspondant à un CommandType partagé, et idem pour les données décrites par les ElementTypes. L'historique d'édition ne contient pas non plus de données propres à un logiciel, les commandes n'y étant pas enregistrées, tout comme les données qui sont conservées dans une base de données à part. À l'instar du presse-papier qui permet de copier et coller des données entre différents logiciels, l'historique pourrait devenir un outil proposé par l'environnement du système d'exploitation et utilisé par différents logiciels qui auraient chacun leur implémentation des commandes disponibles et leur base de données. Partager de nouvelles données pourrait se faire de la même manière que le presse-papier en utilisant des formats de données répandus.

Intégration de ReTracer au langage de programmation

Une autre piste d'intégration est celle des langages de programmation. Comme l'historique doit être au courant des opérations effectuées, l'intégrer au langage permettrait d'alléger le code nécessaire pour l'utilisation de l'historique dans un logiciel. Cela simplifierait également l'interface de l'historique. Elle serait allégée des fonctions d'enregistrement explicite des écritures et des lectures, ainsi que des demandes d'exécution de commandes. De plus, comme ces écritures et lectures seraient détectées par le langage, l'interface n'aurait plus à être conçue pour qu'aucune lecture ou écriture ne lui échape. Pour l'instant, les Queries permettent de s'en assurer, comme il faut demander un accès en lecture ou en écriture pour pouvoir accéder aux données. Elles servent de façade pour contrôler les accès. Dans le cas d'une intégration au langage de programmation, l'interface pourrait être plus ouverte et peut être même ne plus nécessiter de Queries.

Le typage abstrait proposé par ReTracer pourrait optionnelement être automatiquement généré à partir des définitions de classes, même si ce découplage a des intérêts, comme cela vient d'être présenté.

Une autre piste d'intégration est celle des Elements avec la notion de variables. Au lieu de n'être que la représentation d'une valeur, le langage pourrait la faire évoluer vers ce que sont les Elements dans ReTracer : des entités dont l'état évolue au fil de l'édition. Cela ressemble à l'intégration de la gestion d'états au langages de programmation [Lun+13 ; Lee86]. L'intérêt ici serait d'automatiser le chargement de l'état courant d'un Element

dans une variable, ainsi que l'enregistrement de nouveaux états dans l'historique et des valeurs associées dans la base de données.

4.1.3 Autres usages de ReTracer

Le cœur d'ESCI est un graphe représentant l'évolution de données. Cette structure peut être utile à d'autres outils, en plus des historiques de commandes.

ReTracer en tant que SCV

Les systèmes de contrôle de versions effectuent un travail similaire aux historiques de commandes. À la différence des historiques, ils ne sont pas informés des opérations effectuées, mais uniquement du résultat de la séquence d'opérations exécutées entre deux enregistrements de versions. Comme l'enregistrement est manuel, les données qui constituent les SCV sont une sélection parmi tous les états qui ont existé au cours de l'édition de ceux jugés par l'utilisateur comme utiles à conserver.

Comme les historiques enregistrent tout par défaut, il faut effectuer l'opération opposée et supprimer de l'historique tout état qui n'est pas à conserver. La fonctionnalité d'édition de l'historique de ReTracer peut être utilisée à cet effet. Ce nettoyage peut être effectué pour les données anciennes uniquement, conservant un historique récent complet pour garder une liberté de parcours et de réutilisation.

Il est aussi possible d'utiliser ReTracer pour développer un SCV. Au lieu de lui donner des commandes d'édition, celles-ci correspondront à la palette de commandes d'un SCV, comprenant la fusion, le rebasage, la restitution de versions passées et une commande d'enregistrement de l'état actuel des fichiers suivis. Ce suivi peut être effectué à l'aide de la notion d'Elements. À chaque nouveau fichier et dossier est attribué un nouvel Element, qui l'identifiera dans le SCV. Cela permet de ne restituer l'état que d'une sélection d'Elements par exemple. Les branches sont créées automatiquement à la suite d'une restitution de versions et de l'enregistrement de nouvelles modifications. La fusion de deux branches se fait en fusionnant les dernières versions de chanque branche, ce qui fonctionne de la même manière que la fusion de deux états d'un Element obtenus par exploration que l'on voudrait fusionner. La fusion peut échouer, auquel cas il sera possible de fournir des informations complémentaires pour résoudre les conflits telles que la source à privilégier pour une région spécifique. Le rebasage est quant à lui la ré-application de modifications sur une autre version, ce qui correspond à la fonctionnalité de ré-exécution de Commands. L'échec d'un rebasage est équivalent à un paradoxe, et peut être résolu avec les Overrides. La base de données peut être

adaptée au type de fichiers enregistrés, ce qui permet de gérer des fichiers texte comme des fichiers de n'importe quel autre format. Les Sessions représentent les différentes copies du dépôt, et permettent de travailler en parallèle sur les mêmes données. La synchronisation des dépôts se fait en diffusant les Changes de chaque SCV. On peut ainsi avoir un SCV décentralisé, à la manière de Git ou Pijul [Pija].

Les fonctionnalités d'ESCI sont transférable à un scénario de contrôle de versions, rendant le modèle apte à l'emploi pour créer un SCV possédant la polyvalence de ce modèle.

ReTracer pour d'autres usages

On peut imaginer utiliser ReTracer dans d'autres situations, comme fournir à des algorithmes utilisant le backtracking [GB65] un moyen de l'effectuer. Un autre domaine qui pourrait faire usage de ReTracer est celui du débogage. Depart la position du débogueur situé entre le programme et son environnement d'exécution, il a le contrôle sur les lectures et écritures effectuées par le programme. ReTracer peut en bénéficier et ainsi tracer l'évolution des valeurs prises par les variables du programme et mettre cet historique à disposition du débogueur à des fins de parcours, d'étude de l'évolution des données de variables et de restitutions d'états passés. Plus généralement, toute structure séquentielle ou séquentialisable ayant une notion d'état et de changement peut être représentée par ESCI. Un diaporama peut être enregistré ainsi, chaque état représentant une diapositive, reliés pas des changements. Le fil de déroulement de la présentation sera représenté par une Session, qu'il suffit de rejouer progressivement. Représenter chaque composant des diapositives par un Element permet de les réutiliser facilement dans les diaposisives suivantes. La présentation peut comporter des retours vers une diapositive passée et à partir de là développer un autre point, créant une structure en arborescence des diapositives. Cet exemple n'est qu'une illustration de l'utilisation de la structure ESCI comme structure du document édité.

4.1.4 Validation de l'intégrabilité de ReTracer

Le développement d'un logiciel de dessin matriciel est une première base pour tester l'intégration de ReTracer dans un logiciel d'édition. Dans une démarche exploratoire, le projet a commencé en suivant l'architecture Redux. À partir de cela, il sera possible de décliner le logiciel avec d'autres architectures, en ajoutant plusieurs Stores par exemple comme dans Flux. Pour être capable d'altérer librement l'architecture, la librairie Redux.js n'a finalement pas été utilisée.

4.2 La tâche d'édition avec un historique ESCI

ESCI permet d'implémenter des commandes pour répondre au différents besoins des utilisateurs. L'utilisateur peut consulter les états passés et les commandes exécutées, et restaurer ces états et ré-exécuter ces commandes. Cela lui permet de corriger des erreurs de tout types, allant de la faute de frappe au dessin sur le mauvais calque. L'historique est résilient, car aucune opération de navigation ni d'exécution de commandes ne peut engendrer une perte irrémédiable de données, ce qui n'est pas le cas des historiques linéaires classiques de type "Linear Undo". L'utilisateur garde cependant le contrôle sur les données contenues dans l'historique au travers du mode contrôle d'historique lui permettant entre autres de supprimer toute donnée inutile ou confidentielle. Cela donne à l'utilisateur une plus grande liberté d'édition.

4.2.1 Liberté d'édition

Exploration d'idées. Comme mentionné plus tôt, l'exploration d'idées fait partie du travail de conception. ESCI supporte cette tâche au travers de ses fonctionnalités de parcours libre de l'historique et de la réutilisation de ses données. Il est aussi possible de partager les différents résultats obtenu avec des collaborateurs en partageant des StateRecords. Les résultats sont combinables en un nouvel état.

Ajustement et édition des données d'entrée. À une échelle plus petite, l'exploration de variantes est rendue possible par les mêmes fonctionnalités. L'utilisateur peut exécuter une Command pour voir le résultat qu'il obtient, modifier ses Arguments et ré-exécuter la Command à partir du même état. Cela permet d'ajuster les Arguments pour obtenir le résultat souhaité. Les opérations de restitution et de ré-exécution successives peuvent être automatisées par le logiciel pour que l'utilisateur n'ait pas à s'en charger. L'édition de données d'entrée est également possible. L'enregistrement systématique des données d'entrée permet de réutiliser ces données par la suite ou de créer des Elements à partir des Inputs, ce qui permet d'éditer ces données. Dans le contexte des logiciels de dessin numérique, cela signifie que l'on peut annuler la commande, modifier le trait effectué à l'aide d'outils d'éditions similaires à ceux du dessin vectoriel, changeant la forme du trait, avant de ré-exécuter la commande avec ce trait édité. Et cela peut se faire avec n'importe quelle commande. On peut alors implémenter des commandes propres au dessin vectoriel pour du dessin matriciel. Par exemple, on peut prendre une séquence de commandes, mettre à l'échelle les tracés passés en entrée, puis les annuler et les exécuter avec ces nouveaux tracés plus grands. Cela permet d'agrandir une image sans avoir

d'effet de pixelisation. Les tracés peuvent être manipulés librement, puis appliqués à un calque. Certains outils comme Procreate donnent à l'utilisateur la possibilité d'arrondir le tracé qu'il est en train d'effectuer, mais une fois terminé, il n'est plus éditable [Pro]. Cette fonctionnalité est un indicateur d'une potentielle demande d'opérations de ce genre. La possibilité de mélanger dessin vectoriel et matriciel dans un même projet proposée par de nombreux logiciels montre que l'édition de données d'entrée peut répondre à un besoin existant [Krib ; Phoa].

Suppression des contraintes de correction. La commande Undo est parfois dite addictive, car on risque de vouloir réessayer le tracé d'un trait par exemple pour obtenir un meilleur résultat que celui que l'on a. Certes, le fait de pouvoir le faire peut inciter les utilisateurs à toujours annuler et réessayer. Mais peut-être que la suppression des données au bout d'un moment joue un rôle dans cette motivation, ainsi que l'impossibilité d'annuler sélectivement. En effet, si l'on ne corrige pas le trait maintenant, après il sera trop tard, devant annuler tout ce que l'on a fait depuis, voir même ne pouvant plus annuler la commande du tout, celle-ci ayant atteint le fond de l'historique, et ayant été supprimée pour ne pas dépasser la taille maximale de celui-ci [3]. L'impression de "C'est maintenant ou jamais", "après ce sera trop tard", pousse l'utilisateur à user de la commande, craignant d'être coincé par la suite. L'origine de cette crainte peut varier, la peur d'un résultat "pas assez bon" en étant un [Par22]. L'absence de ce risque de perdre la possibilité de l'annuler par la suite peut aider à ne pas se sentir forcé de corriger tout immédiatement, et ainsi de laisser le temps à l'utilisateur d'évaluer la nécessité d'effectuer une correction.

Changement d'approche de l'édition

Avec ReTracer, le système possède maintenant l'entièreté des états des éléments, générés au cours de l'édition. Le document enregistré n'en est qu'un extrait. Ce document contient alors un seul état de l'artefact, n'enregistrant pas l'état des autres éléments. Certains logiciels comme Blender enregistrent l'état des outils ainsi que de l'interface dans le document, mais ce n'est pas une pratique courante. À la place, d'autres logiciels proposent d'enregistrer les autres données dans des documents à part. Par exemple, les logiciels de dessin numérique permettent d'enregistrer les états des pinceaux pour pouvoir les réutiliser ailleurs et les partager. Tous ces documents peuvent être extraits de l'historique.

3. La fonctionnalité d'annulation est constamment mentionnée comme une différence clef entre dessin numérique et physique [Aca ; The20 ; Ahm22 ; Ana22 ; Cli]

Lors de l'édition, on passe beaucoup de temps à éditer l'artefact. L'état courant de la vue met à disposition les éléments dont on peut avoir besoin pour cette tâche sous la forme d'un arbre d'éléments. Mais parfois, on passe du temps à éditer un outil, comme les propriétés du pinceau par exemple, ou la palette de couleurs. Ces tâches nécessitent parfois d'autres éléments, comme un endroit pour tester le pinceau, ou un outil pour mélanger les couleurs. Certaines interfaces de logiciels s'adaptent alors à ces besoins différents. Le développeur peut alors choisir de modifier la structure de l'arbre de la vue pour n'avoir dans l'état présent que ce qui est nécessaire. Par exemple, si l'interface cache le canevas pour le remplacer par les propriétés du pinceau que l'utilisateur veut éditer, on peut vouloir "zoomer" sur la partie outils de l'arbre. La vue n'est qu'un pointeur vers l'état d'un élément, et peut être déplacée vers n'importe quel état de n'importe quel élément. Cela permet d'adapter la vue à ce qui est nécessaire. C'est un peu comme changer de mode dans une interface modale. Différentes opérations sont proposées par mode.

Comme pour le canevas auquel on peut ajouter et supprimer des calques, il est possible d'ajouter et de supprimer des éléments de la vue. On peut également ajouter à une position différente dans l'arbre un élément qui est déjà dans la vue. Cela permet par exemple de voir le même canevas dans deux états différents, et de les éditer, pour ensuite choisir celui qui nous plait. Nous y reviendrons dans la section sur les interfaces utilisateur.

On peut remarquer que l'élément à la racine de la vue, le Projet_Paysage, est une sorte d'élément conteneur, un environnement d'édition. Il possède d'un côté une boîte à outil, et de l'autre un canevas, mettant ainsi à disposition les outils nécessaires à l'édition du canevas. Si l'on se met à éditer un outil par exemple, on va pouvoir avoir besoin d'autres Elements qui ne sont pas aussi utiles lors de l'édition de l'artefact, et vice versa. Éditer le pinceau demande de pouvoir interagir avec ses paramètres, et de les tester. Un environnement qui y serait adapté serait donc le pinceau à éditer, ainsi qu'une surface de test. On peut alors déplacer la vue de Projet_Paysage vers un autre Element Éditeur_de_Pinceau possèdant en plus du pinceau à éditer un Element surface de test. Il est ainsi possible d'adapter la structure de la vue à l'Element édité.

Apprentissage et enseignement

Les utilisateurs peuvent étudier leur processus d'édition (aussi nommé workflow) à l'aide de ReTracer et le partager avec d'autres pour qu'ils apprennent de nouvelles techniques. Le partage du workflow est assez commun dans le domaine du dessin

numérique et de la sculpture numérique. Ce partage se fait en effectuant une capture d'écran vidéo des sessions d'édition et en publiant une version accélérée de la vidéo. Le partage de l'historique permettrait non seulement de rejouer toutes les sessions de manière chronologique ou causale, mais aussi de pouvoir explorer l'historique soi même, et d'en extraire des séquences de commandes utiles par exemple.

4.2.2 Organisation des données

L'historique de commandes contient bien plus de données que celles d'un document. Il faut donc que l'utilisateur puisse se repérer dans l'historique et organiser ses données selon ses besoins.

Les concepts d'ESCI. Le parcours de l'historique est structuré par les States et les Changes. Le but de ces deux notions d'état et de changement est d'être simple et intuitif. La navigation entre états peut se faire de manière chronologique en suivant les changements d'une Session, ou causale en suivant les liens TRIP. Ces liens contiennent beaucoup d'informations, mais on peut n'utiliser que les liens Target–Result représentant un changement d'état d'un élément. Une fonction de recherche peut aussi recroiser des caractéristiques d'état et de changement pour les retrouver. On peut imaginer une requête demandant tous les états d'un élément créés dans un certain intervalle de temps et générés par des changements d'un certain type.

Les métadonnées. Il est possible d'ajouter des informations aux états et changements au travers de métadonnées. À la manière des fichiers dans un système de fichiers, il est possible de marquer comme favori un état ou un changement, de les marquer comme étant caché, de leur ajouter des tags pour les classifier, ajouter le nom de l'auteur de chaque changement, et toute autre donnée supplémentaire que l'utilisateur juge nécessaire. Regrouper les changements en fonction des tâches de plus haut niveau auxquelles elles appartiennent permet d'ajouter un niveau de granularité à l'historique et sa navigation. Tous les changements ayant contribué au dessin d'un arbre seront marqués du nom de la tâche qu'ils ont permis d'accomplir. Liu et coll. proposent un système pour segmenter automatiquement la session d'édition en tâches [LLM20]. Marquer certains évènements liés au travail mené par l'utilisateur peut également servir de repère. Dans le contexte du développement logiciel, Hauswirth et coll décrivent les différents frises chronologies qui s'entremêlent [HA17]. Par frise ils entendent un déroulement d'évènements associés à un objet ou une tâche. L'édition du code en est une, sa navigation peut être considérée comme une autre frise indépendante,

comme dans Vim où la navigation possède un historique à part [Vimd]. La création et le parcours de versions enregistrées dans un SCV forment également une frise, ainsi que les exécutions de code [HA17]. Pour que l'utilisateur s'y retrouve, on peut enregistrer les commandes d'enregistrement du document, de compilation et de lancement du logiciel à des fins de marquage de ces évènements extérieurs, même si elles ne modifient pas de données. Le repère d'ouverture et de fermeture du fichier est déjà présent au travers des Sessions. Ces repères peuvent s'avérer important pour l'utilisateur, car pour des logiciels d'édition conservant l'historique entre les sessions comme Visual Studio Code [Micb] ou Vim [Vimc], un moyen de désactiver cette persistance a parfois été demandé [Gitc]. Cette demande est en général motivée par l'expression de la perte d'une "origine d'édition". Sans la notion de début de session, les commandes seront annulées jusqu'à arriver à la création du projet. Or l'état du projet au début d'une session est un point de repère pour l'utilisateur qui ouvre le projet et travaille à partir de cet état initial. Les Sessions représentent ces parties de l'historique, et on peut imaginer implémenter une commande d'annulation s'arrêtant au début de la session, ou demandant confirmation auprès de l'utilisateur pour dépasser ce point.

Le mode contrôle de l'historique. L'historique peut être modifié pour correspondre aux besoins de l'utilisateur. Les états intermédiaires jugés inutiles peuvent ainsi être supprimés de l'historique, ne conservant que les version importantes du document. À la manière des SCV, ces versions pourront être marquées d'un identifiant de version pour s'y référer plus tard.

4.2.3 Validation de l'utilité de ReTracer

Des expériences utilisateur pourront être menées avec le logiciel de dessin matriciel développé. Le but est d'identifier les fonctionnalités utiles aux utilisateurs et leur manière de les intégrer à leur processus d'édition.

Le cas du dessin matriciel a été choisi, car comme mentionné précédamment, la sémantique de tracés n'est pas conservée par le canevas. L'ajout de la possibilité d'éditer les données d'entrée permettrait d'éditer le dessin à la manière de dessins vectoriels. De plus, un projet de dessin est composé d'un canevas, de pinceaux et de palettes de couleurs, qui peuvent être édités séparément et réutilisés dans d'autres projets. Une tâche de dessin est un bon scénario pour explorer des idées, et l'annulation d'un trait peut être difficile à effectuer sans l'aide d'une commande prévue à cet effet. Finalement, cela permet aussi d'évaluer l'utilité de ReTracer pour l'apprentissage et l'enseignement

du dessin numérique.

L'évaluation de l'utilité d'un historique pour les utilisateurs doit prendre en compte les capacités de l'historique, mais aussi leur utilisabilité par l'utilisateur au travers de l'interface utilisateur. Il est important que l'utilisateur soit capable d'utiliser efficacement ces fonctionnalités. Il doit pour cela maîtriser l'interface et pouvoir comprendre le comportement des fonctionnalités pour pouvoir anticiper les résultats. Présenter clairement et de manière intuitive l'utilité et le comportement des fonctionnalités mises à disposition de l'utilisateur sans que la complexité de l'historique ne lui nuise est nécessaire pour pouvoir étudier l'utilisation de /rt.

4.3 Interfaces utilisateur adaptées à ESCI

Pour permettre à l'utilisateur de bénéficier des fonctionnalités d'un historique ReTracer présentées dans la section précédente, il faut faire évoluer les interfaces utilisateur d'historiques de commandes classiques. Le couple de boutons Undo–Redo et la liste de commandes de l'historique restreint l'utilisation de l'historique aux seules fonctionnalités d'un historique suivant le modèle "Linear Undo". Voici quelques interfaces existantes et des pistes de développement.

4.3.1 Annulation sélective et ré-exécution

L'annulation sélective demande de pouvoir sélectionner les commandes à annuler et gérer les potentiels paradoxes en résultant.

Annulation sélective. Aquamarine propose une interface permettant de sélectionner les commandes à annuler [Mye+15]. Une fois annulées, elles apparaîtront grisées dans l'historique.

Annulation et réutilisation régionale. L'interface pour l'annulation régionale présentée par Nancel et coll dans le contexte du dessin numérique [NC14] permettrait notamment de visualiser les annulations sélectives et autres manipulations de commandes, ainsi que les paradoxes potentiels. Elle permettrait également la sélection de commandes à partir de régions du calque qu'elles ont modifiées. L'interface de micro-versionnage de Mikami et coll permet au moyen d'un menu déroulant de proposer d'anciennes versions du texte à cette position dans le document [MSI17]. Des assistants peuvent utiliser l'historique pour ajuster leurs propositions. L'outil d'auto complétion en est un exemple pour les éditeurs de texte.

Blender 2.7X permettait l'annulation de commandes par objet. Exécuter la command d'annulation n'affectait que les commandes relatives à l'édition du maillage de l'objet en cours d'édition [Bled]. Les historiques de palette sont aussi une sorte d'historique par objet.

Interfaces d'historiques de logiciels de CAO. Les interfaces d'historiques de commandes des logiciels de CAO permettent d'insérer, de modifier et de supprimer des commandes de l'historique, et de corriger les paradoxes s'il y en a de manière interactive [Fus]. Blender ne possède pas ce genre d'interface mais possède une interface d'ajustement des paramètres de la commande qui vient d'être exécutée [Blea].

4.3.2 Consultation et partage de données

L'interface peut proposer d'afficher des vignettes d'états enregistrés dans l'historique. Cela permet de garder une référence ou de comparer ces deux états. Dans le contexte des logiciels d'édition de texte, on peut ainsi avoir un aperçu d'une ancienne version d'une région de code, remplaçant l'usage de commentaires pour les garder dans l'état courant.

Permettre la juxtaposition de deux fenêtres vers des états du canevas différents permet d'effectuer des expérimentations sur les deux, de copier des données de l'un vers l'autre, puis de fermer celles qui ne sont plus utiles. Des interfaces de comparaison de versions ont été proposées, notamment par Yoon et coll [YB15]. Des interfaces existantes peuvent s'adapter à ce genre de cas d'usage. Certains logiciels comme Krita ou Gimp permettent d'ouvrir différents projets dans des fenêtres flottantes. On peut réutiliser ce système pour ouvrir des états du même canevas.

Kim et coll présentent l'historique sous la forme d'une frise chronologique parcourable [KS12].

Les systèmes de partage du document pourraient être étendus au partage de States.

4.3.3 Synchronisation de la collaboration.

Dans DistEdit, l'interface de l'éditeur permet de changer de mode de synchronisation, ce qui permet de contrôler la manière de collaborer avec d'autres utilisateurs [PK94].

Conclusion

Résumé des contributions

Dans le chapitre 1, j'ai décrit la tâche d'édition, contexte d'étude de prédilection des historiques de commandes, donnant à l'utilisateur le contrôle sur l'état des données à éditer. L'historique enregistre une trace d'exécution des commandes et procure au travers des fonctionnalités un moyen de réutiliser ces données. Ces fonctionnalités peuvent être l'annulation ou la ré-exécution de commandes, séquentiellement ou sélectivement. Elles permettent d'annuler les effets de commandes exécutées par le passé, de les ré-appliquer, de répéter l'exécution de commandes ou encore de restituer des états passés. Les fonctionnalités sélectives posent également la question de la gestion des paradoxes qui peuvent survenir. Différentes structures d'historique existent pour proposer une sélection ou l'intégralité de ces fonctionnalités, les résultats de l'utilisation d'une telle fonctionnalité pouvant varier de modèle en modèle. J'ai décrit plusieurs caractéristiques de structure d'historique changeant le comportement de ces fonctionnalités, menant à une catégorisation des modèles. Les modèles peuvent être Primitif ou Méta, choisissant d'enregistrer ou non une trace de l'exécution des commandes d'historique. L'annulation et la ré-exécution peuvent être séquentielles ou sélectives. L'annulation peut se faire soit par reconstruction, soit par inversion. Les paradoxes peuvent être gérés de plusieurs manières, dont la stratégie d'annulation en cascade, l'abandon et l'intervention du

logiciel ou de l'utilisateur pour changer les arguments de la commande paradoxale. Les données concernées par un changement peuvent être décrites plus ou moins précisément, en décomposant ou non l'état en sous-parties et en utilisant ou non un système de description de régions impactées. La trace d'exécution peut être conservée dans son intégralité ou uniquement partiellement. Finalement, l'historique peut être adapté à l'édition collaborative ou non. Cette caractérisation des structures permet de relier les différentes notions trouvées dans la littérature, et permet d'expliquer les problèmes de compatibilité entre modèles. L'utilisateur bénéficierait cependant d'une unification de ces modèles, lui permettant ainsi d'utiliser toutes les fonctionnalités et de choisir le comportement adapté à la situation. Au cours de ses tâches d'édition, il peut rencontrer des moments d'expérimentation et commettre des erreurs, et la capacité de restituer tout état passé ainsi que celle de pouvoir réutiliser les commandes exécutées par le passé aident l'utilisateur à gérer ces situations. De plus, la popularisation de l'édition collaborative et le développement de l'interopérabilité demande d'y adapter les historiques. Le chapitre termine par la description de six propriétés qui permettent à un modèle d'historique d'unifier les autres modèles et ainsi de proposer leurs fonctionnalités.

Dans le chapitre 2, j'ai présenté le modèle d'historique de commandes ESCI qui a été développé pour posséder ces propriétés. Le modèle décrit de manière précise la trace d'édition à l'aide de quatre entités, les Elements, les Sessions, les Commands et les Inputs. Les Elements représentent ce qui est édité, et les Commands les outils pour mener la tâche d'édition au cours d'une Session d'édition. L'exécution de Commands génère des changements (Changes) d'état (States) des Elements. Pour cela, les Commands prennent en Argument des données déjà enregistrées dans l'historique ou de nouvelles données (Inputs). Les liens TRIP décrivent les relations entre ces entités. Le modèle peut être considéré comme hybride, permettant de reproduire le fonctionnement des historiques Primitifs et Méta, car il conserve les liens chronologiques et de causalité. L'annulation et la ré-exécution peuvent être séquentielles comme sélectives. L'annulation peut se faire par reconstruction ou par inversion. Les paradoxes potentiels peuvent être gérés à l'aide de stratégies de résolution. Les données éditées sont décomposées en Elements, et les régions concernées par un changement d'état peuvent être décrites précisément. Le modèle ESCI peut être utilisé dans un contexte d'édition collaborative, qu'elle soit en temps réel ou asynchrone. Le modèle n'impose pas de stratégie de résolution de paradoxe, ni de langage de description de régions, et permet ainsi au logiciel de les définir et de les choisir librement. La trace d'exécution est conservée dans son intégralité, n'imposant ainsi aucune perte automatique de données au logiciel et à l'utilisateur. Le mode contrôle d'historique permet cependant d'effectuer des opérations destructrices

sur l'historique, notamment pour en supprimer des données privées. Cette flexibilité du modèle lui donne des propriétés intéressantes. L'édition est décrite selon une granularité et un cadre défini par le logiciel. Le logiciel peut à tout moment consulter les données de l'historique. Il contrôle également leur suppression de l'historique qui conserve à tout moment un état cohérent. Le logiciel, et par extension l'utilisateur, a ainsi un contrôle important sur les données de l'historique, historique qui lui propose une variété de fonctionnalités qui permettent de répondre aux besoins de l'utilisateur.

Dans le chapitre 3, j'ai présenté l'architecture ReTracer qui propose une manière d'implémenter le modèle ESCI. Différents aspects du modèle ont été approfondis, et des considérations techniques ont été discutées.

Dans le chapitre 4, j'ai discuté de l'intégration d'un historique ReTracer dans les logiciels d'édition et des apports pour l'utilisateur d'un tel logiciel, et la conception d'interfaces utilisateur adaptées. Outre la liberté d'expérimenter et de créer des macros de commandes exécutées par le passé, l'utilisateur bénéficie avec un tel historique d'une plus grande liberté d'édition, ne craignant pas de perte de données ou de capacité d'annulation. La trace d'exécution peut être pour lui une ressource d'amélioration et de partage de son processus d'édition, plus précise et réutilisable qu'une vidéo accélérée.

Au delà des contextes d'édition

La notion de restitution d'un état passé peut se retrouver ailleurs dans les tâches informatiques, comme le débogage de programmes ou dans la gestion de versions de projets. La question de la position des historiques peut alors se poser, ainsi que celle de leur interaction. Est-ce que, de la même manière qu'il y a un outil de presse papier permettant de gérer de manière centralisée les actions de copie et collage de données, il peut y avoir un outil de gestion d'historiques centralisé ? Cela permettrait une coordination et un transfert de données d'historique entre logiciels.

L'intégration de l'historique au niveau du langage de programmation pourrait aider à simplifier l'intégration d'historiques dans les logiciels.

Projets à venir

Le projet est maintenant arrivé à un état qui permet d'étudier son utilisation.

Étude de l'intégration d'un historique ReTracer dans un logiciel d'édition

Une première piste de recherche est l'étude de l'intégration d'un historique ReTracer aux différentes architectures logicielles existantes. Dans un premier temps, il s'agit de développer un logiciel possédant un tel historique. Cela permettra de valider concrètement son intégrabilité. Dans un second temps, il faudra identifier les facilités et les freins à l'intégration d'un historique ReTracer à une variété d'architectures logicielles. L'intégration de l'historique à des langages de programmation, à l'environnement du système d'exploitation ainsi qu'à d'autres outils fera aussi l'objet d'études. Ces études pourront déboucher sur le développement d'autres prototypes intégrant un historique ReTracer, ou des outils facilitant son intégration pour un environnement.

Le but de ces études est de valider l'architecture ReTracer, et de développer des outils permettant d'aider son intégration à des projets.

Développement d'une interface graphique pour un historique ReTracer et étude de l'utilité de ses fonctionnalités

Une seconde piste de recherche porte sur l'interaction avec l'utilisateur. Du fait qu'un historique ReTracer possède les fonctionnalités d'autres modèles, l'interface utilisateur adaptée à ReTracer doit également posséder cette polyvalence. Les interfaces utilisateur proposées par les modèles de la littérature seront d'abord étudiées. Cette étude débouchera sur le développement d'une interface adaptée à la polyvalence de ReTracer. Une expérience utilitateur permettra de valider l'utilité des fonctionnalités de ReTracer ainsi que l'utilisabilité de l'interface utilisateur. Des étapes d'itération pourront avoir lieu pour améliorer l'interface utilisateur. L'interface utilisateur validée pourra alors être adaptée à d'autres types d'édition. Cela permettra d'étudier les différences d'utilité de ReTracer entre les différents type d'édition, dont la saisie de texte, de code, le dessin matriciel et le dessin vectoriel.

Bibliographie

[Aca] AcademyOfFineArt. *Differences between digital and traditional drawing | Learn to draw and paint in the Academy of Fine Art Germany*. URL : https://academy-of-fine-art.com/en/2019/07/11/differences-between-digital-and-traditional-drawing/.

[ACS84] James E. Archer, Richard Conway et Fred B. Schneider. « User Recovery and Reversal in Interactive Systems ». In : *ACM Transactions on Programming Languages and Systems* 6.1 (1 1er jan. 1984), p. 1-19. ISSN : 01640925. DOI : 10.1145/357233.357234. URL : http://portal.acm.org/citation.cfm?doid=357233.357234.

[AD92] Gregory D Abowd et Alan J Dix. « Giving Undo Attention ». In : *Interacting with Computers* 4.3 (3 1er déc. 1992), p. 317-342. ISSN : 0953-5438. DOI : 10.1016/0953-5438(92)90021-7. URL : http://www.sciencedirect.com/science/article/pii/0953543892900217.

[Adoa] Adobe. *Photoshop : Snapshots*. URL : https://helpx.adobe.com/content/help/en/photoshop/using/undo-history.html.

[Adob] Adobe. *Photoshop : Undo and History*. URL : https://helpx.adobe.com/content/help/en/photoshop/using/undo-history.html.

[Ahm+11] Mehdi Ahmed-Nacer et al. « Evaluating CRDTs for Real-Time Document Editing ». In : *Proceedings of the 11th ACM Symposium on Document Engineering*. DocEng '11. New York, NY, USA : Association for Computing Machinery, 19 sept. 2011, p. 103-112. ISBN : 978-1-4503-0863-2. DOI : 10.1145/2034691.2034717. URL : https://doi.org/10.1145/2034691.2034717.

[Ahm22] Ibrahim Ahmed. *Should You Learn to Draw Traditionally or Digitally ?* Mediacaterer. 1er avr. 2022. URL : https://mediacaterer.com/draw-traditionally-or-digitally/.

[Ana22] AnatomyOfASketch. *Is It Good to Learn How to Draw Traditionally before Drawing Digitally ?* Anatomy of a Sketch. 9 sept. 2022. URL : https://anatomyofasketch.com/is-it-good-to-learn-how-to-draw-traditionally-before-drawing-digitally/.

[Bat20] Alexandre Battut. « Substrats et Instruments Interactifs Pour Les Médias Temporels ». These En Préparation. université Paris-Saclay, 2020. (Visité le 05/09/2023).

[Bea23] Michel Beaudouin-Lafon. *Au-delà des applications : Substrats et instruments d'interaction*. Mars 2023. (Visité le 03/08/2023).

[Ber94] Thomas Berlage. « A Selective Undo Mechanism for Graphical User Interfaces Based on Command Objects ». In : *ACM Trans. Comput.-Hum. Interact.* 1.3 (3 sept. 1994), p. 269-294. issn : 1073-0516. doi : 10.1145/196699.196721. url : http://doi.acm.org/10.1145/196699.196721.

[BG] Thomas Berlage et Andreas Genau. « From Undo to Multi-User Applications ». In : (), p. 12.

[BG81] Philip A. Bernstein et Nathan Goodman. « Concurrency Control in Distributed Database Systems ». In : *ACM Computing Surveys* 13.2 (juin 1981), p. 185-221. issn : 0360-0300. doi : 10.1145/356842.356846. (Visité le 27/07/2023).

[BG93a] Thomas Berlage et Andreas Genau. « A Framework for Shared Applications with a Replicated Architecture ». In : *Proceedings of the 6th Annual ACM Symposium on User Interface Software and Technology*. UIST '93. New York, NY, USA : Association for Computing Machinery, 1er déc. 1993, p. 249-257. isbn : 978-0-89791-628-8. doi : 10.1145/168642.168668. url : https://doi.org/10.1145/168642.168668.

[BG93b] Thomas Berlage et Andreas Genau. « From Undo to Multi-User Applications ». In : (sept. 1993), p. 12.

[Bha21] Gurjot Singh Bhatti. « VERSIONING IN INTERACTIVE SYSTEMS ». 2021.

[BJ21] Arnaud Blouin et Jean-Marc Jézéquel. *Interacto : A Modern User Interaction Processing Model*. Mai 2021.

[BK93] Michael Beaudouin-Lafon et Alain Karsenty. « An Architecture for Real-Time Groupware (Abstract) ». In : *ACM SIGOIS Bulletin* 13.4 (avr. 1993), p. 21. issn : 0894-0819. doi : 10.1145/152716.152733. (Visité le 20/07/2023).

[Blea] Blender. *Adjust Last — Blender Manual*. url : https://docs.blender.org/manual/en/latest/interface/undo_redo.html?highlight=repeat#adjust-last-operation.

[Bleb] Blender. *Blender UndoStep*. url : https://projects.blender.org/blender/blender (visité le 05/09/2023).

[Blec] Blender. *Source/Architecture/Context - Blender Developer Wiki*. url : https://wiki.blender.org/wiki/Source/Architecture/Context (visité le 05/09/2023).

[Bled] Blender. *Undo & Redo — Blender Manual*. url : https://docs.blender.org/manual/en/2.79/interface/undo_redo.html.

[Blee] Blender Foundation. *Blender : Previous Versions*. blender.org. url : https://www.blender.org/download/previous-versions/ (visité le 09/01/2023).

[BP02] Aaron B. Brown et David A. Patterson. « Rewind, Repair, Replay : Three R's to Dependability ». In : *Proceedings of the 10th Workshop on ACM SIGOPS European Workshop* (Saint-Emilion, France). EW 10. New York, NY, USA : ACM, 2002, p. 70-77. doi : 10.1145/1133373.1133387. url : http://doi.acm.org/10.1145/1133373.1133387.

[BR00] Cary Lee Bates et Jeffrey Michael Ryan. « Method and System for Undoing Edits within Selected Portion of Electronic Documents ». Brev. amér. 6108668A. International Business Machines Corp. 22 août 2000. url : https://patents.google.com/patent/US6108668A/en.

[Bue+11] Carlo Bueno et al. « Rewriting History : More Power to Creative People ». In : *Proceedings of the 23rd Australian Computer-Human Interaction Conference* (Canberra, Australia). OzCHI '11. New York, NY, USA : ACM, 2011, p. 62-71. isbn : 978-1-4503-1090-1. doi : 10.1145/2071536.2071545. url : http://doi.acm.org/10.1145/2071536.2071545.

[CF] Aaron G Cass et Chris S T Fernandes. « Using Task Models for Cascading Selective Undo ». In : (), p. 16.

[CF06] Aaron G Cass et Chris S T Fernandes. « Using Task Models for Cascading Selective Undo ». In : (oct. 2006), p. 16.

[CFP06] Aaron G. Cass, Chris S. T. Fernandes et Andrew Polidore. « An Empirical Evaluation of Undo Mechanisms ». In : *Proceedings of the 4th Nordic Conference on Human-computer Interaction : Changing Roles* (Oslo, Norway). NordiCHI '06. New York, NY, USA : ACM, 2006, p. 19-27. isbn : 978-1-59593-325-6. doi : 10.1145/1182475.1182478. url : http://doi.acm.org/10.1145/1182475.1182478.

[Che+14] Yuan Cheng et al. « A Multi-User Selective Undo/Redo Approach for Collaborative CAD Systems ». In : *Journal of Computational Design and Engineering* 1.2 (1er avr. 2014), p. 103-115. issn : 2288-4300. doi : 10.7315/JCDE.2014.011. url : https://www.sciencedirect.com/science/article/pii/S2288430014500164.

[Chi+98] Chii Meng et al. « Visualizing Histories for Selective Undo and Redo ». In : *Proceedings. 3rd Asia Pacific Computer Human Interaction (Cat. No.98EX110)*. Proceedings. 3rd Asia Pacific Computer Human Interaction (Cat. No.98EX110). Juill. 1998, p. 459-464. doi : 10.1109/APCHI.1998.704487.

[Cli] ClipStudio. *Pros and Cons of Digital and Traditional Art*. Art Rocket. url : https://www.clipstudio.net/how-to-draw/archives/155248.

[CNR] CNRS. *PEPR exploratoire Ensemble (outils collaboratifs numériques) | CNRS*. url : https://www.cnrs.fr/fr/pepr/pepr-exploratoire-ensemble-outils-collaboratifs-numeriques (visité le 05/09/2023).

[Cop+19] Sven Coppers et al. « Fortunettes : Feedforward about the Future State of GUI Widgets ». In : *Proceedings of the ACM on Human-Computer Interaction* 3.EICS (EICS 13 juin 2019), 20 :1-20 :20. DOI : 10.1145/3331162. URL : https://doi.org/10.1145/3331162.

[CS01] David Chen et Chengzheng Sun. « Undoing Any Operation in Collaborative Graphics Editing Systems ». In : *Proceedings of the 2001 ACM International Conference on Supporting Group Work*. GROUP '01. New York, NY, USA : Association for Computing Machinery, sept. 2001, p. 197-206. ISBN : 978-1-58113-294-6. DOI : 10.1145/500286.500316. (Visité le 19/07/2023).

[Dev] DeviantArt. *Forum : Drawing on the Wrong Layer !* URL : https://www.deviantart.com/forum/community/complaints/2194245/ (visité le 05/09/2023).

[DIL07] Erik D. Demaine, John Iacono et Stefan Langerman. « Retroactive Data Structures ». In : *ACM Trans. Algorithms* 3.2 (2 mai 2007). ISSN : 1549-6325. DOI : 10.1145/1240233.1240236. URL : http://doi.acm.org/10.1145/1240233.1240236.

[Dix91] Alan Dix. *Formal Methods for Interactive Systems*. 1er jan. 1991. ISBN : 978-0-12-218315-7.

[Doc19a] Cory Doctorow. *Adversarial Interoperability*. Oct. 2019. URL : https://www.eff.org/deeplinks/2019/10/adversarial-interoperability (visité le 05/09/2023).

[Doc19b] Cory Doctorow. *Interoperability : Fix the Internet, Not the Tech Companies*. Juill. 2019. URL : https://www.eff.org/deeplinks/2019/07/interoperability-fix-internet-not-tech-companies (visité le 05/09/2023).

[Dri+89] James R. Driscoll et al. « Making Data Structures Persistent ». In : *Journal of Computer and System Sciences* 38.1 (1er fév. 1989), p. 86-124. ISSN : 0022-0000. DOI : 10.1016/0022-0000(89)90034-2. URL : https://www.sciencedirect.com/science/article/pii/0022000089900342.

[Edw+00] W Keith Edwards et al. « A Temporal Model for Multi-Level Undo and Redo ». In : t. 2. 2000, p. 10.

[EG89] C. A. Ellis et S. J. Gibbs. « Concurrency Control in Groupware Systems ». In : *Proceedings of the 1989 ACM SIGMOD International Conference on Management of Data*. SIGMOD '89. New York, NY, USA : Association for Computing Machinery, 1er juin 1989, p. 399-407. ISBN : 978-0-89791-317-1. DOI : 10.1145/67544.66963. URL : https://doi.org/10.1145/67544.66963.

[Eur] Europe. *EU Digital Markets Act and Digital Services Act Explained*. URL : https://web.archive.org/web/20211214110227/https://www.europarl.europa.eu/pdfs/news/expert/2021/12/story/20211209ST019124/20211209ST019124%5C_en.pdf (visité le 05/09/2023).

[FC00] Jonathan D. Fouss et Kai H. Chang. « Classifying Groupware ». In : *Proceedings of the 38th Annual on Southeast Regional Conference*. ACM-SE 38. New York, NY, USA : Association for Computing Machinery, avr. 2000, p. 117-124. isbn : 978-1-58113-250-2. doi : 10.1145/1127716.1127744. (Visité le 19/07/2023).

[Fek96] Jean-Daniel Fekete. « Les trois services du noyau sémantique indispensables a l'IHM ». In : (1996).

[Flo67] Robert W. Floyd. « Nondeterministic Algorithms ». In : *Journal of the ACM* 14.4 (1er oct. 1967), p. 636-644. issn : 0004-5411. doi : 10.1145/321420.321422. url : https://doi.org/10.1145/321420.321422.

[Fou] Blender Foundation. *Multiple Object Editing*. (Visité le 05/09/2023).

[Fus] Fusion360. *Fusion360 : Resolving Timeline Warning or Errors in Fusion 360 | Fusion 360 | Autodesk Knowledge Network*. url : https://knowledge.autodesk.com/support/fusion-360/troubleshooting/caas/sfdcarticles/sfdcarticles/Resolving-Timeline-Warning-or-Errors-in-Fusion-360.html.

[GB65] Solomon W. Golomb et Leonard D. Baumert. « Backtrack Programming ». In : *Journal of the ACM* 12.4 (oct. 1965), p. 516-524. issn : 0004-5411. doi : 10.1145/321296.321300. (Visité le 08/01/2023).

[Gima] Gimp. *Gimp Documentation : 13. Presets*. url : https://docs.gimp.org/en/gimp-tools-presets.html (visité le 05/09/2023).

[Gimb] Gimp. *Gimp Documentation : 2. Common Features*. url : https://docs.gimp.org/en/gimp-filters-common.html (visité le 05/09/2023).

[Gita] Git. *Git - Basic Branching and Merging*. url : https://git-scm.com/book/en/v2/Git-Branching-Basic-Branching-and-Merging (visité le 05/09/2023).

[Gitb] Git. *Git - Rebasing*. url : https://git-scm.com/book/en/v2/Git-Branching-Rebasing (visité le 05/09/2023).

[Gitc] microsoft/vscode Github. *Settings for Disabled Persisted Undo · Issue #94778 · Microsoft/Vscode*. GitHub. url : https://github.com/microsoft/vscode/issues/94778 (visité le 21/03/2023).

[GLL84] Robert F Gordon, George B Leeman et Clayton H Lewis. *CONCEPTS AND IMPLICATIONS OF INTERACTIVE RECOVERY*. Research Report 47293. IBM, 6 avr. 1984, p. 45.

[GM94] Saul Greenberg et David Marwood. « Real Time Groupware as a Distributed System : Concurrency Control and Its Effect on the Interface ». In : *Proceedings of the 1994 ACM Conference on Computer Supported Cooperative Work*. CSCW '94. New York, NY, USA : Association for Computing Machinery, 22 oct. 1994, p. 207-217. isbn : 978-0-89791-689-9. doi : 10.1145/192844.193011. url : https://doi.org/10.1145/192844.193011.

[GMF10] Tovi Grossman, Justin Matejka et George Fitzmaurice. « Chronicle : Capture, Exploration, and Playback of Document Workflow Histories ». In : *Proceedings of the 23Nd Annual ACM Symposium on User Interface Software and Technology* (New York, New York, USA). UIST '10. New York, NY, USA : ACM, 2010, p. 143-152. ISBN : 978-1-4503-0271-5. DOI : 10.1145/1866029.1866054. URL : http://doi.acm.org/10.1145/1866029.1866054.

[GNU] GNU. *Emacs : 20.4 Minibuffer History | Emacs Docs*. URL : https://emacsdocs.org/docs/elisp/Minibuffer-History.

[GWH02] John Grundy, Xing Wang et John Hosking. « Building Multi-Device, Component-Based, Thin-Client Groupware : Issues and Experiences ». In : *Proceedings of the Third Australasian Conference on User Interfaces - Volume 7*. AUIC '02. AUS : Australian Computer Society, Inc., jan. 2002, p. 71-80. ISBN : 978-0-909925-85-7. (Visité le 19/07/2023).

[HA17] Matthias Hauswirth et Mohammad Reza Azadmanesh. « The Entangled Strands of Time in Software Development ». In : *Proceedings of the 3rd ACM SIGPLAN International Workshop on Programming Experience*. PX/17.2. New York, NY, USA : Association for Computing Machinery, 22 oct. 2017, p. 11-16. ISBN : 978-1-4503-5522-3. DOI : 10.1145/3167107. URL : https://doi.org/10.1145/3167107.

[Ham+06] Mark Hamburg et al. « Maintaining Document State History ». Brev. amér. 7062497B2. Adobe Inc. 13 juin 2006. URL : https://patents.google.com/patent/US7062497B2/en.

[Har+08] Björn Hartmann et al. « Design as Exploration : Creating Interface Alternatives through Parallel Authoring and Runtime Tuning ». In : *Proceedings of the 21st Annual ACM Symposium on User Interface Software and Technology*. UIST '08. New York, NY, USA : Association for Computing Machinery, 19 oct. 2008, p. 91-100. ISBN : 978-1-59593-975-3. DOI : 10.1145/1449715.1449732. URL : https://doi.org/10.1145/1449715.1449732.

[Hee+08] Jeffrey Heer et al. « Graphical Histories for Visualization : Supporting Analysis , Communication, and Evaluation ». In : *IEEE transactions on visualization and computer graphics* 14 (1er nov. 2008), p. 1189-96. DOI : 10.1109/TVCG.2008.137.

[HZ22] Md. Yousuf Hossain et Loutfouz Zaman. « NCAlt : Alternatives and Difference Visualizations for Behavior Trees in Game Development Learning ». In : *Proceedings of the ACM on Human-Computer Interaction* 6 (CHI PLAY 31 oct. 2022), 245 :1-245 :31. DOI : 10.1145/3549508. URL : https://doi.org/10.1145/3549508.

[JIJ12] Autor Karel Jakubec, Katedra Katedra Softwarového Inženýrství et Karel Jakubec. « Title : Management of Undo/Redo Operations in Complex Environments ». 2012.

[KF00] David Kurlander et Steven Feiner. « A Visual Language for Browsing, Undoing, and Redoing Graphical Interface Commands ». In : (9 mars 2000).

[KF92] David Kurlander et Steven Feiner. « A History-Based Macro by Example System ». In : *Proceedings of the 5th Annual ACM Symposium on User Interface Software and Technology*. UIST '92. New York, NY, USA : Association for Computing Machinery, déc. 1992, p. 99-106. isbn : 978-0-89791-549-6. doi : 10.1145/142621.142633. (Visité le 05/09/2023).

[KHM17] Mary Beth Kery, Amber Horvath et Brad Myers. « Variolite : Supporting Exploratory Programming by Data Scientists ». In : *Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems* (Denver, Colorado, USA). CHI '17. New York, NY, USA : ACM, 2017, p. 1265-1276. isbn : 978-1-4503-4655-9. doi : 10.1145/3025453.3025626. url : http://doi.acm.org/10.1145/3025453.3025626.

[Kle+02] Scott Klemmer et al. « Where Do Web Sites Come from ? Capturing and Interacting with Design History ». In : 1er jan. 2002, p. 1-8. doi : 10.1145/503376.503378.

[Kria] Krita. *Undo History — Krita Manual 5.2.0 Documentation*. url : https://docs.krita.org/en/reference%5C_manual/dockers/undo%5C_history.html (visité le 05/09/2023).

[Krib] Krita. *Vector Graphics — Krita Manual 5.2.0 Documentation*. url : https://docs.krita.org/en/user%5C_manual/vector%5C_graphics.html (visité le 07/09/2023).

[Kri22] Krita. *Libs/Command · Master · Graphics / Krita · GitLab*. Nov. 2022. url : https://invent.kde.org/graphics/krita/-/tree/master/libs/command (visité le 05/09/2023).

[KS12] DoHyoung Kim et Frank M. Shipman. « Visualizing History to Improve Users' Location and Comprehension of Collaborative Work ». In : *Proceedings of the 2012 ACM International Conference on Supporting Group Work*. GROUP '12. New York, NY, USA : Association for Computing Machinery, oct. 2012, p. 11-20. isbn : 978-1-4503-1486-2. doi : 10.1145/2389176.2389179. (Visité le 19/07/2023).

[L B19] Matthew L. Bolton. *A Task-Based Taxonomy of Erroneous Human Behavior | Elsevier Enhanced Reader*. 15 oct. 2019. doi : 10.1016/j.ijhcs.2017.06.006. url : https://reader.elsevier.com/reader/sd/pii/S1071581917301003?token=9C6A01B725274D1FBBDCCC3C045D347C41F29788E17C0DC13F0E190E9D78606997F346FFA6A0A67299ABAEFEDE0479D3.

[Lee86] George B. Leeman. « A Formal Approach to Undo Operations in Programming Languages ». In : *ACM Transactions on Programming Languages and Systems* 8.1 (2 jan. 1986), p. 50-87. issn : 0164-0925. doi : 10.1145/5001.5005. url : https://doi.org/10.1145/5001.5005.

[Lit+22] Geoffrey Litt et al. « Peritext : A CRDT for Collaborative Rich Text Editing ». In : *Proceedings of the ACM on Human-Computer Interaction* 6.CSCW2 (nov. 2022), 531 :1-531 :36. DOI : 10.1145/3555644. (Visité le 29/05/2023).

[LLM20] Zipeng Liu, Zhicheng Liu et Tamara Munzner. « Data-Driven Multi-level Segmentation of Image Editing Logs ». In : *Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems*. New York, NY, USA : Association for Computing Machinery, 21 avr. 2020, p. 1-12. ISBN : 978-1-4503-6708-0. URL : https://doi.org/10.1145/3313831.3376152.

[Lun+13] Aran Lunzer et al. *Experiments with Worlds*. 2013.

[Lv+19] Xiao Lv et al. « An Optimized RGA Supporting Selective Undo for Collaborative Text Editing Systems ». In : *Journal of Parallel and Distributed Computing* 132.C (oct. 2019), p. 310-330. ISSN : 0743-7315. DOI : 10.1016/j.jpdc.2019.05.005. (Visité le 19/07/2023).

[Man96] Roberta Mancini. « Modelling Interactive Computing Exploiting the Undo ». 1996. URL : https://www.hcibook.net/docs/Roberta-Mancini-PhD-thesis-1996-ocr.pdf.

[McC56] John McCarthy. « The Inversion of Functions Defined by Turing Machines ». In : *Automata Studies. (AM-34)*. Sous la dir. de C. E. Shannon et J. McCarthy. Princeton University Press, 31 déc. 1956, p. 177-182. ISBN : 978-1-4008-8261-8. DOI : 10.1515/9781400882618-009. URL : https://www.degruyter.com/document/doi/10.1515/9781400882618-009/html.

[MDL01] Roberta Mancini, Alan Dix et Stefano Levialdi. « Reflections on Undo ». In : (17 mai 2001).

[Mica] Microsoft. *Microsoft Word : Undo Redo Repeat*. URL : https://support.microsoft.com/en-us/office/undo-redo-or-repeat-an-action-84bdb9bc-4e23-4f06-ba78-f7b893eb2d28.

[Micb] Microsoft. *VS Code Persistent History March 2020*. URL : https://code.visualstudio.com/updates/v1_44#_keep-undo-stack-when-reopening-files (visité le 21/03/2023).

[MJM12] Emerson Murphy-Hill, Rahul Jiresal et Gail C. Murphy. « Improving Software Developers' Fluency by Recommending Development Environment Commands ». In : *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering* (Cary, North Carolina). FSE '12. New York, NY, USA : ACM, 2012, 42 :1-42 :11. ISBN : 978-1-4503-1614-9. DOI : 10.1145/2393596.2393645. URL : http://doi.acm.org/10.1145/2393596.2393645.

[MK96] Brad A. Myers et David S. Kosbie. « Reusable Hierarchical Command Objects ». In : *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (Vancouver, British Columbia, Canada). CHI '96. New York, NY, USA : ACM, 1996, p. 260-267. ISBN : 978-0-89791-777-3. DOI : 10.1145/238386.238526. URL : http://doi.acm.org/10.1145/238386.238526.

[MP19] Mary Lou Maher et Josiah Poon. « Co-Evolution and Emergence in Design ». In : *Design Studies* 65 (1er nov. 2019). DOI : 10.1016/j.destud.2019.10.005.

[MP96] Mary Lou Maher et Josiah Poon. « Modeling Design Exploration as Co-Evolution ». In : *Computer-Aided Civil and Infrastructure Engineering* 11.3 (1er mai 1996), p. 195-209. ISSN : 1467-8667. DOI : 10.1111/j.1467-8667.1996.tb00323.x. URL : https://onlinelibrary.wiley.com/doi/10.1111/j.1467-8667.1996.tb00323.x.

[MSI17] Hiroaki Mikami, Daisuke Sakamoto et Takeo Igarashi. « Micro-Versioning Tool to Support Experimentation in Exploratory Programming ». In : *Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems*. CHI '17. New York, NY, USA : Association for Computing Machinery, 2 mai 2017, p. 6208-6219. ISBN : 978-1-4503-4655-9. DOI : 10.1145/3025453.3025597. URL : https://doi.org/10.1145/3025453.3025597.

[Mye+15] Brad A. Myers et al. « Selective Undo Support for Painting Applications ». In : *Proceedings of the 33rd Annual ACM Conference on Human Factors in Computing Systems - CHI '15*. The 33rd Annual ACM Conference. Seoul, Republic of Korea : ACM Press, 2015, p. 4227-4236. ISBN : 978-1-4503-3145-6. DOI : 10.1145/2702123.2702543. URL : http://dl.acm.org/citation.cfm?doid=2702123.2702543.

[Mye98] Brad A. Myers. « Scripting Graphical Applications by Demonstration ». In : *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems - CHI '98*. The SIGCHI Conference. Los Angeles, California, United States : ACM Press, 1998, p. 534-541. ISBN : 978-0-201-30987-4. DOI : 10.1145/274644.274716. URL : http://portal.acm.org/citation.cfm?doid=274644.274716.

[NC14] Mathieu Nancel et Andy Cockburn. « Causality | Proceedings of the SIGCHI Conference on Human Factors in Computing Systems ». In : 26 avr. 2014. URL : https://dl.acm.org/doi/10.1145/2556288.2556990.

[Nor86] D Norman. *Cognitive Engineering*. 1er jan. 1986, p. 62. 31 p.

[OST13] Tatsuhito Oe, Buntarou Shizuki et Jiro Tanaka. « Undo/Redo by Trajectory ». In : *Human-Computer Interaction. Interaction Modalities and Techniques*. Sous la dir. de Masaaki Kurosu. Lecture Notes in Computer Science. Berlin, Heidelberg : Springer, 2013, p. 712-721. ISBN : 978-3-642-39330-3. DOI : 10.1007/978-3-642-39330-3_77.

[Pal97] Leysia Ann Palen. « Groupware Adoption & Adaptation ». In : *CHI '97 Extended Abstracts on Human Factors in Computing Systems*. CHI EA '97. New York, NY, USA : Association for Computing Machinery, mars 1997, p. 67-68. isbn : 978-0-89791-926-5. doi : 10.1145/1120212.1120257. (Visité le 19/07/2023).

[Par22] Pamela Park. *This Is Why You Don't Overuse the Undo Button*. Wacom Americas' Blog. 19 mai 2022. url : https://community.wacom.com/us/this-is-why-you-dont-overuse-the-undo-button/.

[Phoa] Photoshop. *Add Vector Graphics to Your Designs*. url : https://helpx.adobe.com/content/help/en/photoshop/how-to/vector-objects.html (visité le 07/09/2023).

[Phob] Photoshop. *Undo and History*. url : https://helpx.adobe.com/content/help/en/photoshop/using/undo-history.html.

[Pija] Pijul. *Theory - The Pijul Manual*. url : https://pijul.org/manual/theory.html?highlight=patches%5C#theory (visité le 05/09/2023).

[Pijb] Pijul. *Why Pijul - The Pijul Manual*. url : https://pijul.org/manual/why%5C_pijul.html%5C#pijul-for-gitmercurialsvn-users (visité le 05/09/2023).

[PK92] Atul Prakash et Michael J. Knister. « Undoing Actions in Collaborative Work ». In : *Proceedings of the 1992 ACM Conference on Computer-supported Cooperative Work*. CSCW '92. New York, NY, USA : Association for Computing Machinery, 1er déc. 1992, p. 273-280. isbn : 978-0-89791-542-7. doi : 10.1145/143457.143527. url : https://doi.org/10.1145/143457.143527.

[PK94] Atul Prakash et Michael J. Knister. « A Framework for Undoing Actions in Collaborative Systems ». In : *ACM Trans. Comput.-Hum. Interact.* 1.4 (4 déc. 1994), p. 295-330. issn : 1073-0516. doi : 10.1145/198425.198427. url : http://doi.acm.org/10.1145/198425.198427.

[Pro] Procreate. *QuickShape - Procreate® Handbook*. url : https://procreate.com/handbook/procreate/guides/quickshape/ (visité le 05/09/2023).

[Qta] Qt. *Overview of Qt's Undo Framework | Qt 5.15*. url : https://doc.qt.io/qt-5/qundo.html.

[Qtb] Qt. *QUndoCommand.mergeWith*. url : https://doc.qt.io/qt-6/qundocommand.html%5C#mergeWith (visité le 05/09/2023).

[Qtc] Qt. *QUndoStack Class | Qt Widgets 5.15.8*. url : https://doc.qt.io/qt-5/qundostack.html.

[Reaa] React. *Context – React*. url : https://legacy.reactjs.org/docs/context.html (visité le 05/09/2023).

[Reab] React. *State : A Component's Memory – React*. url : https://react.dev/learn/state-a-components-memory (visité le 07/09/2023).

[Red21] Redux. *Redux : Store, getState*. Oct. 2021. URL : https://redux.js.org/api/store (visité le 07/09/2023).

[Red23a] Redux. *Redux Fundamentals, Part 1 : Redux Overview | Redux*. Avr. 2023. URL : https://redux.js.org/tutorials/fundamentals/part-1-overview (visité le 05/09/2023).

[Red23b] Redux. *Redux : Store*. Juin 2023. URL : https://redux.js.org/understanding/thinking-in-redux/glossary (visité le 07/09/2023).

[Rek99] Jun Rekimoto. « Time-Machine Computing : A Time-centric Approach for the Information Environment ». In : *Proceedings of the 12th Annual ACM Symposium on User Interface Software and Technology* (Asheville, North Carolina, USA). UIST '99. New York, NY, USA : ACM, 1999, p. 45-54. ISBN : 978-1-58113-075-1. DOI : 10.1145/320719.322582. URL : http://doi.acm.org/10.1145/320719.322582.

[RG99] Matthias Ressel et Rul Gunzenhäuser. « Reducing the Problems of Group Undo ». In : *Proceedings of the International ACM SIGGROUP Conference on Supporting Group Work - GROUP '99*. The International ACM SIGGROUP Conference. Phoenix, Arizona, United States : ACM Press, 1999, p. 131-139. ISBN : 978-1-58113-065-2. DOI : 10.1145/320297.320312. URL : http://portal.acm.org/citation.cfm?doid=320297.320312.

[RS96] Charles Rich et Candace L. Sidner. « Adding a Collaborative Agent to Graphical User Interfaces ». In : *Proceedings of the 9th Annual ACM Symposium on User Interface Software and Technology - UIST '96*. Seattle, Washington, United States : ACM Press, 1996, p. 21-30. ISBN : 978-0-89791-798-8. DOI : 10.1145/237091.237094. (Visité le 20/07/2023).

[RS97] Charles Rich et Candace L. Sidner. « Segmented Interaction History in a Collaborative Interface Agent ». In : *Proceedings of the 2nd International Conference on Intelligent User Interfaces*. IUI '97. New York, NY, USA : Association for Computing Machinery, jan. 1997, p. 23-30. ISBN : 978-0-89791-839-8. DOI : 10.1145/238218.238222. (Visité le 19/07/2023).

[Sch+20] Philippe Schmid et al. « Interaction Interferences : Implications of Last-Instant System State Changes ». In : *Proceedings of the 33rd Annual ACM Symposium on User Interface Software and Technology*. UIST '20 : The 33rd Annual ACM Symposium on User Interface Software and Technology. Virtual Event USA : ACM, 20 oct. 2020, p. 516-528. ISBN : 978-1-4503-7514-6. DOI : 10.1145/3379337.3415883. URL : https://dl.acm.org/doi/10.1145/3379337.3415883.

[Sei+12] Thomas Seifried et al. « Regional Undo/Redo Techniques for Large Interactive Surfaces ». In : *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (Austin, Texas, USA). CHI '12. New York, NY, USA : ACM, 2012, p. 2855-2864. ISBN : 978-1-4503-1015-4. DOI : 10.1145/2207676.2208690. URL : http://doi.acm.org/10.1145/2207676.2208690.

[Shn83] Ben Shneiderman. « Direct Manipulation : A Step Beyond Programming Languages ». In : (1983), p. 13.

[Ste+87] Mark Stefik et al. « Beyond the Chalkboard : Computer Support for Collaboration and Problem Solving in Meetings ». In : *Communications of the ACM* 30.1 (jan. 1987), p. 32-47. issn : 0001-0782. doi : 10.1145/7885.7887. (Visité le 27/07/2023).

[Sun] Chengzheng Sun. « Real Differences between OT and CRDT for Co-Editors ». In : (), p. 33.

[Sun00] Chengzheng Sun. « Undo Any Operation at Any Time in Group Editors ». In : *Proceedings of the 2000 ACM Conference on Computer Supported Cooperative Work*. CSCW '00. New York, NY, USA : Association for Computing Machinery, 1er déc. 2000, p. 191-200. isbn : 978-1-58113-222-9. doi : 10.1145/358916.358990. url : https://doi.org/10.1145/358916.358990.

[Sys] Dassault Systems. *SolidWorks : Viewing Feature Relationships - 2023 - SOLIDWORKS Help*. url : https://help.solidworks.com/2023/English/SolidWorks/sldworks/c_viewing_feature_relationships.htm.

[Ter+04] Michael Terry et al. « Variation in Element and Action : Supporting Simultaneous Development of Alternative Solutions ». In : *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (Vienna, Austria). CHI '04. New York, NY, USA : ACM, 2004, p. 711-718. isbn : 978-1-58113-702-6. doi : 10.1145/985692.985782. url : http://doi.acm.org/10.1145/985692.985782.

[The20] TheBeginnerDrawingCourse. *12 Drawing Tips for Beginners - Digital Art*. THE BEGINNER DRAWING COURSE. 19 juill. 2020. url : https://www.thebeginnerdrawingcourse.com/blogforartists/2020/7/17/12-drawing-tips-for-beginners-digital-art.

[TM02] Michael Terry et Elizabeth D. Mynatt. « Recognizing Creative Needs in User Interface Design ». In : *Proceedings of the Fourth Conference on Creativity & Cognition - C&C '02*. The Fourth Conference. Loughborough, UK : ACM Press, 2002, p. 38-44. isbn : 978-1-58113-465-0. doi : 10.1145/581710.581718. url : http://portal.acm.org/citation.cfm?doid=581710.581718.

[Vima] Vim. *Vim Documentation : Repeat*. url : https://vimhelp.org/repeat.txt.html#repeating.

[Vimb] Vim. *Vim Documentation : Undo*. url : https://vimhelp.org/undo.txt.html#undo-tree.

[Vimc] Vim. *Vim Persistent Undo ; Documentation*. url : https://vimhelp.org/undo.txt.html#undo-persistence.

[Vimd] Vim. *Vim : Motion.Txt*. url : https://vimhelp.org/motion.txt.html#jump-motions.

[Vit84] Jeffrey Scott Vitter. « US&R : A New Framework for Redoing (Extended Abstract) ». In : *ACM SIGPLAN Notices* 19.5 (25 avr. 1984), p. 168-176. ISSN : 0362-1340. DOI : 10.1145/390011.808262. URL : https://doi.org/10.1145/390011.808262.

[WDP99] Bradley C. Wheeler, Alan R. Dennis et Laurence I. Press. « Groupware Comes to the Internet : Charting a New World ». In : *ACM SIGMIS Database : the DATABASE for Advances in Information Systems* 30.3-4 (sept. 1999), p. 8-21. ISSN : 0095-0033. DOI : 10.1145/344241.344242. (Visité le 19/07/2023).

[WG91] Haiying Wang et Mark Green. « An Event-Object Recovery Model for Object-Oriented User Interfaces ». In : *Proceedings of the 4th Annual ACM Symposium on User Interface Software and Technology*. UIST '91. New York, NY, USA : Association for Computing Machinery, 11 nov. 1991, p. 107-115. ISBN : 978-0-89791-451-2. DOI : 10.1145/120782.120794. URL : https://doi.org/10.1145/120782.120794.

[YAN19] YIYA YANG. *Undo Support Models*. 15 oct. 2019. DOI : 10.1016/S0020-7373(88)80056-7. URL : https://reader.elsevier.com/reader/sd/pii/S0020737388800567?token=CBD2B2DC25EAC27D2C1C944421ABBB9D223777E64834ADE1A469E8D694C09C2116A6FFB5E45F2638CE2905CF694FBE0E.

[Yan92] Yiya Yang. « Motivation, Practice and Guidelines for "Undoing." » In : *Interacting with Computers* (1992). DOI : 10.1016/0953-5438(92)90011-4.

[YB15] Young Seok Yoon et Brad A. Myers. *Supporting Selective Undo in a Code Editor | Proceedings of the 37th International Conference on Software Engineering - Volume 1*. Mai 2015. URL : https://dlnext.acm.org/doi/abs/10.5555/2818754.2818784 (visité le 29/11/2019).

[YM12] YoungSeok Yoon et Brad Myers. « An Exploratory Study of Backtracking Strategies Used by Developers ». In : 2012 5th International Workshop on Co-operative and Human Aspects of Software Engineering, CHASE 2012 - Proceedings. 2 juin 2012. DOI : 10.1109/CHASE.2012.6223012.

[YM14] Young Seok Yoon et Brad A. Myers. « A Longitudinal Study of Programmers' Backtracking ». In : *2014 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. 2014 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC). Juill. 2014, p. 101-108. DOI : 10.1109/VLHCC.2014.6883030.

[YM19] Young Seok Yoon et Brad A. Myers. *Supporting Selective Undo in a Code Editor | Proceedings of the 37th International Conference on Software Engineering - Volume 1*. 29 nov. 2019. URL : https://dlnext.acm.org/doi/abs/10.5555/2818754.2818784.

[Zha+15] Zhenpeng Zhao et al. « Sketcholution : Interaction Histories for Sketching ». In : *International Journal of Human-Computer Studies* 82 (1er oct. 2015), p. 11-20. ISSN : 1071-5819. DOI : 10.1016/j.ijhcs.2015.04.003. URL : https://www.sciencedirect.com/science/article/pii/S1071581915000725.

Table des matières

Développement d'historiques de commandes avancés pour améliorer le processus d'édition numérique

Développement d'un modèle et d'une architecture d'historique de commandes multi-fonction

Résumé

Les historiques de commandes sont omniprésents dans le monde de l'édition. Au cours du temps, plusieurs modèles d'historiques ont émergé, abordant différentes fonctionnalités comme des types d'annulations et de ré-exécution, ainsi que la gestion des conséquences de leur utilisation. Ces modèles, bien que souvent adaptés à un cas d'usage spécifique, partagent des caractéristiques structurelles et fonctionnelles communes. Ces caractéristiques peuvent être catégorisées pour étudier les différences entre modèles. Certains de ces choix de caractéristiques vont avoir des conséquences sur le comportement des fonctionnalités proposées, c'est à dire qu'une fonctionnalité aura des effets différents selon certaines caractéristiques du modèle la proposant.

Cette thèse porte sur le lien entre les fonctionnalités proposées par les historiques de commande et la structure de ces historiques. L'objectif de cette thèse est de présenter une architecture logicielle d'historique de commandes laissant au logiciel le choix du comportement des fonctionnalités parmi au moins ceux étudiés dans la littérature. Cela demande de pouvoir les implémenter de manière non exclusive et combinable avec une même structure d'historique.

En effet, les différents comportements et fonctionnalités proposés par les différents modèles s'avèrent être utiles pour l'utilisateur lors d'une tâche d'édition. Ces modèles ne sont cependant pas combinables, forçant un choix en amont par le développeur du logiciel d'édition, et limitant ainsi les capacités de l'historique, qui ne pourra alors pas répondre à certains besoins de l'utilisateur lors de son travail d'édition. Ce constat est la motivation principale de cette thèse, et a mené à *trois contributions* :

- **La catégorisation des caractéristiques des historiques de commandes** permet de mettre en relation et d'organiser les différents modèles et discussions de l'état de l'art dans le domaine des historiques de commandes comprenant de nombreuses sous parties. Elle établit la liste des choix à faire lors de l'implémentation d'une fonctionnalité et les options existantes qui mènent vers des comportements différents. Elle propose une manière d'approcher le sujet dans sa globalité, et établit ainsi une fondation pour un modèle unifié.
- **Le modèle abstrait ESCI** unifie le fonctionnement des historiques en présentant une structure d'historique qui permet d'implémenter les différents comportements des fonctionnalités de la littérature. Il reprend les concepts fondamentaux de Causality et les étend. La différence principale entre les deux modèles réside dans leur gestion des états, ESCI ajoutant de la flexibilité quant au suivi de l'évolution de données éditées.
- **L'architecture logicielle ReTracer** propose une manière d'implémenter ESCI. Elle permet d'apporter quelques précisions et aborde certains détails d'implémentation du modèle. L'implémentation du modèle permet de mener des expériences pour étudier les apports d'un tel historique au travail d'édition de l'utilisateur ainsi que les interfaces d'interaction adaptées, ce qui fait l'objet de travaux futurs.

Mots clés : ihm, historiques de commandes, structures de données, édition, exploration, collaboration

---

Centre Inria de l'Université de Lille

Parc scientifique de la Haute-Borne – 40, avenue Halley - Bât. A - Park Plaza – 59650 Villeneuve d'Ascq – France

Developing advanced command histories to improve digital editing processes

A Swiss army knife of command histories model and architecture

Abstract

Command histories are ubiquitous in the field of editing. Several history models have emerged with time, exploring various functionalities like different types of undoing and re-execution as well as managing the consequences of their usage. These models share structural and functional characteristics, albeit often adapted to a specific use case. These characteristics can be categorised to study the differences between models. Choosing between the possible characteristics will have consequences on the behaviour of the proposed functionalities, that is to say a functionality will have different effects depending on the model's characteristics.

In this thesis, we study the link between the command histories' functionalities and the structure of these histories. The objective is to present a command history software architecture giving the system the choice over the functionalities' behaviour. This requires implementing them in a non exclusive and combinable way through one unique history structure.

Being able to choose between the various behaviours can become useful for the user during an editing task. However, the models in the literature aren't combinable, forcing the software developer to make a behaviour choice ahead of time. This limits the history's capabilities, preventing it from answering some user needs. *Three contributions* derive from this statement:

- **A categorisation of command history characteristics** links together the various models and discussions about command histories in the literature. A list of choices to be made at the time of implementation of a functionality and the existing options available is established. This sets a foundation for building a unified model.
- **The abstract command history model ESCI** unifies the various models into one structure capable of replicating the various behaviours presented in the literature. This model is based on Causality and extends its concepts. The main difference between these two models is their state management. ESCI also adds some flexibility in the tracing of data evolution.
- **The ReTracer software architecture** is a way to implement ESCI. It goes into technical details. Implementing this model allows for studies about the benefits of such a command history as well as adequate interaction interfaces, which is a future project.

Keywords: hci, command histories, data structures, editing, exploration, collaboration