

# **Synthesis of User Interfaces with Categorical Methods**

by

**Marta Zheplinska**

A thesis  
presented to the University of Waterloo  
in fulfillment of the  
thesis requirement for the degree of  
Master of Applied Science  
in  
Systems Design Engineering

Waterloo, Ontario, Canada, 2024

© Marta Zheplinska 2024

### **Author's Declaration**

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

## **Abstract**

We apply category theory to modeling user interfaces, focusing on the interaction between functional configuration and user perception. By representing user interfaces as directed labeled multigraphs and applying pullback constructions to the category of directed labeled multigraphs, the study formalizes interface structures in a way that encompasses both technical operations and users' perceptual capabilities. Users interact with interfaces by identifying affordances that hint at possible actions. We propose a perception-driven interpretation of the interface as a set of affordances available to the user. Conversely, the structure of the interface is formed by its functional components and presented in different states. This double strategy aims to provide a tool for studying the usability and unambiguity of interactive systems and analyzing how interfaces communicate functionality to users through structural design.

A major component of this research is the study of user profiles, which define the relationship between the interface and the cognitive and physical characteristics of users. Profiles are encoded in the interface representation by graphs as certain filters of the pullback graph to match the human perception. This approach provides a basis for assessing usability for people with different cognitive and physical abilities.

Ambiguity is the presence of multiple possible meanings, interpretations, or outcomes that can cause uncertainty. It affects the user's sense of control over the interface. The thesis examines ambiguity in user interfaces within a categorical formal representation. By applying the conditions to the pullback, the study proposes a formal method to detect the ambiguity of the interface from the perspective of a particular user profile, supported by provided case studies.

## **Acknowledgements**

The support of the Natural Sciences and Engineering Research Council of Canada (NSERC), funding reference number RGPIN-2019-04669, and NSERC Special Fund for Trainees (Ukraine) is gratefully acknowledged.

Special thanks to Dr. Amena Assem and my supervisor, Professor Chrystopher L. Nehaniv, and the thesis readers, Professor Britt Anderson and Professor Catherine Burns, for their invaluable feedback and guidance on this thesis.

## **Dedication**

This research is dedicated to the people of Ukraine in response to the Russian invasion.

# Table of Contents

|                                                                 |            |
|-----------------------------------------------------------------|------------|
| <b>Author's Declaration</b>                                     | <b>ii</b>  |
| <b>Abstract</b>                                                 | <b>iii</b> |
| <b>Acknowledgements</b>                                         | <b>iv</b>  |
| <b>Dedication</b>                                               | <b>v</b>   |
| <b>List of Figures</b>                                          | <b>ix</b>  |
| <b>List of Tables</b>                                           | <b>xii</b> |
| <b>1 Introduction</b>                                           | <b>1</b>   |
| <b>2 Category Theory Framework</b>                              | <b>7</b>   |
| 2.1 Terminology . . . . .                                       | 7          |
| 2.2 Category of Directed labeled Multigraphs . . . . .          | 10         |
| 2.2.1 Universal Constructions in <i>DLMG</i> Category . . . . . | 11         |
| 2.2.2 Product . . . . .                                         | 11         |
| 2.2.3 Coproduct . . . . .                                       | 15         |
| 2.2.4 Pullback . . . . .                                        | 20         |
| 2.2.5 Pushout . . . . .                                         | 25         |

|          |                                                                |           |
|----------|----------------------------------------------------------------|-----------|
| <b>3</b> | <b>Representation of User Interface as a Pullback</b>          | <b>32</b> |
| 3.1      | User Interface as Universal Construction . . . . .             | 32        |
| 3.1.1    | Action Graph $A$ . . . . .                                     | 34        |
| 3.1.2    | Affordance Graph $D$ . . . . .                                 | 38        |
| 3.1.3    | Modes . . . . .                                                | 42        |
| 3.1.4    | Abstract Graph $M$ . . . . .                                   | 44        |
| 3.1.5    | Pullback Graph $P$ . . . . .                                   | 48        |
| 3.1.6    | Puzzles as Interacting Systems . . . . .                       | 54        |
| 3.1.7    | Example - Generic Form . . . . .                               | 56        |
| 3.2      | Physical vs. Digital Interfaces . . . . .                      | 59        |
| <b>4</b> | <b>Profiles</b>                                                | <b>63</b> |
| 4.1      | Pullback Relation to User Perception . . . . .                 | 63        |
| <b>5</b> | <b>Unambiguity of User Interfaces</b>                          | <b>66</b> |
| 5.1      | Unambiguity . . . . .                                          | 66        |
| 5.1.1    | Strict Unambiguity . . . . .                                   | 71        |
| 5.1.2    | Profiles and Unambiguity . . . . .                             | 75        |
| 5.1.3    | When Ambiguity is Detected . . . . .                           | 76        |
| 5.2      | Case Study of a Generic Form . . . . .                         | 78        |
| 5.3      | Case Study of a Generic Form - 2 . . . . .                     | 83        |
| 5.4      | Case Study of Light Control System (E5 Seminar Room) . . . . . | 86        |
| 5.4.1    | Strategy “Feedback” . . . . .                                  | 95        |
| 5.5      | Discussion . . . . .                                           | 97        |
| <b>6</b> | <b>Generalizations</b>                                         | <b>98</b> |
| 6.1      | Event-Driven Interfaces . . . . .                              | 98        |
| 6.1.1    | Clock Example . . . . .                                        | 99        |
| 6.1.2    | Event-Driven Interfaces as Pushouts . . . . .                  | 103       |
| 6.2      | Multi-User Interfaces . . . . .                                | 104       |
| 6.3      | Discussion . . . . .                                           | 105       |

|                                                         |            |
|---------------------------------------------------------|------------|
| <b>7 Summary of Contributions and Future Directions</b> | <b>106</b> |
| <b>References</b>                                       | <b>108</b> |
| <b>APPENDICES</b>                                       | <b>111</b> |
| <b>A Pseudocode</b>                                     | <b>112</b> |

# List of Figures

|      |                                                                                                                                                                                                 |    |
|------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|----|
| 3.1  | High level diagram of user interfaces pullback representation with the categorical framework. . . . .                                                                                           | 33 |
| 3.2  | Different states of the text reader. A segment refers to a specific portion of some content displayed on the page. Each column represents some division of the same text into segments. . . . . | 35 |
| 3.3  | The action graph of the text reader interacting system. . . . .                                                                                                                                 | 37 |
| 3.4  | The action graph of the recorder. . . . .                                                                                                                                                       | 38 |
| 3.5  | Perceived action graph of the recorder. . . . .                                                                                                                                                 | 38 |
| 3.6  | Simple context, consisting of four affordances, <i>in</i> , <i>out</i> , <i>r</i> and <i>l</i> . . . . .                                                                                        | 39 |
| 3.7  | Nine unique contexts provided to the user by a book reader interface. . . . .                                                                                                                   | 40 |
| 3.8  | The affordance graph with one context, and four loops. . . . .                                                                                                                                  | 41 |
| 3.9  | The affordance graph with nine contexts, and affordances realizations. . . . .                                                                                                                  | 42 |
| 3.10 | <i>Penultimate</i> iPad application that allows user to write in different modes. . . . .                                                                                                       | 43 |
| 3.11 | The pullback graph of a text reader 1.1. . . . .                                                                                                                                                | 52 |
| 3.12 | The pullback graph of a text reader 1.2. . . . .                                                                                                                                                | 53 |
| 3.13 | Vizualization of a 3-puzzle (top), 7-puzzle (middle), 8-puzzle(bottom) in the center, together with their affordance graphs to the left and the abstract graphs to the right. . . . .           | 54 |
| 3.14 | The action graph vs the pullback graph of a 3-puzzle game. . . . .                                                                                                                              | 55 |
| 3.15 | The action graph of a simple form with states defined by the content in the textbox. . . . .                                                                                                    | 57 |
| 3.16 | The affordance graph of a simple form that provides to the user a fixed set of possible actions (affordances) - <b>a</b> , <b>b</b> , and <b>delete</b> . . . . .                               | 58 |

|      |                                                                                                                                                                                                                                                                                                                                                                                                                                                              |     |
|------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-----|
| 3.17 | The abstract graph of a simple form. . . . .                                                                                                                                                                                                                                                                                                                                                                                                                 | 58  |
| 3.18 | The pullback graph of a simple form. A form is encoded by a state of the textbox and buttons available. Edges describe resulting context-states pairs after actions realizations. . . . .                                                                                                                                                                                                                                                                    | 60  |
| 5.1  | The affordance graph of the generic form, where five possible actions are available.                                                                                                                                                                                                                                                                                                                                                                         | 78  |
| 5.2  | The action graph of the generic form, where a state is defined by the textbox content. . . . .                                                                                                                                                                                                                                                                                                                                                               | 79  |
| 5.3  | The abstract graph of the generic form. . . . .                                                                                                                                                                                                                                                                                                                                                                                                              | 80  |
| 5.4  | The pullback graph of the generic form that enables user interaction with the text.                                                                                                                                                                                                                                                                                                                                                                          | 82  |
| 5.5  | The affordance graph of a generic form, where two modes are present. . . . .                                                                                                                                                                                                                                                                                                                                                                                 | 83  |
| 5.6  | The action graph of a generic form, where a state is defined by a text in the text box. . . . .                                                                                                                                                                                                                                                                                                                                                              | 84  |
| 5.7  | The abstract graph of the failed form 1.2. . . . .                                                                                                                                                                                                                                                                                                                                                                                                           | 86  |
| 5.8  | The pullback graph for a generic form that enables user interaction with the text within two modes. . . . .                                                                                                                                                                                                                                                                                                                                                  | 87  |
| 5.9  | Four light blocks in the seminar room correspond to the buttons and are associated with specific colors (1 - red; 2 - blue; 3 - purple; 4 - orange). . . . .                                                                                                                                                                                                                                                                                                 | 88  |
| 5.10 | Formal representation of light control consisting of 8 buttons. The upper and lower buttons are enumerated from 1 to 4, and suffixed with 't' and 'b', respectively.                                                                                                                                                                                                                                                                                         | 89  |
| 5.11 | The affordance graph of the light contexts in the seminar room. . . . .                                                                                                                                                                                                                                                                                                                                                                                      | 90  |
| 5.12 | The action graph of the light states in the seminar room. . . . .                                                                                                                                                                                                                                                                                                                                                                                            | 91  |
| 5.13 | The abstract graph of the light control in the seminar room. . . . .                                                                                                                                                                                                                                                                                                                                                                                         | 91  |
| 5.14 | The pullback graph of the light control interface. . . . .                                                                                                                                                                                                                                                                                                                                                                                                   | 94  |
| 5.15 | Modified pullback graph of the light control interface with a focus on feedback. .                                                                                                                                                                                                                                                                                                                                                                           | 96  |
| 6.1  | The affordance graph demonstrates the transitions between a quiet context and a ringing context. The quiet context allows adding an alarm, which does not change the state immediately. Transition to the ringing context occurs when the alarm starts, either automatically or manually. The ringing context can revert to quiet through manual stopping or automatic cessation. Time progression within each context is represented by self-loops. . . . . | 100 |

|     |                                                                                                                                                                                                                                                                                                                                                                                            |     |
|-----|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-----|
| 6.2 | The action graph representing the system states and transitions. The graph highlights three main states: no alarm, set alarm, and activated alarm, with transitions based on actions like adding, activating, stopping, or postponing the alarm. Time changes are represented with self-loops, reflecting the evolving state of the clock logic. . . . .                                   | 100 |
| 6.3 | The abstract graph shows two main states: quiet and ringing. Transitions in the quiet state are triggered by time and alarms, while the ringing state handles time progression, stopping, and holding the alarm. . . . .                                                                                                                                                                   | 101 |
| 6.4 | The pullback graph illustrates the relationship between three states: a quiet context without any active alarm, a quiet context with an active alarm, and a ringing context with a stop alarm button. Transitions between these states include adding an alarm, activating it, stopping it, and postponing it. The graph also shows self-loops for time progression in each state. . . . . | 102 |
| 6.5 | The user graph represents interaction between two users when they take turns. . .                                                                                                                                                                                                                                                                                                          | 105 |
| A.1 | UML diagram of the DLMG project. . . . .                                                                                                                                                                                                                                                                                                                                                   | 112 |

# List of Tables

|     |                                                                                                     |     |
|-----|-----------------------------------------------------------------------------------------------------|-----|
| 3.1 | The homomorphism from the action graph to the abstract graph of a reader. . . .                     | 46  |
| 3.2 | The homomorphism from the affordance graph to the abstract graph of a reader. .                     | 46  |
| 3.3 | The homomorphism from the affordance graph to the abstract graph of reader 1.2.                     | 47  |
| 3.4 | The homomorphism from the action graph to the abstract graph, reader example<br>1.2. . . . .        | 48  |
| 3.5 | The homomorphism from the action graph to the abstract graph of the generic<br>form. . . . .        | 58  |
| 3.6 | The homomorphism from the affordance graph to the abstract graph of the form.                       | 59  |
| 5.1 | The homomorphism from the affordance graph to abstract graph, failed form<br>example. . . . .       | 81  |
| 5.2 | The homomorphism from the action graph to abstract graph, failed form example.                      | 81  |
| 5.3 | The homomorphism from the action graph to abstract graph of failed form 1.2. .                      | 85  |
| 5.4 | The homomorphism from the affordance graph to abstract graph of failed form 1.2.                    | 85  |
| 5.5 | The homomorphism from the affordance graph to the abstract graph, light control<br>example. . . . . | 92  |
| 5.6 | The homomorphism from the action graph to abstract graph, light control example.                    | 93  |
| 6.1 | The homomorphism from the action graph to the abstract graph of the clock. . . .                    | 101 |
| 6.2 | The homomorphism from the affordance graph to the abstract graph of the clock.                      | 102 |

# Chapter 1

## Introduction

This thesis explores the application of category theory, specifically within the category of directed labeled multigraphs, to model and analyze interactive systems. The primary focus is on representing user interfaces as pullback universal constructions. The proposed formal representation does not dictate the order or quantity of controls to include or specify a design process. However, the work has potential in evaluating user interfaces with a focus on human perception of interactive context. Additionally, we illustrate how hypothetical scenarios can demonstrate ambiguities in interface usage that may negatively impact user experience.

We aim to provide a structured formalization for studying ambiguous interaction elements in user interfaces and their strong connection to various user profiles. We conceptualize the interactive system as a universal construction that combines two components: functional and interactive. Both components are directly linked to the user's profile, and their representation depends on the aspects of human perception, such as the ability to recognize controls, interpret visual cues, and understand feedback. The technical component represents the states of the interface relevant to the content it provides, while the interactive component consists of users affordances.

This thesis formalizes the interactions between users and systems and presents a method for analyzing their unambiguity. It is based on the assumption that a person can recognize the states and interactive components of the interface. In other words, we consider a fixed user profile. By imposing straightforward conditions on pullback graphs, we can identify ambiguous parts of interfaces, providing an automated tool for analyzing interactive systems. We investigate the appropriateness of a category of directed labeled multigraphs as a tool for representing such interactive systems. While previous research has often focused on state transition diagrams, we expand a presentation by taking into account user affordances.

The research of this thesis is a direct continuation of the work of Loeliger and Nehaniv presented in the paper [19], where the authors propose representing interfaces through directed multigraphs, particularly the affordance graph. This graph encodes an interface by highlighting affordances—hints that guide users in interacting with the system (a more detailed discussion on affordances will follow). In our thesis, we adopt the terminology introduced by the authors while providing complete definitions and roles of the graphs, as this aspect was not fully addressed in their work. The authors demonstrate the application of their formal presentation of interfaces using web pages, constructing both an action graph and an affordance graph (of [19]). The action graph is associated with state transitions combining two types of perceived affordances relevant to web pages, event listeners and hyperlinks. On the other hand, affordance graph construction relates to the physical device the application is running on. The paper marks a significant first step in encoding interfaces using universal constructions within the category of graphs. However, it is essential to note that the structure of web pages has evolved significantly over the past 13 years, requiring the development of new parsing software, yet the foundational ideas proposed by the authors remain relevant and applicable. Additionally, the authors introduce the concept of a user profile, which reflects the cognitive and physical properties of individual users. This approach emphasizes the importance of analyzing interfaces directly associated with specific user profiles rather than attempting to generalize across the users' diversity.

It has been asserted, that cognitive technology as the instrument of our minds is affected by the physical limitations of our bodies and environments [2], and is a tool for transferring information from our heads to our physical environment. As cognitive technology develops, issues such as visibility and interaction become increasingly complex. For example, digital screens can display large amounts of information, but users often have difficulty scrolling and zooming, limited by manual dexterity and reaction time. The authors highlight that many engineers may find it difficult to anticipate user needs due to a focus on technical aspects. Regardless, some studies have considered the importance of considering the limitations certain groups of users face [8], and provided an overview of adaptation techniques [33].

Our research also focuses on user profiles but proposes alternative methods for modifying the pullback graph representation of the user interfaces to better align it with the user's cognitive characteristics. This allows for a targeted accessibility analysis for specific groups based on their abilities. While we acknowledge the relevance of such an analytical approach, our work splits by concentrating on analyzing ambiguity rather than accessibility.

## **Affordances**

The term *affordance* was first introduced by James J. Gibson in his book *The Ecological Approach to Visual Perception* [9]. Gibson defined affordances as the actionable possibilities of-

ferred by the environment, perceived relative to an individual's cognitive and physical capabilities. Affordances, according to Gibson, are not subjective experiences but objective properties of the environment, based on its physical characteristics. They represent the potential for action that exists between the environment and the observer.

Donald Norman [25], in *The Design of Everyday Things*, extended Gibson's concept, especially in the context of Human-Computer Interaction (HCI). Norman's adaptation of the term focuses on perceived affordances, highlighting how users interpret and understand interaction possibilities within designed environments, such as user interfaces. In addition to perceived affordances, Norman also introduces the concept of signifiers, which are cues or markers that indicate how to interact with an object. While affordances refer to the potential actions an object allows, signifiers are the visual or sensory cues that help users recognize these affordances. Unlike Gibson's objective perspective, Norman emphasized the importance of perception—whether users recognize that an action is possible. For instance, a button that looks clickable affords pressing, but this affordance is only meaningful if users perceive it as such. Norman's work underscores that successful design ensures users can easily understand how to interact with interface elements without the need for additional instructions.

In this context, affordances are essential for interface design, as they enable users to intuitively interact with objects. Effective user interfaces rely on clear affordances, reducing cognitive load and providing seamless interaction. This aligns with the idea that interface elements should be easily recognizable and actionable, as emphasized in other design literature [31]. However, Norman's term of perceived affordance specifically stresses how users' recognition of these possibilities plays a critical role in usability and user experience.

## **User Interfaces Representation with Graphs**

Graphs are the most commonly used complex data structure in computer science and engineering [21]. Representation by graphs leads to automation and opens the door to various directions of research. User interfaces have been represented by various variations of graphs, such as UML diagrams [18, 32], state-transition automaton [4], and petri-nets [26]. Each of these possible formal representations aims to bridge the gap between design and engineering, which is the focus of HCI research. The capabilities and limits of each approach are different. However, most representations are often used as initial points for defining system components and outlining their transition scenarios in the context of software design, rather than user experience.

The use of UML diagrams in the design and development process was proposed in [18, 32]. The authors indicated the important components to consider when creating interfaces and suggested using UML diagrams. They also emphasized that a successful and attractive graphic

design does not imply the usability characteristics of the interface. In contrast, the formal presentation methods of interfaces cannot always evaluate usability. The authors discuss the essential components defining a user interface (UI) for applications, emphasizing the need to address information and look-and-feel requirements. While existing UML constructs are insufficient and require extensions to support the complex relationships between tasks, the paper provides possible basic extensions in the context of UI development life cycle [18]. However, it was highlighted that while UML diagrams can effectively visualize relationships and functionalities, they may not fully capture the complexities of user interactions, which could lead to ambiguity in interpreting task flows. It was unclear whether UML diagrams can serve as automated tools for interface design, and to what extent they require manual creation of the diagram or the addition of details related to specific user interface with consideration of user behavior scenarios.

An alternative framework for modeling user interfaces is Petri nets, which excel at representing concurrent actions and states. They offer a clear and effective way to visualize complex interactions, especially when concurrency is present, which is difficult to model using other formal methods. Petri nets are particularly useful in reactive systems, including user interfaces. In the paper [26], the authors highlight how an object-oriented approach can be easily integrated with Petri nets, allowing for the modularization of interface components. The paper also underscores the advantage of Petri nets in identifying potential issues like accessibility limitations and “dead spots” in the interface. However, it is important to note that automata can also be used for this type of analysis [19]. A limitation of Petri nets is their difficulty in analyzing interfaces that must adapt to different user profiles, as they primarily encode application-level information without considering a user-oriented details.

Another framework for formulating user interfaces is the interface automata presented in the paper [4]. Luca de Alfaro and Thomas A. Henzinger introduce a formal framework focused on modeling the interactions between software components through interfaces, focusing on input and output actions rather than internal behaviors [4]. The authors present an interaction-centric approach, which is key for ensuring compatibility between components and preventing errors like deadlocks. Unlike typical finite-state machines, interface automata are designed to capture both input and output actions, making them well-suited for systems that rely on communication between multiple components (for example, user and interface). This approach allows for specifying how components behave when combined, focusing on the software components and their interactions via input and output actions. While users are not explicitly mentioned, their interaction can be interpreted indirectly, as users often drive the inputs that a system’s interface receives, therefore opening doors to user-centered analysis.

Our choice lies in directed labeled multigraphs because they not only facilitate universal constructions in category theory but also align with the goal of formalizing interface design. These graphs inherently support the representation of structures and morphisms, providing a

categorical framework “for free.” More importantly, their flexibility allows for a detailed and adaptable representation of interactive systems, enabling a meaningful connection to the user through the formal presentation of affordances.

## **Category Theory with Applications in Human-Computer Interaction**

Category theory, introduced in 1945 by Eilenberg and MacLane [5], is a remarkably powerful mathematical framework characterized by its simplicity and elegance. It focuses on the relationships between objects through morphisms without delving into details. Although some consider category theory unnecessary as it does not provide us with new discoveries, there are also those who are engaged in mathematical research exclusively in category theory itself. However, a significant amount of research is aimed at applying category theory across various fields, including mathematics, computer science, cognitive science, physics, and programming [27, 30, 21]. Named as the “mathematics of mathematics,” [3] category theory provides an elegant structure for studying relationships and identifying universal constructs among many objects of study.

As a field of interest in this thesis, category theory is applied to the study of the relationships among signs and symbols in semiotics. Harris [14] illustrated how signs can be conceptualized as objects within a categorical framework. Goguen expanded on this idea through his work on algebraic semiotics, which formalizes the connections between semiotic elements and mathematical structures [13, 20]. In another paper [12], the author introduces a semiotic approach to understanding user interfaces, which focuses on the formal representation of meaning and interaction, establishing a framework for structured user interactions.

Goguen emphasizes that considering an interface without the user is meaningless. For example, a book is the interface connecting the text with the reader, and a coffee cup is the interface between the coffee and the drinker. In the case of computer systems, the system’s interface mediates between the abstract machine and its visually represented application, consisting of buttons, forms, and so on, with the computer itself serving as the bridge between the application and the user. Goguen describes interfaces as semiotic mappings from one system to another [13]. However, our work takes a different approach, proposing that the interface is a pullback universal construction that encompasses both the abstract machine and human interpretation.

Conceptual blending, developed by Fauconnier and Turner, examines how people combine different mental spaces to create new meanings, which underlies mixed forms of human expression, including language and art [7]. The theory states that blending involves four components: the input spaces, the common space, the blended space, and the mapping between these spaces. From a category theory perspective, we can create conceptual blending using constructs such as objects and morphisms, where each input space represents an object in the category, and the map-

pings between the input spaces and the general space are regarded as morphisms. A mixed space displays universal properties similar to colimits in the category theory, representing a unique synthesis of input spaces based on defined mappings. Goguen referred to Fauconnier’s and Turner’s concepts of conceptual blending, highlighting the importance of semantic correspondence in the chosen metaphors [13, 10]. However, a drawback of the approach is the need for concept classifications for automatic mapping, which is particularly relevant for user interfaces that depend on an existing taxonomy of buttons and interface elements, but requires significant manual effort to match perceived capabilities with these terms.

Many papers build upon Goguen’s theory of signs, including an empirical study focused on a framework based on this system of signs [15, 16], and a method of construction of the *interaction diagrams* [28]. Shifting from a sign framework, Schorlemmer and Plaza introduced a computational model for Turner’s conceptual blending [29]. They stated that the proposed *amalgamas* are equivalent to the  $\frac{3}{2}$ -pushouts defined by Goguen [13], that Goguen also used in the formal representations of *concepts* [11]. Additional contributions in applying category theory to Human-Computer Interaction contain the use of categorical structures to formalize interactions between users and systems [6], the development of recommendation systems for mobile interface patterns [17], the management of relationships among components in multimodal system design [22], and an introduction of conditional categorical activity model for describing user behavior and emotions using category and activity theories [1].

## How To Read This Thesis

In this work, we aim to provide a formal representation of user interfaces that is computationally realizable and enables structured analysis of accessibility and the unambiguous functioning of interfaces with the aim of category theory. The work is organized into seven main chapters, including introduction. We begin by defining a category of directed labeled graphs, providing proofs for existence and constructive description of limits and colimits in the category of directed labeled multigraphs, and computational implementation. Chapter 3 introduces interface-driven graphs, which we combine within a universal design framework supported by examples. We compare physical and virtual interfaces and discuss how they can be modeled with the provided graphs. The following chapter 4 presents our approach to user profiles, and how they are encoded within a graph-based representation. Chapter 5 explores the mathematical aspects of interface unambiguity, considering the roles of available controls, their influence on states, and how users perceive results with provided computational implementation. The unambiguity hypotheses are primarily based on physical devices, limiting current assumptions about their applicability to virtual interfaces. Lastly, we provide brief evidence for formalizing interfaces to event-driven types, where not all state transitions are user-controlled.

# Chapter 2

## Category Theory Framework

This chapter introduces the foundational concepts of category theory and applies them to formalize the category of directed labeled multigraphs (DLMG). Category theory's abstract structure allows us to define relationships and constructions of particular mathematical structures.

We will apply a categorical framework to construct directed labeled multigraphs that represent user-centered interfaces and define uniqueness conditions for pullback structures. In order to do this, we need to develop the framework and prove the existence and constructive descriptions of universal structures within the category of directed labeled multigraphs.

The chapter begins with basic terminology and definitions essential to understanding categories and then studies universal constructions within the DLMG category. The provided proofs are, to some extent, based on those in the referenced work [23], although they consider slightly different categories of graphs and do not include proofs for the pullback construction.

### 2.1 Terminology

**Definition 2.1.1.** A category  $C$  consists of a 5-tuple  $(O, M, s, t, c)$ , where

- $O$  is the set of objects;
- $M$  is the set of morphisms;
- $s : M \rightarrow O$  assigns each morphism its source object;
- $t : M \rightarrow O$  assigns each morphism its target object;

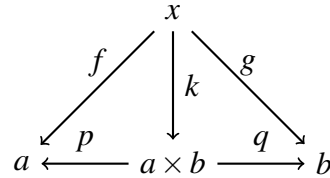
- $c : D \rightarrow M$  defines the composition of morphisms,  
where  $D = \{(a, b) \in M \times M \mid s(b) = t(a)\}$  is the set of composable morphisms.

In any category  $C$ , composition of morphisms is associative, and each object has an identity morphism satisfying the required unit laws. An essential concept in category theory is universal properties, which are extreme cases of specific constructions in the categories. Each universal construction is an initial object in some category.

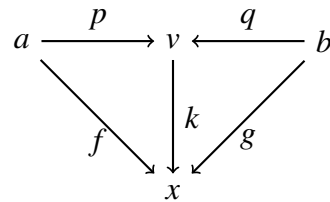
**Definition 2.1.2.** An object  $I$  in a category  $C$  is called *initial* if for every object  $X \in C$ , there exists a unique morphism from  $I$  to  $X$ .

**Definition 2.1.3.** An object  $T$  in a category  $C$  is called *terminal* if for every object  $X \in C$ , there exists a unique morphism from  $X$  to  $T$ .

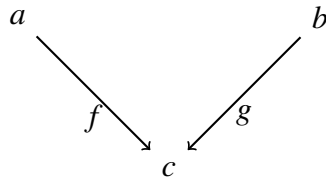
**Definition 2.1.4.** Given objects  $a$  and  $b$  in a category  $C$ , their *product* is an object  $a \times b$  together with morphisms  $p : a \times b \rightarrow a$  and  $q : a \times b \rightarrow b$ , such that for any object  $x \in C$  with morphisms  $f$  and  $g$  to  $a$  and  $b$  respectively, there exists a unique morphism  $k : x \rightarrow a \times b$  that makes the following diagram commute,



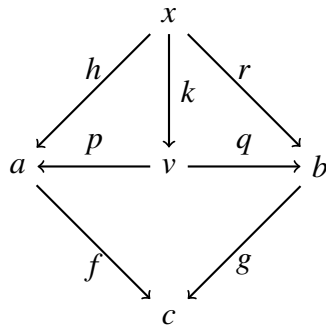
**Definition 2.1.5.** Given objects  $a$  and  $b$  in a category  $C$ , their *coproduct* is an object  $v$  together with morphisms  $p : a \rightarrow v$  and  $q : b \rightarrow v$ , such that for any object  $x \in C$  with morphisms  $f$  and  $g$  from  $a$  and  $b$  respectively, there exists a unique morphism  $k : v \rightarrow x$  making the following diagram commute,



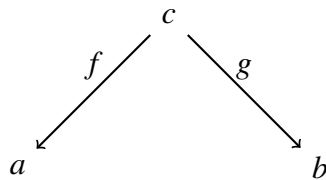
**Definition 2.1.6.** Given the following diagram in the category  $C$ ,



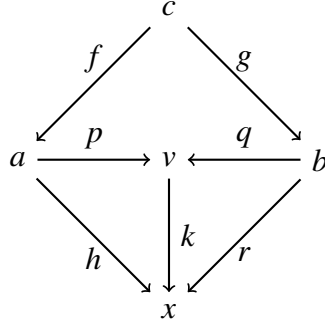
A *pullback* of this diagram is an object  $v$  equipped with morphisms  $p : v \rightarrow a$  and  $q : v \rightarrow b$ , such that for any object  $x \in C$  with morphisms  $h$  and  $r$  to  $a$  and  $b$  which make the outer diagram commute, there exists a unique morphism  $k : x \rightarrow v$  that makes the following diagram commute,



**Definition 2.1.7.** Given the following diagram in the category  $C$ ,



A *pushout* of this diagram is an object  $v$  equipped with morphisms  $p : a \rightarrow v$  and  $q : b \rightarrow v$ , such that for any object  $x \in C$  with morphisms  $h$  and  $r$  from  $a$  and  $b$  which make the outer diagram commute, there exists a unique morphism  $k : v \rightarrow x$  that makes the following diagram commute,



Category theory enables us to generalize mathematical concepts, capturing their essential structure without getting lost in the specifics. However, to work within a particular category, we must sometimes shift our focus back to the details. This is what we will do in the following sections, where we explore specific categories in more depth.

## 2.2 Category of Directed labeled Multigraphs

The specific category that we consider and computationally implement is the category of *directed labeled multigraphs* (*DLMG*). The objects of this category are directed labeled multigraphs, and the arrows represent morphisms (structure-preserving mappings) between them.

**Definition 2.2.1.** A directed labeled multigraph is a 6-tuple  $\Gamma = (V, E, \Lambda, \sigma : E \rightarrow V, \tau : E \rightarrow V, \lambda : E \rightarrow \Lambda)$ , where

- $V$  is the set of vertices,
- $E$  is the set of edges,
- $\Lambda$  is a finite alphabet of edge labels,
- $\sigma : E \rightarrow V$  is the function assigning a tail to the edge,
- $\tau : E \rightarrow V$  is the function assigning a head to the edge,
- $\lambda : E \rightarrow \Lambda$  is the function giving labels to edges.

**Definition 2.2.2.** A morphism  $\phi : \Gamma \rightarrow \Gamma'$  in the category *DLMG* is a 3-tuple  $\phi = (\phi^{vertex} : V \rightarrow V', \phi^{edge} : E \rightarrow E', \phi^{label} : \Lambda \rightarrow \Lambda')$  such that

- $\sigma_{\Gamma'}(\phi^{edge}(e)) = \phi^{vertex}(\sigma_{\Gamma}(e)),$
- $\tau_{\Gamma'}(\phi^{edge}(e)) = \phi^{vertex}(\tau_{\Gamma}(e)),$
- $\lambda_{\Gamma'}(\phi^{edge}(e)) = \phi^{label}(\lambda_{\Gamma}(e)).$

## 2.2.1 Universal Constructions in *DLMG* Category

**Proposition 1.** *Initial object in the DLMG category is an empty graph.*

*Proof.* Trivially, for every graph  $\Gamma$  there is the unique morphism from an empty graph to  $\Gamma$ .  $\square$

**Proposition 2.** *Terminal object in the DLMG category is a graph with exactly one vertex, one edge label and one edge.*

*Proof.* For every graph  $\Gamma$  there exists the unique morphism to the graph with exactly one vertex and one labeled edge, where all vertices of the graph  $\Gamma$  are mapped to one vertex, all labels are mapped to one label, and all edges are mapped to a single loop.  $\square$

## 2.2.2 Product

**Definition 2.2.3.** Let  $A = (V_A, E_A, \Lambda_A, \sigma_A : E_A \rightarrow V_A, \tau_A : E_A \rightarrow V_A, \lambda_A : E_A \rightarrow \Lambda_A)$  and  $B = (V_B, E_B, \Lambda_B, \sigma_B : E_B \rightarrow V_B, \tau_B : E_B \rightarrow V_B, \lambda_B : E_B \rightarrow \Lambda_B)$  be two directed labeled multigraphs. The *product* of these graphs is a directed labeled multigraph  $\Pi = (V_{\Pi}, E_{\Pi}, \Lambda_{\Pi}, \sigma_{\Pi} : E_{\Pi} \rightarrow V_{\Pi}, \tau_{\Pi} : E_{\Pi} \rightarrow V_{\Pi}, \lambda_{\Pi} : E_{\Pi} \rightarrow \Lambda_{\Pi})$ , where

- $V_{\Pi} = V_A \times V_B,$
- $E_{\Pi} = E_A \times E_B,$
- $\Lambda_{\Pi} = \Lambda_A \times \Lambda_B,$
- $\sigma_{\Pi} : E_{\Pi} \rightarrow V_{\Pi}, e = (e_a, e_b) \mapsto (\sigma_A(e_a), \sigma_B(e_b)),$
- $\tau_{\Pi} : E_{\Pi} \rightarrow V_{\Pi}, e = (e_a, e_b) \mapsto (\tau_A(e_a), \tau_B(e_b)),$
- $\lambda_{\Pi} : E_{\Pi} \rightarrow \Lambda_{\Pi}, e = (e_a, e_b) \mapsto (\lambda_A(e_a), \lambda_B(e_b)).$

Additionally, we define two morphisms from  $\Pi$  to  $A$  and  $B$ , respectively. The morphism  $\pi_A : \Pi \rightarrow A$  is given by,

- $\pi_A^{vertex} : V_\Pi \rightarrow V_A, (v_a, v_b) \mapsto v_a,$
- $\pi_A^{edge} : E_\Pi \rightarrow E_A, (e_a, e_b) \mapsto e_a,$
- $\pi_A^{label} : \Lambda_\Pi \rightarrow \Lambda_A, (l_a, l_b) \mapsto l_a.$

The morphism  $\pi_B : \Pi \rightarrow B$  is given by

- $\pi_B^{vertex} : V_\Pi \rightarrow V_B, (v_a, v_b) \mapsto v_b,$
- $\pi_B^{edge} : E_\Pi \rightarrow E_B, (e_a, e_b) \mapsto e_b,$
- $\pi_B^{label} : \Lambda_\Pi \rightarrow \Lambda_B, (l_a, l_b) \mapsto l_b.$

**Proposition 3.** *The product  $\Pi$  is the terminal object over the diagram consisting of two objects  $A$  and  $B$ , and no morphisms.*

*Proof.* To show that the product  $\Pi$  is terminal above a diagram consisting of two objects  $A$  and  $B$  and without any morphisms, we must show that for any graph  $\Gamma$  together with morphisms  $\gamma_A : \Gamma \rightarrow A$  and  $\gamma_B : \Gamma \rightarrow B$ , there exists a unique morphism  $\phi : \Gamma \rightarrow \Pi$  making the following diagram commute.

$$\begin{array}{ccccc}
 & & \Gamma & & \\
 & \swarrow \gamma_A & \downarrow \phi & \searrow \gamma_B & \\
 A & \xleftarrow{\pi_A} & \Pi & \xrightarrow{\pi_B} & B
 \end{array}$$

First, we will show that morphisms  $\pi_A$  and  $\pi_B$  are well-defined. For every edge  $e = (e_A, e_B) \in E_\Pi$ ,

$$\begin{aligned}
 \sigma_A(\pi_A^{edge}(e)) &= \sigma_A(\pi_A^{edge}(e_A, e_B)) \\
 &= \sigma_A(e_A) \\
 &= \pi_A^{vertex}(\sigma_\Pi(e_A, e_B)) \\
 &= \pi_A^{vertex}(\sigma_\Pi(e)),
 \end{aligned}$$

and

$$\begin{aligned}
\tau_A(\pi_A^{edge}(e)) &= \tau_A(\pi_A^{edge}(e_A, e_B)) \\
&= \tau_A(e_A) \\
&= \pi_A^{vertex}(\tau_\Pi(e_A, e_B)) \\
&= \pi_A^{vertex}(\tau_\Pi(e)),
\end{aligned}$$

and

$$\begin{aligned}
\lambda_A(\pi_A^{edge}(e)) &= \lambda_A(\pi_A^{edge}(e_A, e_B)) \\
&= \lambda_A(e_A) \\
&= \pi_A^{label}(\lambda_\Pi(e_A, e_B)) \\
&= \pi_A^{label}(\lambda_\Pi(e)).
\end{aligned}$$

Similarly,  $\pi_B$  is well-defined. Then, the morphism  $\phi : \Gamma \rightarrow \Pi$  is given by:

- $\phi^{vertex} : V_\Gamma \rightarrow V_\Pi$ ,  
 $v_\Gamma \mapsto (\gamma_A^{vertex}(v_\Gamma), \gamma_B^{vertex}(v_\Gamma)),$
- $\phi^{edge} : E_\Gamma \rightarrow E_\Pi$ ,  
 $e_\Gamma \mapsto (\gamma_A^{edge}(e_\Gamma), \gamma_B^{edge}(e_\Gamma)),$
- $\phi^{label} : \Lambda_\Gamma \rightarrow \Lambda_\Pi$ ,  
 $\lambda_\Gamma \mapsto (\gamma_A^{label}(\lambda_\Gamma), \gamma_B^{label}(\lambda_\Gamma)).$

Let us show that morphism  $\phi$  is well-defined. For every edge  $e_\Gamma \in E_\Gamma$ ,

$$\begin{aligned}
\sigma_\Pi(\phi^{edge}(e_\Gamma)) &= \sigma_\Pi(\gamma_A^{edge}(e_\Gamma), \gamma_B^{edge}(e_\Gamma)) \\
&= (\sigma_A(\gamma_A^{edge}(e_\Gamma)), \sigma_B(\gamma_B^{edge}(e_\Gamma))) \\
&= (\gamma_A^{vertex}(\sigma_\Gamma(e_\Gamma)), \gamma_B^{vertex}(\sigma_\Gamma(e_\Gamma))) \\
&= \phi^{vertex}(\sigma_\Gamma(e_\Gamma)) \in V_\Pi,
\end{aligned}$$

and

$$\begin{aligned}
\tau_\Pi(\phi^{edge}(e_\Gamma)) &= \tau_\Pi(\gamma_A^{edge}(e_\Gamma), \gamma_B^{edge}(e_\Gamma)) \\
&= (\tau_A(\gamma_A^{edge}(e_\Gamma)), \tau_B(\gamma_B^{edge}(e_\Gamma))) \\
&= (\gamma_A^{vertex}(\tau_\Gamma(e_\Gamma)), \gamma_B^{vertex}(\tau_\Gamma(e_\Gamma))) \\
&= \phi^{vertex}(\tau_\Gamma(e_\Gamma)) \in V_\Pi,
\end{aligned}$$

and

$$\begin{aligned}
\lambda_\Pi(\phi^{edge}(e_\Gamma)) &= \lambda_\Pi(\gamma_A^{edge}(e_\Gamma), \gamma_B^{edge}(e_\Gamma)) \\
&= (\lambda_A(\gamma_A^{edge}(e_\Gamma)), \lambda_B(\gamma_B^{edge}(e_\Gamma))) \\
&= (\gamma_A^{label}(\lambda_\Gamma(e_\Gamma)), \gamma_B^{label}(\lambda_\Gamma(e_\Gamma))) \\
&= \phi^{label}(\lambda_\Gamma(e_\Gamma)) \in \Lambda_\Pi.
\end{aligned}$$

Therefore,  $\phi$  is well-defined. It is clear that the diagram commutes. Indeed, for every vertex  $v_\Gamma \in V_\Gamma$ ,

$$\pi_A^{vertex}(\phi^{vertex}(v_\Gamma)) = \pi_A^{vertex}(\gamma_A^{vertex}(v_A), \gamma_B^{vertex}(v_B)) = \gamma_A^{vertex}(v_A),$$

$$\pi_B^{vertex}(\phi^{vertex}(v_\Gamma)) = \pi_B^{vertex}(\gamma_A^{vertex}(v_A), \gamma_B^{vertex}(v_B)) = \gamma_B^{vertex}(v_B).$$

Similarly, for every edge  $e_\Gamma \in E_\Gamma$ ,

$$\pi_A^{edge}(\phi^{edge}(e_\Gamma)) = \gamma_A^{edge}(e_\Gamma),$$

$$\pi_B^{edge}(\phi^{edge}(e_\Gamma)) = \gamma_B^{edge}(e_\Gamma),$$

and for every label  $\lambda_\Gamma \in \Lambda_\Gamma$ ,

$$\pi_A^{label}(\phi^{label}(\lambda_\Gamma)) = \gamma_A^{label}(\lambda_\Gamma),$$

$$\pi_B^{label}(\phi^{label}(\lambda_\Gamma)) = \gamma_B^{label}(\lambda_\Gamma).$$

Showing that  $\phi$  is unique completes the proof. Suppose there is another morphism  $\phi_2$  such that the diagram commutes. For every vertex  $v_\Gamma \in V_\Gamma$ , we put  $\phi_2^{vertex}(v_\Gamma) = (v_A, v_B)$ . Then,

$$\pi_A^{vertex}(\phi_2^{vertex}(v_\Gamma)) = \gamma_A^{vertex}(v_\Gamma) \implies v_A = \gamma_A^{vertex}(v_\Gamma),$$

and similarly,

$$v_B = \gamma_B^{vertex}(v_\Gamma).$$

Therefore,

$$\phi_2^{vertex}(v_\Gamma) = (\gamma_A^{vertex}(v_\Gamma), \gamma_B^{vertex}(v_\Gamma)) = \phi^{vertex}(v_\Gamma).$$

For every edge  $e_\Gamma \in E_\Gamma$ , we put  $\phi_2^{edge}(e_\Gamma) = (e_A, e_B)$ . Then,

$$\pi_A^{edge}(\phi_2^{edge}(e_\Gamma)) = \gamma_A^{edge}(e_\Gamma) \implies e_A = \gamma_A^{edge}(e_\Gamma),$$

and similarly,

$$e_B = \gamma_B^{edge}(e_\Gamma).$$

Therefore,

$$\phi_2^{edge}(e_\Gamma) = (\gamma_A^{edge}(e_\Gamma), \gamma_B^{edge}(e_\Gamma)) = \phi^{edge}(e_\Gamma).$$

Finally, for every label  $\lambda_\Gamma \in \Lambda_\Gamma$ , we put  $\phi_2^{label}(\lambda_\Gamma) = (\lambda_A, \lambda_B)$ . Then,

$$\pi_A^{label}(\phi_2^{label}(\lambda_\Gamma)) = \gamma_A^{label}(\lambda_\Gamma) \implies \lambda_A = \gamma_A^{label}(\lambda_\Gamma),$$

and similarly,

$$\lambda_B = \gamma_B^{label}(\lambda_\Gamma).$$

Therefore,

$$\phi_2^{label}(\lambda_\Gamma) = (\gamma_A^{label}(\lambda_\Gamma), \gamma_B^{label}(\lambda_\Gamma)) = \phi^{label}(\lambda_\Gamma).$$

We conclude that  $\phi$  is unique, and the proof is complete.  $\square$

### 2.2.3 Coproduct

**Definition 2.2.4.** Let  $A = (V_A, E_A, \Lambda_A, \sigma_A : E_A \rightarrow V_A, \tau_A : E_A \rightarrow V_A, \lambda_A : E_A \rightarrow \Lambda_A)$  and  $B = (V_B, E_B, \Lambda_B, \sigma_B : E_B \rightarrow V_B, \tau_B : E_B \rightarrow V_B, \lambda_B : E_B \rightarrow \Lambda_B)$  be two directed labeled multigraphs. The *coproduct* of these graphs is a directed labeled multigraph  $\Delta = (V_\Delta, E_\Delta, \Lambda_\Delta, \sigma_\Delta : E_\Delta \rightarrow V_\Delta, \tau_\Delta : E_\Delta \rightarrow V_\Delta, \lambda_\Delta : E_\Delta \rightarrow \Lambda_\Delta)$ , where

- $V_\Delta = V_A \sqcup V_B$ ,
- $E_\Delta = E_A \sqcup E_B$ ,
- $\Lambda_\Delta = \Lambda_A \sqcup \Lambda_B$ ,
- $\sigma_\Delta : E_\Delta \rightarrow V_\Delta, e = (e', k) \mapsto \begin{cases} \sigma_A(e') & \text{if } k = a, \\ \sigma_B(e') & \text{if } k = b, \end{cases}$
- $\tau_\Delta : E_\Delta \rightarrow V_\Delta, e = (e', k) \mapsto \begin{cases} \tau_A(e') & \text{if } k = a, \\ \tau_B(e') & \text{if } k = b, \end{cases}$
- $\lambda_\Delta : E_\Delta \rightarrow \Lambda_\Delta, e = (e', k) \mapsto \begin{cases} \lambda_A(e') & \text{if } k = a, \\ \lambda_B(e') & \text{if } k = b, \end{cases}$

together with two directed labeled multigraph morphisms  $\delta_K : K \rightarrow \Delta$  where  $K \in \{A, B\}$  by

- $\delta_K^{vertex} : V_K \rightarrow V_\Delta$  with  $v \mapsto \begin{cases} (v, a) & \text{if } K = A, \\ (v, b) & \text{if } K = B. \end{cases}$
- $\delta_K^{edge} : E_K \rightarrow E_\Delta$  with  $e \mapsto \begin{cases} (e, a) & \text{if } K = A, \\ (e, b) & \text{if } K = B. \end{cases}$
- $\delta_K^{label} : \Lambda_K \rightarrow \Lambda_\Delta$  with  $l \mapsto \begin{cases} (l, a) & \text{if } K = A, \\ (l, b) & \text{if } K = B. \end{cases}$

**Proposition 4.** *The coproduct  $\Delta$  is the initial object over the diagram consisting of two objects  $A$  and  $B$ , and no morphisms.*

*Proof.* To show that the coproduct  $\Delta$  is initial under a diagram consisting of two objects  $A$  and  $B$  without any morphisms, we must show that for any graph  $\Gamma$  together with morphisms  $\gamma_A : A \rightarrow \Gamma$  and  $\gamma_B : B \rightarrow \Gamma$ , there exists a unique morphism  $\phi : \Delta \rightarrow \Gamma$  making the diagram commute.

$$\begin{array}{ccccc}
 A & \xrightarrow{\delta_A} & \Delta & \xleftarrow{\delta_B} & B \\
 & \searrow \gamma_A & \downarrow \phi & \swarrow \gamma_B & \\
 & & \Gamma & & 
 \end{array}$$

First, we will show that morphisms  $\delta_A$  and  $\delta_B$  are well-defined. For every edge  $e_A \in E_A$ ,

$$\begin{aligned}
 \sigma_\Delta(\delta_A^{edge}(e_A)) &= \sigma_\Delta(e_A, a) \\
 &= (\sigma_A(e_A), a) \\
 &= \delta^{vertex}(\sigma_A(e_A)).
 \end{aligned}$$

$$\begin{aligned}
 \tau_\Delta(\delta_A^{edge}(e_A)) &= \tau_\Delta(e_A, a) \\
 &= (\tau_A(e_A), a) \\
 &= \delta^{vertex}(\tau_A(e_A)).
 \end{aligned}$$

$$\begin{aligned}
\lambda_\Delta(\delta_A^{edge}(e_A)) &= \lambda_\Delta(e_A, a) \\
&= (\lambda_A(e_A), a) \\
&= \delta^{label}(\lambda_A(e_A)).
\end{aligned}$$

Similarly,  $\delta_B$  is well-defined. Then, the morphism  $\phi$  is defined as follows:

$$\begin{aligned}
- \phi^{vertex} : V_\Delta \rightarrow V_\Gamma, (v_\Gamma) = (v, k) &\mapsto \begin{cases} \gamma_A^{vertex}(v), & \text{if } k = a, \\ \gamma_B^{vertex}(v), & \text{if } k = b, \end{cases} \\
- \phi^{edge} : E_\Delta \rightarrow E_\Gamma, (e_\Gamma) = (e, k) &\mapsto \begin{cases} \gamma_A^{edge}(e), & \text{if } k = a, \\ \gamma_B^{edge}(e), & \text{if } k = b, \end{cases} \\
- \phi^{label} : \Lambda_\Delta \rightarrow \Lambda_\Gamma, (\lambda_\Gamma) = (\lambda, k) &\mapsto \begin{cases} \gamma_A^{label}(\lambda), & \text{if } k = a, \\ \gamma_B^{label}(\lambda), & \text{if } k = b. \end{cases}
\end{aligned}$$

The morphism  $\phi$  is well defined. For every edge  $e_\Delta = (e, k) \in E_\Delta$ ,

$$\begin{aligned}
\sigma_\Gamma(\phi^{edge}(e_\Delta)) &= \sigma_\Gamma(\phi^{edge}(e, k)) \\
&= \begin{cases} \sigma_\Gamma(\gamma_A^{edge}(e)), & \text{if } k = a, \\ \sigma_\Gamma(\gamma_B^{edge}(e)), & \text{if } k = b, \end{cases} \\
&= \begin{cases} \gamma_A^{vertex}(\sigma_A(e)), & \text{if } k = a, \\ \gamma_B^{vertex}(\sigma_B(e)), & \text{if } k = b, \end{cases} \\
&= \begin{cases} \phi^{vertex}(\sigma_A(e), k), & \text{if } k = a, \\ \phi^{vertex}(\sigma_B(e), k), & \text{if } k = b, \end{cases} \\
&= \phi^{vertex}(\sigma_\Delta(e, k)) \\
&= \phi^{vertex}(\sigma_\Delta(e_\Delta)).
\end{aligned}$$

$$\begin{aligned}
\tau_\Gamma(\phi^{edge}(e_\Delta)) &= \tau_\Gamma(\phi^{edge}(e, k)) \\
&= \begin{cases} \tau_\Gamma(\gamma_A^{edge}(e)), & \text{if } k = a, \\ \tau_\Gamma(\gamma_B^{edge}(e)), & \text{if } k = b, \end{cases} \\
&= \begin{cases} \gamma_A^{vertex}(\tau_A(e)), & \text{if } k = a, \\ \gamma_B^{vertex}(\tau_B(e)), & \text{if } k = b, \end{cases} \\
&= \begin{cases} \phi_A^{vertex}(\tau_A(e), k), & \text{if } k = a, \\ \phi_B^{vertex}(\tau_B(e), k), & \text{if } k = b, \end{cases} \\
&= \phi^{vertex}(\tau_\Delta(e, k)) \\
&= \phi^{vertex}(\tau_\Delta(e_\Delta)).
\end{aligned}$$

$$\begin{aligned}
\lambda_\Gamma(\phi^{edge}(e_\Delta)) &= \lambda_\Gamma(\phi^{edge}(e, k)) \\
&= \begin{cases} \lambda_\Gamma(\gamma_A^{edge}(e)), & \text{if } k = a, \\ \lambda_\Gamma(\gamma_B^{edge}(e)), & \text{if } k = b, \end{cases} \\
&= \begin{cases} \gamma_A^{label}(\lambda_A(e)), & \text{if } k = a, \\ \gamma_B^{label}(\lambda_B(e)), & \text{if } k = b, \end{cases} \\
&= \begin{cases} \phi^{label}(\lambda_A(e), k), & \text{if } k = a, \\ \phi^{label}(\lambda_B(e), k), & \text{if } k = b, \end{cases} \\
&= \phi^{label}(\lambda_\Delta(e, k)) \\
&= \phi^{label}(\lambda_\Delta(e_\Delta)).
\end{aligned}$$

It is clear that the diagram commutes. Indeed, for every vertex  $v_K \in V_K$  where  $K \in \{A, B\}$ ,

$$\begin{aligned}
\phi^{vertex}(\delta_K^{vertex}(v_K)) &= \phi^{vertex}(v_K, k), \\
&= \begin{cases} \gamma_A^{vertex}(v_K), & \text{if } K = A, \\ \gamma_B^{vertex}(v_k), & \text{if } K = B. \end{cases}
\end{aligned}$$

Similarly, for every edge  $e_K \in E_K$  where  $K \in \{A, B\}$ ,

$$\begin{aligned}
\phi^{edge}(\delta_K^{edge}(e_K)) &= \phi^{edge}(e_K, k), \\
&= \begin{cases} \gamma_A^{edge}(e_K), & \text{if } K = A, \\ \gamma_B^{edge}(e_k), & \text{if } K = B. \end{cases}
\end{aligned}$$

and for every label  $\lambda_K \in \Lambda_K$  where  $K \in \{A, B\}$ ,

$$\begin{aligned}\phi^{label}(\delta_K^{label}(\lambda_K)) &= \phi^{label}(\lambda_K, k), \\ &= \begin{cases} \gamma_A^{label}(\lambda_K), & \text{if } K = A, \\ \gamma_B^{label}(\lambda_K), & \text{if } K = B. \end{cases}\end{aligned}$$

Showing the uniqueness of morphism  $\phi$  will complete the proof. Suppose there is another morphism  $\phi_2$ . For every vertex  $v_\Delta = (v, k) \in V_\Delta$ , set  $\phi_2^{vertex}(v_\Delta) = v_\Gamma$ . Then,

$$\begin{aligned}v_\Gamma &= \phi_2^{vertex}(v_\Delta) \\ &= \phi_2^{vertex}(v, k) \\ &= \begin{cases} \phi_2^{vertex}(\delta_A^{vertex}(v)), & \text{if } k = a, \\ \phi_2^{vertex}(\delta_B^{vertex}(v)), & \text{if } k = b, \end{cases} \\ &= \begin{cases} \gamma_A^{vertex}(v), & \text{if } k = a, \\ \gamma_B^{vertex}(v), & \text{if } k = b, \end{cases} \\ &= \phi^{vertex}(v, k) \\ &= \phi^{vertex}(v_\Delta).\end{aligned}$$

For every edge  $e_\Delta = (e, k) \in E_\Delta$ , set  $\phi_2^{edge}(e_\Delta) = e_\Gamma$ . Then,

$$\begin{aligned}e_\Gamma &= \phi_2^{edge}(e_\Delta) \\ &= \phi_2^{edge}(e, k) \\ &= \begin{cases} \phi_2^{edge}(\delta_A^{edge}(e)), & \text{if } k = a, \\ \phi_2^{edge}(\delta_B^{edge}(e)), & \text{if } k = b, \end{cases} \\ &= \begin{cases} \gamma_A^{edge}(e), & \text{if } k = a, \\ \gamma_B^{edge}(e), & \text{if } k = b, \end{cases} \\ &= \phi^{edge}(e, k) \\ &= \phi^{edge}(e_\Delta).\end{aligned}$$

For every label  $\lambda_\Delta = (\lambda, k) \in \Lambda_\Delta$ , set  $\phi_2^{label}(\lambda_\Delta) = \lambda_\Gamma$ . Then,

$$\begin{aligned}
\lambda_\Gamma &= \phi_2^{label}(\lambda_\Delta) \\
&= \phi_2^{label}(\lambda, k) \\
&= \begin{cases} \phi_2^{label}(\delta_A^{label}(\lambda)), & \text{if } k = a, \\ \phi_2^{label}(\delta_B^{label}(\lambda)), & \text{if } k = b, \end{cases} \\
&= \begin{cases} \gamma_A^{label}(\lambda), & \text{if } k = a, \\ \gamma_B^{label}(\lambda), & \text{if } k = b, \end{cases} \\
&= \phi^{label}(\lambda, k) \\
&= \phi^{label}(\lambda_\Delta).
\end{aligned}$$

Thus, the morphism  $\phi$  is unique. Therefore, we have shown that the coproduct  $\Delta$  is initial under the given conditions.  $\square$

## 2.2.4 Pullback

**Definition 2.2.5.** Let  $A = (V_A, E_A, \Lambda_A, \sigma_A : E_A \rightarrow V_A, \tau_A : E_A \rightarrow V_A, \lambda_A : E_A \rightarrow \Lambda_A)$ ,  $B = (V_B, E_B, \Lambda_B, \sigma_B : E_B \rightarrow V_B, \tau_B : E_B \rightarrow V_B, \lambda_B : E_B \rightarrow \Lambda_B)$ , and  $C = (V_C, E_C, \Lambda_C, \sigma_C : E_C \rightarrow V_C, \tau_C : E_C \rightarrow V_C, \lambda_C : E_C \rightarrow \Lambda_C)$  be two directed labeled multigraphs,  $f : A \rightarrow C$  and  $g : B \rightarrow C$  be two graph morphisms. The *pullback* above the diagram  $A \xrightarrow{f} C \xleftarrow{g} B$  is the directed labeled multigraph  $P = (V_P, E_P, \Lambda_P, \sigma_P : E_P \rightarrow V_P, \tau_P : E_P \rightarrow V_P, \lambda_P : E_P \rightarrow \Lambda_P)$ , where

- $V_P = V_A \times_C V_B = \{(v_a, v_b) : f^{vertex}(v_a) = g^{vertex}(v_b), v_a \in V_A \text{ \& } v_b \in V_B\}$ ,
- $E_P = E_A \times_C E_B = \{(e_a, e_b) : f^{edge}(e_a) = g^{edge}(e_b), e_a \in E_A \text{ \& } e_b \in E_B\}$ ,
- $\Lambda_P = \Lambda_A \times_C \Lambda_B = \{(l_a, l_b) : f^{label}(l_a) = g^{label}(l_b), l_a \in \Lambda_A \text{ \& } l_b \in \Lambda_B\}$ ,
- $\sigma_P : E_P \rightarrow V_P, e = (e_a, e_b) \mapsto (\sigma_A(e_a), \sigma_B(e_b))$ ,
- $\tau_P : E_P \rightarrow V_P, e = (e_a, e_b) \mapsto (\tau_A(e_a), \tau_B(e_b))$ ,
- $\lambda_P : E_P \rightarrow \Lambda_P, e = (e_a, e_b) \mapsto (\lambda_A(e_a), \lambda_B(e_b))$ ,

together with two morphisms from  $P$  to  $A$  and  $B$ , respectively. The morphism  $\pi_A : P \rightarrow A$  is given by,

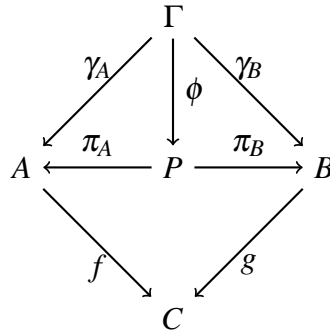
- $\pi_A^{label} : \Lambda_P \rightarrow \Lambda_A, (l_a, l_b) \mapsto l_a.$

The morphism  $\pi_B : P \rightarrow B$  is given by

- $\pi_B^{label} : \Lambda_P \rightarrow \Lambda_B, (l_a, l_b) \mapsto l_b.$

**Proposition 5.** *The pullback  $P$  is the terminal object over the diagram  $A \xrightarrow{f} C \xleftarrow{g} B$ .*

*Proof.* To show that the pullback  $P$  is terminal over a diagram  $A \xrightarrow{f} C \xleftarrow{g} B$ , we must show that for any graph  $\Gamma$  together with morphisms  $\gamma_A : \Gamma \rightarrow A$  and  $\gamma_B : \Gamma \rightarrow B$  such that the outer diagram commutes, meaning  $\gamma_A \circ f = \gamma_B \circ g$ , there exists a unique morphism  $\phi : \Gamma \rightarrow P$  such that the diagram commutes.



It was shown in the Proposition 3 that morphisms  $\pi_A$  and  $\pi_B$  are well-defined. Then, we will show that the diagram commutes. For every vertex  $v_P = (v_A, v_B) \in V_P$ ,

$$\begin{aligned} f^{vertex}(\pi_A^{vertex}(v_A, v_B)) &= f^{vertex}(v_A) \\ &= g^{vertex}(v_B) \\ &= g^{vertex}(\pi_B^{vertex}(v_A, v_B)). \end{aligned}$$

For every label  $\lambda_P = (\lambda_A, \lambda_B) \in \Lambda_P$ ,

$$\begin{aligned} f^{label}(\pi_A^{label}(\lambda_A, \lambda_B)) &= f^{label}(\lambda_A) \\ &= g^{label}(\lambda_B) \\ &= g^{label}(\pi_B^{label}(\lambda_A, \lambda_B)). \end{aligned}$$

For every edge  $e_P = (e_A, e_B) \in E_P$ , since

$$f^{vertex}(\pi_A^{vertex}(\sigma(e_P))) = g^{vertex}(\pi_A^{vertex}(\sigma(e_P))),$$

similarly for  $\tau(e_P)$  and  $\lambda(e_P)$ , we get,

$$\begin{aligned} f^{edge}(\pi_A^{edge}(e_A, e_B)) &= f^{edge}(e_A) \\ &= g^{edge}(e_B) \\ &= g^{edge}(\pi_B^{edge}(e_A, e_B)). \end{aligned}$$

Then, the morphism  $\phi : \Gamma \rightarrow P$  is defined as follows:

- $\phi^{vertex} : V_\Gamma \rightarrow V_P, v_\Gamma \mapsto (\gamma_A(v_\Gamma), \gamma_B(v_\Gamma))$ ,
- $\phi^{edge} : E_\Gamma \rightarrow E_P, e_\Gamma \mapsto (\gamma_A(e_\Gamma), \gamma_B(e_\Gamma))$ ,
- $\phi^{label} : \Lambda_\Gamma \rightarrow \Lambda_P, \lambda_\Gamma \mapsto (\gamma_A(\lambda_\Gamma), \gamma_B(\lambda_\Gamma))$ .

Let us show that morphism  $\phi$  is well-defined. For every edge  $e_\Gamma \in E_\Gamma$ ,

$$\begin{aligned} \sigma_P(\phi^{edge}(e_\Gamma)) &= \sigma_P(\gamma_A^{edge}(e_\Gamma), \gamma_B^{edge}(e_\Gamma)) \\ &= (\sigma_A(\gamma_A^{edge}(e_\Gamma)), \sigma_B(\gamma_B^{edge}(e_\Gamma))) \\ &= (\gamma_A^{vertex}(\sigma_\Gamma(e_\Gamma)), \gamma_B^{vertex}(\sigma_\Gamma(e_\Gamma))) \\ &= \phi^{vertex}(\sigma_\Gamma(e_\Gamma)) \in V_P, \end{aligned}$$

and

$$\begin{aligned} \tau_P(\phi^{edge}(e_\Gamma)) &= \tau_P(\gamma_A^{edge}(e_\Gamma), \gamma_B^{edge}(e_\Gamma)) \\ &= (\tau_A(\gamma_A^{edge}(e_\Gamma)), \tau_B(\gamma_B^{edge}(e_\Gamma))) \\ &= (\gamma_A^{vertex}(\tau_\Gamma(e_\Gamma)), \gamma_B^{vertex}(\tau_\Gamma(e_\Gamma))) \\ &= \phi^{vertex}(\tau_\Gamma(e_\Gamma)) \in V_P, \end{aligned}$$

and

$$\begin{aligned}
\lambda_P(\phi^{edge}(e_\Gamma)) &= \lambda_P(\gamma_A^{edge}(e_\Gamma), \gamma_B^{edge}(e_\Gamma)) \\
&= (\lambda_A(\gamma_A^{edge}(e_\Gamma)), \lambda_B(\gamma_B^{edge}(e_\Gamma))) \\
&= (\gamma_A^{label}(\lambda_\Gamma(e_\Gamma)), \gamma_B^{label}(\lambda_\Gamma(e_\Gamma))) \\
&= \phi^{label}(\lambda_\Gamma(e_\Gamma)) \in \Lambda_P.
\end{aligned}$$

Therefore,  $\phi$  is well-defined. It is clear that the diagram commutes. Indeed, for every vertex  $v_\Gamma \in V_\Gamma$ ,

$$\pi_A^{vertex}(\phi^{vertex}(v_\Gamma)) = \pi_A^{vertex}(\gamma_A^{vertex}(v_A), \gamma_B^{vertex}(v_B)) = \gamma_A^{vertex}(v_A),$$

$$\pi_B^{vertex}(\phi^{vertex}(v_\Gamma)) = \pi_B^{vertex}(\gamma_A^{vertex}(v_A), \gamma_B^{vertex}(v_B)) = \gamma_B^{vertex}(v_B).$$

Similarly, for every edge  $e_\Gamma \in E_\Gamma$ ,

$$\pi_A^{edge}(\phi^{edge}(e_\Gamma)) = \gamma_A^{edge}(e_\Gamma),$$

$$\pi_B^{edge}(\phi^{edge}(e_\Gamma)) = \gamma_B^{edge}(e_\Gamma),$$

and for every label  $\lambda_\Gamma \in \Lambda_\Gamma$ ,

$$\pi_A^{label}(\phi^{label}(\lambda_\Gamma)) = \gamma_A^{label}(\lambda_\Gamma),$$

$$\pi_B^{label}(\phi^{label}(\lambda_\Gamma)) = \gamma_B^{label}(\lambda_\Gamma).$$

Therefore, showing uniqueness of the morphism  $\phi$  completes the proof. Suppose there is another morphism  $\phi_2$  such that the diagram commutes. For every vertex  $v_\Gamma \in V_\Gamma$ , put  $\phi_2^{vertex}(v_\Gamma) = (v_A, v_B)$ . Then,

$$\pi_A^{vertex}(\phi_2^{vertex}(v_\Gamma)) = \gamma_A^{vertex}(v_\Gamma) \implies v_A = \gamma_A^{vertex}(v_\Gamma),$$

and similarly,

$$v_B = \gamma_B^{vertex}(v_\Gamma).$$

Therefore,  $\phi_2^{\text{vertex}}(v_\Gamma) = (\gamma_A^{\text{vertex}}(v_\Gamma), \gamma_B^{\text{vertex}}(v_\Gamma)) = \phi^{\text{vertex}}(v_\Gamma)$ . For every edge  $e_\Gamma \in E_\Gamma$ , put  $\phi_2^{\text{edge}}(e_\Gamma) = (e_A, e_B)$ . Then,

$$\pi_A^{\text{edge}}(\phi_2^{\text{edge}}(e_\Gamma)) = \gamma_A^{\text{edge}}(e_\Gamma) \implies e_A = \gamma_A^{\text{edge}}(e_\Gamma),$$

and similarly,

$$e_B = \gamma_B^{\text{edge}}(e_\Gamma).$$

Therefore,  $\phi_2^{\text{edge}}(e_\Gamma) = (\gamma_A^{\text{edge}}(e_\Gamma), \gamma_B^{\text{edge}}(e_\Gamma)) = \phi^{\text{edge}}(e_\Gamma)$ . Finally, for every label  $\lambda_\Gamma \in \Lambda_\Gamma$ , put  $\phi_2^{\text{label}}(\lambda_\Gamma) = (\lambda_A, \lambda_B)$ . Then,

$$\pi_A^{\text{label}}(\phi_2^{\text{label}}(\lambda_\Gamma)) = \gamma_A^{\text{label}}(\lambda_\Gamma) \implies \lambda_A = \gamma_A^{\text{label}}(\lambda_\Gamma),$$

and similarly,

$$\lambda_B = \gamma_B^{\text{label}}(\lambda_\Gamma).$$

Therefore,  $\phi_2^{\text{label}}(\lambda_\Gamma) = (\gamma_A^{\text{label}}(\lambda_\Gamma), \gamma_B^{\text{label}}(\lambda_\Gamma)) = \phi^{\text{label}}(\lambda_\Gamma)$ . Consequently,  $\phi_2 = \phi$ . Thus, we conclude that the pullback  $P$  is terminal over the diagram  $A \xrightarrow{f} C \xleftarrow{g} B$ .  $\square$

**Lemma 2.2.0.1.** *Let  $P$  be a pullback on above the diagram  $A \xrightarrow{f} C \xleftarrow{g} B$ .  $P$  is isomorphic to the product  $\Pi$  above the diagram consisting of objects  $A$  and  $B$ , if  $C$  is terminal object in the **DLMG** category.*

*Proof.* The graph  $C$  is a terminal object, that is, a graph with one vertex  $v_C \in V_C$ , one loop  $e_C \in E_C$ , and one label  $\lambda_C \in \Lambda_C$ . The following holds:

$$\forall v_A \in V_A \ \forall v_B \in V_B : f^{\text{vertex}}(v_A) = g^{\text{vertex}}(v_B) = v_C \implies (v_A, v_B) \in V_P,$$

$$\forall e_A \in E_A \ \forall e_B \in E_B : f^{\text{edge}}(e_A) = g^{\text{edge}}(e_B) = e_C \implies (e_A, e_B) \in E_P,$$

$$\forall \lambda_A \in \Lambda_A \ \forall \lambda_B \in \Lambda_B : f^{\text{label}}(\lambda_A) = g^{\text{label}}(\lambda_B) = \lambda_C \implies (\lambda_A, \lambda_B) \in \Lambda_P.$$

Therefore, there exist isomorphism  $\phi : P \rightarrow \Pi$  that trivially maps every vertex, edge, and label to itself.  $\square$

**Lemma 2.2.0.2.** *Let  $P$  be a pullback on above the diagram  $A \xrightarrow{f} C \xleftarrow{g} B$ . A graph  $P$  is a subgraph of the product  $\Pi$  above the diagram consisting of objects  $A$  and  $B$  and no morphisms.*

*Proof.* To show that  $P$  is a subgraph of  $\Pi$ , it suffices to show that there exists an injective morphism  $\psi : P \rightarrow \Pi$ . Indeed,  $\psi$  trivially maps a pullback graph to a subgraph of the product  $P$ :

- $\psi^{\text{vertex}} : V_P \rightarrow V_\Pi$ ,  $v_P = (v_A, v_B) \mapsto (v_A, v_B)$ ,
- $\psi^{\text{edge}} : E_P \rightarrow E_\Pi$ ,  $e_P = (e_A, e_B) \mapsto (e_A, e_B)$ ,
- $\psi^{\text{label}} : \Lambda_P \rightarrow \Lambda_\Pi$ ,  $\lambda_P = (\lambda_A, \lambda_B) \mapsto (\lambda_A, \lambda_B)$ ,

We conclude that graph  $P$  is a subgraph of graph  $\Pi$ . □

## 2.2.5 Pushout

**Definition 2.2.6.** Let  $A = (V_A, E_A, \Lambda_A, \sigma_A : E_A \rightarrow V_A, \tau_A : E_A \rightarrow V_A, \lambda_A : E_A \rightarrow \Lambda_A)$ ,  $B = (V_B, E_B, \Lambda_B, \sigma_B : E_B \rightarrow V_B, \tau_B : E_B \rightarrow V_B, \lambda_B : E_B \rightarrow \Lambda_B)$ , and  $C = (V_C, E_C, \Lambda_C, \sigma_C : E_C \rightarrow V_C, \tau_C : E_C \rightarrow V_C, \lambda_C : E_C \rightarrow \Lambda_C)$  be two directed labeled multigraphs,  $s : C \rightarrow A$  and  $t : C \rightarrow B$  be two graph morphisms. The *pushout* on the diagram  $A \xleftarrow{s} C \xrightarrow{t} B$  is the directed labeled multigraph  $H = (V_H, E_H, \Lambda_H, \sigma_H : E_H \rightarrow V_H, \tau_H : E_H \rightarrow V_H, \lambda_H : E_H \rightarrow \Lambda_H)$ , where

- $V_H = V_A \sqcup V_B / \equiv$ , where  $\equiv$  is the equivalence relation generated by the relation  $R$  on  $V_A \sqcup V_B$  where,

$$v_A R v_B \Leftrightarrow \exists v_C \text{ such that } s^{\text{vertex}}(v_C) = v_A \text{ and } t^{\text{vertex}}(v_C) = v_B,$$

- $E_H = E_A \sqcup E_B / \equiv$ , where  $\equiv$  is the equivalence relation generated by the relation  $R$  on  $E_A \sqcup E_B$  where,

$$e_A R e_B \Leftrightarrow \exists v_C \text{ such that } s^{\text{edge}}(e_C) = e_A \text{ and } t^{\text{edge}}(e_C) = e_B,$$

- $\Lambda_H = \Lambda_A \sqcup \Lambda_B / \equiv$ , where  $\equiv$  is the equivalence relation generated by the relation  $R$  on  $\Lambda_A \sqcup \Lambda_B$  where,

$$\gamma R \delta \Leftrightarrow \exists \phi \text{ such that } s^{\text{label}}(\phi) = \gamma \text{ \& } t^{\text{label}}(\phi) = \delta,$$

- $\sigma_H : E_H \rightarrow V_H, [e] \mapsto [v]$ , where  $v = \begin{cases} \sigma_A(e), & \text{if } e \in E_A, \\ \sigma_B(e), & \text{if } e \in E_B, \end{cases}$

- $\tau_H : E_H \rightarrow V_H, [e] \mapsto [w]$ , where  $w = \begin{cases} \tau_A(e), & \text{if } e \in E_A, \\ \tau_B(e), & \text{if } e \in E_B, \end{cases}$

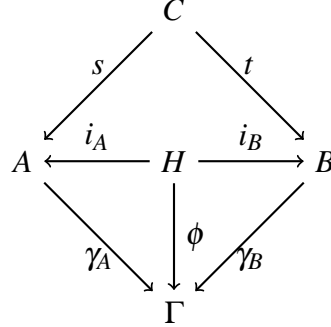
- $\lambda_H : E_H \rightarrow V_H, [e] \mapsto [\lambda]$ , where  $\lambda = \begin{cases} \lambda_A(e), & \text{if } e \in E_A, \\ \lambda_B(e), & \text{if } e \in E_B. \end{cases}$

together with two morphisms  $i_K : K \rightarrow H$ , where  $K \in \{A, B\}$  by

- $i_K^{vertex} : V_K \rightarrow V_H, v \mapsto [v]$ ,
  - $i_K^{label} : \Lambda_K \rightarrow \Lambda_H, \gamma \mapsto [\gamma]$ ,
  - $i_K^{edge} : E_K \rightarrow E_H, e \mapsto [e]$ ,
- more precisely,  $\sigma_K(e) \mapsto [\sigma_K(e)]$ ,  $\tau_K(e) \mapsto [\tau_K(e)]$ , and  $\lambda_K(e) \mapsto [\lambda_K(e)]$ .

**Proposition 6.** *The pushout  $H$  is the initial object over the diagram  $A \xleftarrow{s} C \xrightarrow{t} B$ .*

*Proof.* To show that the pushout  $H$  is initial over the diagram  $A \xleftarrow{s} C \xrightarrow{t} B$ , we must show that for any graph  $\Gamma$  together with morphisms  $\gamma_A : A \rightarrow \Gamma$  and  $\gamma_B : B \rightarrow \Gamma$  such that the outer diagram commutes, there exists the unique morphism  $\phi : H \rightarrow \Gamma$  making the inner diagram commute.



First, we will show that morphisms  $i_A$  and  $i_B$  are well-defined. For every edge  $e_K \in E_K$  where  $K \in \{A, B\}$ ,

$$\begin{aligned} \sigma_H(i_K^{edge}(e_K)) &= \sigma_H([e_K]) \\ &= [\sigma_K(e_K)] \\ &= i_K^{vertex}(\sigma_K(e_K)), \end{aligned}$$

$$\begin{aligned}
\tau_H(i_K^{edge}(e_K)) &= \tau_H([e_K]) \\
&= [\tau_K(e_K)] \\
&= i_K^{vertex}(\tau_K(e_K)),
\end{aligned}$$

$$\begin{aligned}
\lambda_H(i_K^{edge}(e_K)) &= \lambda_H([e_K]) \\
&= [\lambda_K(e_K)] \\
&= i_K^{label}(\lambda_K(e_K)).
\end{aligned}$$

The pushout diagram is commutative. For every vertex  $v_C \in V_C$ ,

$$\begin{aligned}
i_A^{vertex}(s^{vertex}(v_C)) &= [s^{vertex}(v_C)] \\
&= [t^{vertex}(v_C)] \\
&= i_B^{vertex}(s^{vertex}(v_C)).
\end{aligned}$$

Similarly, for every edge  $e_C \in E_C$ ,

$$\begin{aligned}
i_A^{edge}(s^{edge}(e_C)) &= [s^{edge}(e_C)] \\
&= [t^{edge}(e_C)] \\
&= i_B^{edge}(s^{edge}(e_C)),
\end{aligned}$$

and for every label  $\lambda_C \in \Lambda_C$ ,

$$\begin{aligned}
i_A^{label}(s^{label}(\lambda_C)) &= [s^{label}(\lambda_C)] \\
&= [t^{label}(\lambda_C)] \\
&= i_B^{label}(s^{label}(\lambda_C)).
\end{aligned}$$

Then, there is unique morphism  $\phi : H \rightarrow \Gamma$  such that the diagram commutes. The morphism  $\phi$  is defined as follows:

$$\begin{aligned}
\phi^{vertex} : V_H \rightarrow V_\Gamma, v_h = [v] &\mapsto \begin{cases} \gamma_A^{vertex}(v), & \text{if } v \in V_A \\ \gamma_B^{vertex}(v), & \text{if } v \in V_B \end{cases} \\
\phi^{label} : \Lambda_H \rightarrow \Lambda_\Gamma, \lambda_h = [\lambda] &\mapsto \begin{cases} \gamma_A^{label}(\lambda), & \text{if } \lambda \in \Lambda_A \\ \gamma_B^{label}(\lambda), & \text{if } \lambda \in \Lambda_B \end{cases} \\
\phi^{edge} : E_H \rightarrow E_\Gamma, e_h = [e] &\mapsto \begin{cases} \gamma_A^{edge}(e), & \text{if } e \in E_A \\ \gamma_B^{edge}(e), & \text{if } e \in E_B \end{cases}
\end{aligned}$$

The morphism  $\phi^{vertex}$  is well-defined.

(1) For all vertices  $v_A \in V_A$  and  $v_B \in V_B$  such that  $\exists v_C \in V_C$  with  $s^{vertex}(v_C) = v_A$  and  $t^{vertex}(v_C) = v_B$ . Then,

$$\begin{aligned}\phi^{vertex}([v_A]) &= \gamma_A(v_A) \\ &= \gamma_A(s^{vertex}(v_C)) \\ &= \gamma_B(t^{vertex}(v_C)) \\ &= \gamma_B(v_B) \\ &= \phi^{vertex}([v_B]).\end{aligned}$$

(2) For all vertices  $v_A, v'_A \in V_A$  and  $v_B \in V_B$  such that  $\exists v_C, v'_C \in V_C$  with  $s^{vertex}(v_C) = v_A, s^{vertex}(v'_C) = v'_A$  and  $t^{vertex}(v_C) = t^{vertex}(v'_C) = v_B$ . Then,

$$\begin{aligned}\phi^{vertex}([v_A]) &= \gamma_A^{vertex}(v_A) \\ &= \gamma_A^{vertex}(s^{vertex}(v_C)) \\ &= \gamma_B^{vertex}(t^{vertex}(v_C)) \\ &= \gamma_B^{vertex}(t^{vertex}(v'_C)) \\ &= \gamma_A^{vertex}(s^{vertex}(v'_C)) \\ &= \gamma_A^{vertex}(v'_A) \\ &= \phi^{vertex}([v'_A]).\end{aligned}$$

(3) For all vertices  $v_A \in V_A$  and  $v_B, v'_B \in V_B$  such that  $\exists v_C, v'_C \in V_C$  with  $s^{vertex}(v_C) = s^{vertex}(v'_C) = v_A, t^{vertex}(v_C) = v_B$  and  $t^{vertex}(v'_C) = v'_B$ . Then,

$$\begin{aligned}\phi^{vertex}([v_B]) &= \gamma_B^{vertex}(v_B) \\ &= \gamma_B^{vertex}(t^{vertex}(v_C)) \\ &= \gamma_A^{vertex}(s^{vertex}(v_C)) \\ &= \gamma_A^{vertex}(s^{vertex}(v'_C)) \\ &= \gamma_B^{vertex}(t^{vertex}(v'_C)) \\ &= \gamma_A^{vertex}(v'_B) \\ &= \phi^{vertex}([v'_B]).\end{aligned}$$

Therefore  $\phi^{vertex}$  is well-defined. Similarly,  $\phi^{label}$  is well-defined.

(1) For all labels  $\lambda_A \in \Lambda_A$  and  $\lambda_B \in \Lambda_B$  such that  $\exists \lambda_C \in \Lambda_C$  with  $s^{label}(\lambda_C) = \lambda_A$  and  $t^{label}(\lambda_C) =$

$\lambda_B$ . Then,

$$\begin{aligned}
\phi^{label}([\lambda_A]) &= \gamma_A(\lambda_A) \\
&= \gamma_A(s^{label}(\lambda_C)) \\
&= \gamma_B(t^{label}(\lambda_C)) \\
&= \gamma_B(\lambda_B) \\
&= \phi^{label}([\lambda_B]).
\end{aligned}$$

(2) For all vertices  $\lambda_A, \lambda'_A \in \Lambda_A$  and  $\lambda_B \in \Lambda_B$  such that  $\exists \lambda_C, \lambda'_C \in \Lambda_C$  with  $s^{label}(\lambda_C) = \lambda_A, s^{label}(\lambda'_C) = \lambda'_A$  and  $t^{label}(\lambda_C) = t^{label}(\lambda'_C) = \lambda_B$ . Then,

$$\begin{aligned}
\phi^{label}([\lambda_A]) &= \gamma_A^{label}(\lambda_A) \\
&= \gamma_A^{label}(s^{label}(\lambda_C)) \\
&= \gamma_B^{label}(t^{label}(\lambda_C)) \\
&= \gamma_B^{label}(t^{label}(\lambda'_C)) \\
&= \gamma_A^{label}(s^{label}(\lambda'_C)) \\
&= \gamma_A^{label}(\lambda'_A) \\
&= \phi^{label}([\lambda'_A]).
\end{aligned}$$

(3) For all vertices  $\lambda_A \in \Lambda_A$  and  $\lambda_B, \lambda'_B \in \Lambda_B$  such that  $\exists \lambda_C, \lambda'_C \in \Lambda_C$  with  $s^{label}(\lambda_C) = s^{label}(\lambda'_C) = \lambda_A$ ,  $t^{label}(\lambda_C) = \lambda_B$  and  $t^{label}(\lambda'_C) = \lambda'_B$ . Then,

$$\begin{aligned}
\phi^{label}([\lambda_B]) &= \gamma_B^{label}(\lambda_B) \\
&= \gamma_B^{label}(t^{label}(\lambda_C)) \\
&= \gamma_A^{label}(s^{label}(\lambda_C)) \\
&= \gamma_A^{label}(s^{label}(\lambda'_C)) \\
&= \gamma_B^{label}(t^{label}(\lambda'_C)) \\
&= \gamma_B^{label}(\lambda'_B) \\
&= \phi^{label}([\lambda'_B]).
\end{aligned}$$

Consider some edge  $[e] \in E_H$ . Since the diagram commutes on labels and vertices, then

$$\phi^{label}([\lambda(e)]) \in \Lambda_\Gamma \wedge \phi^{vertex}([\sigma(e)]), \phi^{vertex}([\tau(e)]) \in V_\Gamma \implies \phi^{edge}([e]) \in E_\Gamma.$$

Therefore, the diagram commutes on edges as well. We conclude that the morphism  $\phi$  is well-defined. It is clear that the diagram 2.2.3 commutes. Let us show that  $\phi$  is unique. Suppose, there is another morphism  $\phi_2$ . For given vertex  $v_H \in V_H$  there exist a representative of equivalence class  $[v]$ ,  $v \in V_A \sqcup V_B$ ,  $v_H \in [v]$ . Then,

$$\begin{aligned}\phi_2^{\text{vertex}}([v]) &= \begin{cases} \phi_2^{\text{vertex}}(i_A^{\text{vertex}}(v)), & \text{if } v \in V_A \\ \phi_2^{\text{vertex}}(i_B^{\text{vertex}}(v)), & \text{if } v \in V_B \end{cases} \\ &= \begin{cases} \gamma_A^{\text{vertex}}(v), & \text{if } v \in V_A \\ \gamma_B^{\text{vertex}}(v), & \text{if } v \in V_B \end{cases} \\ &= \phi^{\text{vertex}}([v])\end{aligned}$$

For given label  $\lambda_H \in \Lambda_H$  there exist a representative of equivalence class  $[\lambda]$ ,  $\lambda \in \Lambda_A \sqcup \Lambda_B$ ,  $\lambda_H \in [\lambda]$ . Then,

$$\begin{aligned}\phi_2^{\text{label}}([\lambda]) &= \begin{cases} \phi_2^{\text{label}}(i_A^{\text{label}}(\lambda)), & \text{if } \lambda \in \Lambda_A \\ \phi_2^{\text{label}}(i_B^{\text{label}}(\lambda)), & \text{if } \lambda \in \Lambda_B \end{cases} \\ &= \begin{cases} \gamma_A^{\text{label}}(\lambda), & \text{if } \lambda \in \Lambda_A \\ \gamma_B^{\text{label}}(\lambda), & \text{if } \lambda \in \Lambda_B \end{cases} \\ &= \phi^{\text{label}}([\lambda])\end{aligned}$$

Finally, following that  $\phi_2^{\text{vertex}} = \phi^{\text{vertex}}$  and  $\phi_2^{\text{label}} = \phi^{\text{label}}$ , for any edge  $[e] \in E_H$ ,

$$\phi_2^{\text{edge}}([e]) = \phi^{\text{edge}}([e])$$

We conclude that  $\phi$  is unique and that graph  $H$  together with morphisms  $i_A$  and  $i_B$  is a pushout on the diagram  $A \xleftarrow{s} C \xrightarrow{t} B$ .  $\square$

**Lemma 2.2.0.3.** *Let  $H$  be a pushout on a diagram  $A \xleftarrow{s} C \xrightarrow{t} B$ . The pushout graph  $H$  is isomorphic to the coproduct  $\Delta$  on the diagram consisting of objects  $A$  and  $B$  and no morphisms, if the graph  $C$  is an initial object in the **DLMG** category.*

*Proof.* Since  $C$  is the empty graph (initial object), the following holds:

(1) For all  $v_A \in V_A$  and  $v_B \in V_B$ :

$$\nexists v_C \in V_C = \emptyset \text{ such that } s^{\text{vertex}}(v_C) = v_A \text{ and } t^{\text{vertex}}(v_C) = v_B \implies [v_A] \neq [v_B],$$

(2) For all  $v_A, v'_A \in V_A$ :

$$\nexists v_B \in V_B \text{ such that } [v_A] = [v_B] = [v'_A] \implies [v_A] \neq [v'_A],$$

(3) For all  $v_B, v'_B \in V_B$ :

$$\nexists v_A \in V_A \text{ such that } [v_B] = [v_A] = [v'_B] \implies [v_B] \neq [v'_B].$$

Therefore, for all  $v \in V_A \sqcup V_B$ , we have  $|[v]| = 1$ . Similarly, for all  $e \in E_A \sqcup E_B$ ,  $|[e]| = 1$ , and for all  $\lambda \in \Lambda_A \sqcup \Lambda_B$ ,  $|\lambda| = 1$ .

Thus, the isomorphism  $\phi : H \rightarrow \Delta$  is defined as follows:

$$\begin{aligned} \phi^{\text{vertex}} : V_H &\rightarrow V_\Delta, [v] \mapsto \begin{cases} (v, a), & \text{if } v \in V_A, \\ (v, b), & \text{if } v \in V_B, \end{cases} \\ \phi^{\text{edge}} : E_H &\rightarrow E_\Delta, [e] \mapsto \begin{cases} (e, a), & \text{if } e \in E_A, \\ (e, b), & \text{if } e \in E_B, \end{cases} \\ \phi^{\text{label}} : \Lambda_H &\rightarrow \Lambda_\Delta, [\lambda] \mapsto \begin{cases} (\lambda, a), & \text{if } \lambda \in \Lambda_A, \\ (\lambda, b), & \text{if } \lambda \in \Lambda_B. \end{cases} \end{aligned}$$

We conclude that  $H$  and  $\Delta$  are isomorphic. □

**Implementation of universal constructions in DLMG category.** We developed algorithms for implementing universal constructions with the C# programming language on the .NET platform. Information on the project architecture and pseudocode for universal constructions implementation can be accessed in Appendix A. Our implementation of universal constructions can be accessed at [categories repository](https://github.com/mzheplin/Categories)<sup>1</sup>.

---

<sup>1</sup><https://github.com/mzheplin/Categories>

## Chapter 3

# Representation of User Interface as a Pullback

This chapter introduces a formal representation of user interfaces with graph structures. By leveraging a category of *directed labeled multigraphs*, the framework enables automated analysis and identification of potential weaknesses within the interface. Each of the graphs presented can be analyzed individually, highlighting specific aspects of the interface’s design and functionality.

### 3.1 User Interface as Universal Construction

We examine user interfaces from different perspectives and integrate them using category theory. UIs can be represented as pullback universal constructions, providing a framework for structuring relationships between states, actions, and affordances. Each interface is associated with the system’s states, encoded according to the analysis requirements. The interface aims to implement some functionality that can be represented as a state-transition diagram, and communicate this functionality to the user. It provides users with possible actions, affordances, allowing them to manipulate system states. While interacting systems consist of states and transitions, they also include an interactive component that bridges the user and the system. Pullbacks allow us to combine formal and interactive components of user interfaces or other interactive systems. Pullbacks in modeling user interfaces focus on the conditional relations of system components being a suitable choice of universal construction. The pullback’s ability to integrate multiple strategies into a single interface aligns with the objective of this work. In contrast, pushouts are dual to pullbacks and are relevant to combining systems into a larger whole, distinguishing the

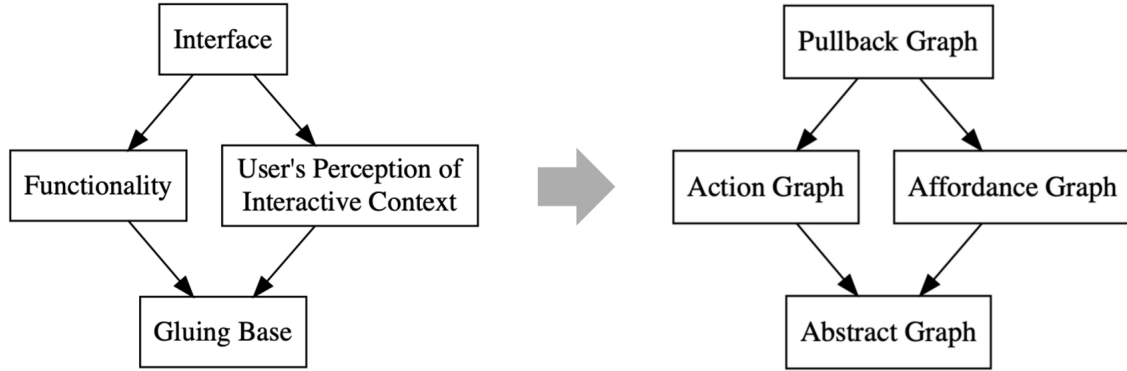


Figure 3.1: High level diagram of user interfaces pullback representation with the categorical framework.

elements in common. This distinction makes pullbacks more applicable for representing user interactions and interfaces, emphasizing possibility of different combined representations of the interactive system instead of integrating different systems into one.

User interfaces and other interactive systems can be represented by directed labeled multi-graphs. In this thesis, we will model graphs of the technical and interactive components of the interfaces, determining their conditional connections and obtaining a pullback universal construction in the category of directed labeled multigraphs. The reader will see the following diagram:

$$\begin{array}{ccccc}
 & & \pi_A & & \pi_D \\
 & & \longleftarrow & & \longrightarrow \\
 A & & P & & D \\
 & \searrow f & & & \swarrow g \\
 & & M & & 
 \end{array}$$

where  $A$  represents the action graph,  $D$  the affordance graph,  $M$  the abstract graph, and  $P$  the pullback graph. The morphism  $f$  connects the action graph to the abstract graph, while  $g$  connects the affordance graph to the abstract graph. The projections  $\pi_A$  and  $\pi_D$  are from the pullback to the action and affordance graphs, respectively. We construct our graphs entirely from the user's viewpoint. Below, we describe four graphs: three as part of a diagram and one as a categorical pullback derived from the diagram. Their detailed descriptions and roles are provided in the following sections.

### 3.1.1 Action Graph $A$

Let  $A = (V_A, E_A, \Lambda_A, \sigma_A : E_A \rightarrow V_A, \tau_A : E_A \rightarrow V_A, \lambda_A : E_A \rightarrow \Lambda_A)$  denote an action graph. The action graph  $A$  formally represents transitions between states in the interactive system (interface) among a set of possible states. The state of an object is determined by its essential characteristics. Arrows in the graph represent transitions between states and contain details about state changes or represent the high level conceptual reasoning behind them.

#### Vertices of Action Graph

**Definition 3.1.1.** Let  $s = [s_1, \dots, s_n]$  represent the state of the system  $S$  as a vector of features. There is a corresponding *vertex*  $s \in V_A$  in the action graph  $A$  denoted as  $s$  to match the state name.

**Example 3.1.1.** Consider the example of a book reader. Its purpose is to allow the user to read the text and navigate through it. Therefore, we consider the state of the text (or more abstractly, the amount of text) that fits on the screen. So, if there is a need to increase the text itself by reducing its amount on the page, the user can be given the opportunity to divide it into smaller parts. Each smaller text part is also a state of the interactive system. Alternatively, we can create multiple versions of this division, such as dividing the text into three parts, then six, and finally twelve. Together, all these pieces of text represent different states of the system. For representation of the states in the action graph, refer to the Figure 3.2.

#### Edges of Action Graph

**Definition 3.1.2.** Let  $s_1$  and  $s_2$  be states of the system  $S$ . There is an edge  $e_A \in E_A$  in the action graph  $A$  such that  $\sigma_A(e) = s_1$  and  $\tau_A(e) = s_2$  if a transition exists from state  $s_1$  to state  $s_2$ .

**Definition 3.1.3.** Let  $e_A \in E_A$  be an edge in the action graph  $A$ . The label  $\lambda(e_A)$  on edge  $e_A$  names the action performed to transition from state  $s_1$  to state  $s_2$ . This action can also be applied to other states.

Since each state can be represented as a vector of features, the label can formally indicate which specific features have changed and the corresponding new values. Alternatively, the label may denote the higher-level action name of the state transition.

**Example 3.1.2.** Consider the previous example 3.1.1 of a large text is segmented into multiple parts, each arranged sequentially and assigned a serial number. We restrict navigation to adjacent

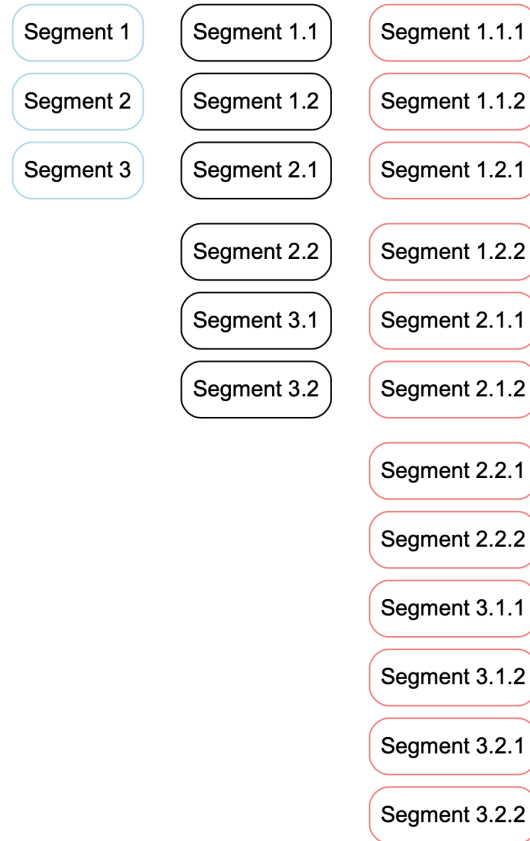


Figure 3.2: Different states of the text reader. A segment refers to a specific portion of some content displayed on the page. Each column represents some division of the same text into segments.

parts, which implies that there exists an edge in the graph between states  $s_1$  and  $s_2$ , and another edge between  $s_2$  and  $s_1$ . On the one hand, these transitions can be labeled as  $\{x\} \rightarrow \{x+1\}$  and  $\{x\} \rightarrow \{x-1\}$ , respectively, where  $x-1, x, x+1$  are serial numbers of the text parts. On the other hand, the edges that lead to the next sections can be labeled as “next,” while those leading to previous sections can be labeled as “previous.” The latter approach offers a more high-level, conceptual description, which may be more comprehensible to the human mind.

Suppose, we want to include different options for dividing the text into varying numbers of sections. Each division is represented as a separate set of nodes in the graph, and these sets do not overlap. Now, we want the ability to transition between these divisions, such as moving from

a larger text segment to one that is half its size, representing only the first part, and conversely, from a smaller segment to a larger one, where the first half remains consistent with the original state. In this scenario, there are now edges between states  $x$  and  $y$ , where  $x$  belongs to one division and  $y$  to another. These transitions can be labeled  $x \rightarrow y$  and  $y \rightarrow x$ , or more abstractly as “zoom in” and “zoom out”. For visualization, see Figure 3.3.

**Perceived Action Graph.** As mentioned, not all characteristics must be included in the action graph. For instance, when dealing with a web page, there is no need to include details like the state of the computer, other open applications, or the device’s battery charge because these factors are not relevant to the page as a user interface. This leads to the following proposition.

**Proposition 7.** *Let  $A$  be an action graph. A system’s state  $s$  is included in the graph  $A$  if and only if the user can perceive and distinguish it from other states.*

Analysis of the interface makes sense only when viewed through the eyes of the target user. If the user sees two separate states as one, we must not treat them as different and create two separate nodes. The next proposition arises:

**Proposition 8.** *Let  $A$  be an action graph. If a system’s states  $s_1, s_2, \dots, s_n$  for  $n \in \mathbb{N}$  are perceived by a user as one. Then there exist a graph  $A'$  and a morphism  $\phi : A \rightarrow A'$  such that*

$$\phi(s_1) = \phi(s_2) \cdots = \phi(s_n) = [s] \in V_{A'}$$

$$\phi(s') = s', s' \notin s_1, \dots, s_n$$

$$\phi(e) = e, e \in E_A$$

$$\phi(\lambda) = \lambda, \lambda \in \Lambda_A$$

where  $[s]$  is an equivalence class of vertices  $s_1, s_2, \dots, s_n$ .

Informally speaking, we glue indistinguishable vertices into one, and leave the remaining part of a graph untouched. With this proposition, we want to emphasize that the focus is always on the user who interacts with the system, so details are included if they relate to the user’s level of perception and understanding.

**Example 3.1.3.** Consider an example of a tape recorder from Don Norman’s book, “The Design of Everyday Things.” The author illustrates a case of an unsuccessful design where perception of the user was not taken into account. The tape recorder can be in three states: a recorder without a tape (00), a recorder with a tape but not operating (10), and a recorder that is actively running

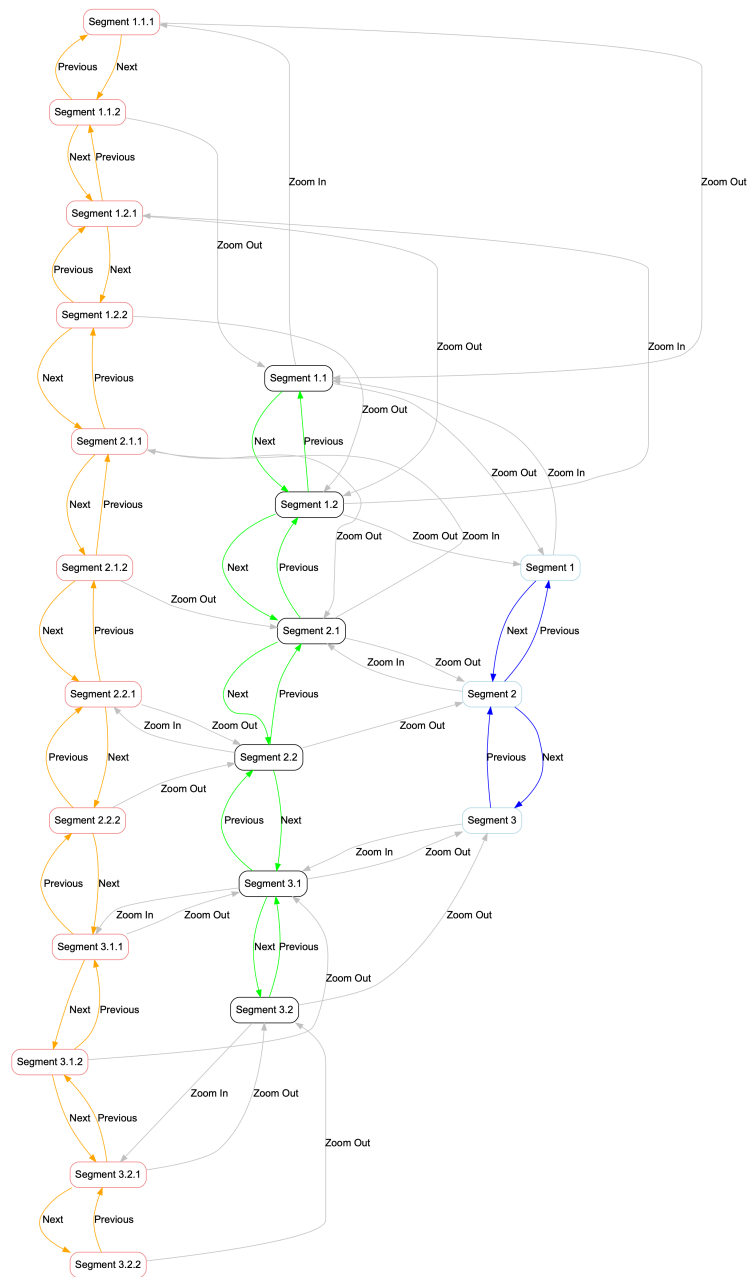


Figure 3.3: The action graph of the text reader interacting system.

(11). Transitions occur from state 00 to state 10, and from state 10 to state 11, and vice versa. Illustration an action graph is shown below.

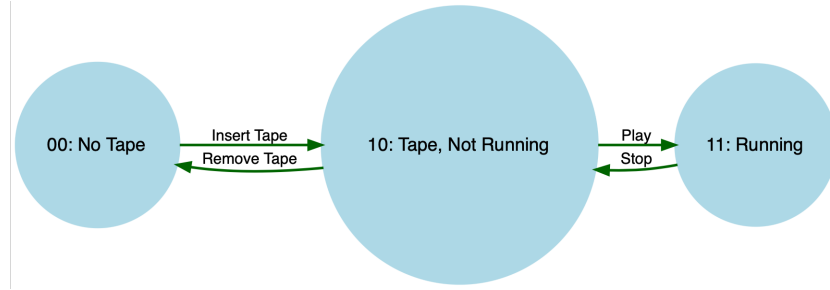


Figure 3.4: The action graph of the recorder.

Suppose, that the presence of a tape is not perceived, meaning that a user cannot ascertain whether the recorder contains a tape simply by observing it. In this scenario, the perceptible action graph is reduced to two states: running (1) and not running (0). Notably, there is a self-loop from state 0 to itself, labeled “Insert/Remove Tape,” since the recorder cannot operate without the tape.

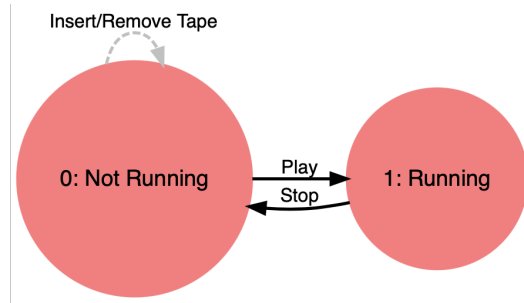


Figure 3.5: Perceived action graph of the recorder.

### 3.1.2 Affordance Graph $D$

Let  $D = (V_D, E_D, \Lambda_D, \sigma : E_D \rightarrow V_D, \tau : E_D \rightarrow V_D, \lambda_D : E_D \rightarrow \Lambda_D)$  denote an affordance graph. The affordance graph  $D$  represents the contexts offered to the user, as well as the transitions between them. By context we mean a unique set of affordances (possibilities of actions) that the user can observe when interacting with UI. Arrows must contain information about the method of

performing the action. It's important to point out that we generalize the concept of affordances by Norman, who makes distinction between real affordances, perceived affordances, and signifiers. In our affordances graph, we group all of these interactable components under the term “affordances.”

### Vertices of Affordance Graph

**Definition 3.1.4.** Let  $C$  be a set of affordances  $a_1, a_2, \dots, a_n$ , available to the user at the same time. We define  $C = \{a_1, a_2, \dots, a_n\}$  as the *context* in which the user interacts with the system. Each *unique context* corresponds to a vertex  $C$  in the affordance graph  $D$ .

**Example 3.1.4.** Without delving into the system's states or functionality, let us focus solely on its interactive aspect. Imagine a system presented to the user via a vertical page. The user either knows or is provided with a signifier to understand the available affordances, specifically the ability to swipe to the right or to the left. Also, user can perform pinch-in and pinch-out gestures. In this case, there is always only one context  $C$  consisting right-click  $r$ , left-click  $l$ , pinch-in  $in$ , pinch-out  $out$  affordances. Formally,  $c = \{r, l, in, out\}$ .

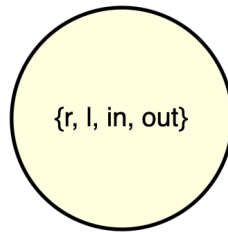


Figure 3.6: Simple context, consisting of four affordances, *in*, *out*, *r* and *l*.

**Example 3.1.5.** Let us revisit the previous example of a page where the user has the ability to perform actions such as swiping, zooming in, and zooming out. Now, consider scenarios where, in some cases, there are limitations on the user's ability to swipe in specific directions, or where the zooming functions are restricted. In our example, we include nine distinct contexts, as shown in the figure below. In one context, the user can access all four actions, meaning all four affordances are available. In a different context, one of the affordances may not be accessible, but the remaining three actions are still usable. However, it is important to include any affordance that is available in a given context, even if it does not impact the state (see the definition of vertex in the action graph 3.1.1).

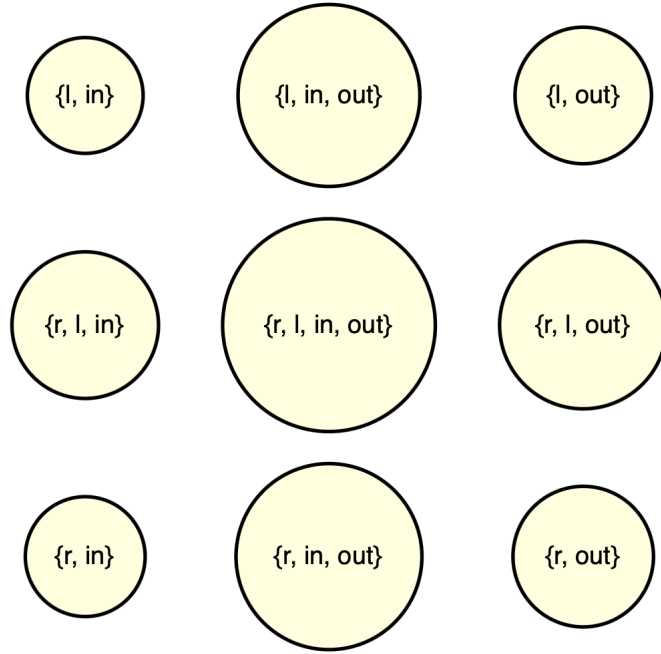


Figure 3.7: Nine unique contexts provided to the user by a book reader interface.

### Edges of Affordance Graph

**Definition 3.1.5.** Let  $a$  be an affordance represented in the context  $C$ ,  $a \in C$ . Then, there is *at least one* edge  $e_D$  in the affordance graph  $D$  such that  $\sigma(e_D) = C$  and  $\lambda(e_D) = a$ .

If a particular context, represented by a set of possible actions (affordances), includes an affordance  $a$ , then there is an edge originating from the corresponding vertex in the affordance graph  $D$  that represents the realization of this affordance. It may lead to a different context, or the same. It is important to note that more than one edge may exist in the graph, indicating the realization of an affordance. This realization can be the same or different.

**Example 3.1.6.** Let us return to the example 3.1.4. Now, we can complete our affordance graph with four edges (loops) that do not change a context. Always the same set of affordances is present. For visualization, see the figure 3.8.

**Definition 3.1.6.** Let  $\lambda(e_D) = a$  be a label of the edge  $e_D$ . The edge label  $\lambda(e_D)$  specifies the affordance being implemented, and a method of implementation.

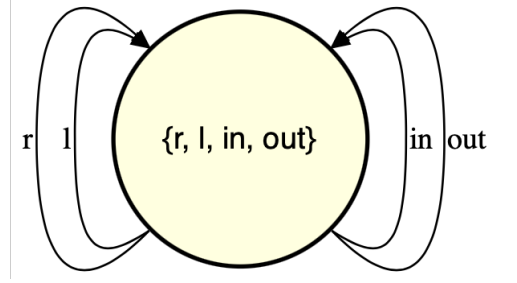


Figure 3.8: The affordance graph with one context, and four loops.

The edge label can show how the affordance  $a$  is triggered. Users may not know about these methods. For example, a single button (an example of an affordance) might perform different functions depending on whether it is clicked once or double-clicked. It is important to understand if users can recognize these different interactions. More importantly, if users are capable of performing these actions. In different modes of the same context, a single affordance with the same activation method can perform different functions. Therefore, it may be beneficial to differentiate the modes in which the affordance is triggered. More details on modes are discussed in 3.1.3. To analyze the interface, we must make different assumptions on the level of the user knowledge of the interface and perceived abilities. With labels we can reflect the way of perception and change them accordingly. Now that we know what an edge in an affordance graph represents, we can define when two contexts are the same.

**Definition 3.1.7.** Let  $c_1$  and  $c_2$  be two contexts. We say that  $c_1$  and  $c_2$  are *equivalent* if they represent the same set of affordances and have the same relations to other contexts, as determined by affordance triggers. More precisely,  $c_1 = c_2$  as sets, and there exists a homomorphism  $\phi : D \rightarrow D$  such that  $\phi^{\text{vertex}}(c_1) = c_2$  and  $\phi^{\text{vertex}}(c_2) = c_1$ , while all other elements of  $D$  are fixed under  $\phi$ .

**Example 3.1.7.** Let return to the example 3.1.5. A total of nine unique contexts are included, each represented by a distinct set of affordances. Each affordance is realized by an edge with a matching label originating from the respective vertex. It is important to note that some affordances are implemented more than once. For instance, the affordance  $l$  from the context  $\{l, r, in\}$  might either remain within the same context or transition to the context  $\{r, in\}$ . To illustrate, when swiping left, we can either swipe left again or reach a limit where further swipes are impossible. For visualization, see the figure 3.9. Notably, this affordance graph is non-deterministic, as there are multiple edges with identical labels originating from the same vertex. We will later demonstrate that this characteristic generally does not result in ambiguities or non-deterministic elements within the interface.

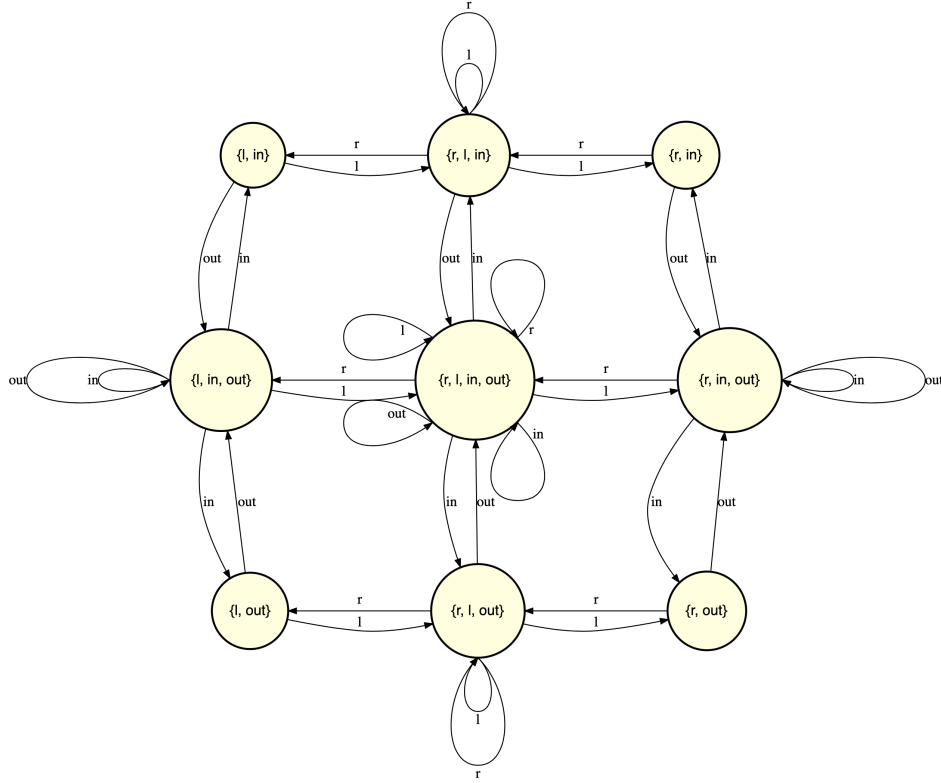


Figure 3.9: The affordance graph with nine contexts, and affordances realizations.

### 3.1.3 Modes

Earlier, we defined the concept of equivalent contexts. In the affordance graph, contexts are considered equal not only if they share the same set of affordances but also if they have identical relationships with other contexts 3.1.7. The affordance graph can have multiple vertices with the same affordances but different relationships with other contexts. In the next section, we will explore this aspect in consideration of interface ambiguities, introducing the concept of modes.

Modes frequently appear in design, especially with physical objects, where the number of available affordances is limited. Modes allow for more functionality within these constraints. In one mode, an affordance may perform one function, while in another mode, it performs a different one.

We distinguish between two types of modes: *explicit* and *implicit*. Explicit modes are distinctly identifiable within states, meaning the state itself includes an indicator of the mode. A



Figure 3.10: *Penultimate* iPad application that allows user to write in different modes.

mode is only considered explicit if there is a clear association with the state, making it part of that state. It remains unclear when an indicator is considered part of the state versus just part of the context. We assume that a mode belongs to the state if it clearly distinguishes the affordance function and uniquely identifies itself in the context of user profile, making it recognizable as a component of the state.

**Example 3.1.8.** A good example of modes application is in an iPad app that imitates writing on paper (see Figure 3.10). Traditionally, we need separate pens, pencils, or markers when interacting with real papers, but we can switch between these functions with a single electronic pen when working with an electronic device. The toolbar clearly displays options for selecting the pen's function. These modes are distinct and easily recognizable, making them part of the system's visible state.

Implicit modes involve identical sets of affordances that look the same but aim to have different functionality within the pullback. Usually, there is some state dependence that cannot be perceived.

**Definition 3.1.8 (Implicit Modes).** Consider the equivalence classes of states in the interacting system, denoted as  $[s_1], \dots, [s_n]$ , and the equivalence classes of contexts, denoted as  $[c_1], \dots, [c_n]$ , where for each  $i$ ,  $f^{\text{vertex}}(s_i) = g^{\text{vertex}}(c_i)$ . If there exists some state  $s_{i,j} \in [s_i]$  such that  $f^{\text{vertex}}(s_{i,j}) = g^{\text{vertex}}([c_k]) = g^{\text{vertex}}([c_r])$  for distinct  $k$  and  $r$ , then additional copies of the contexts  $[c_k]$  and  $[c_r]$  must be introduced. These added contexts will be referred to as *implicit modes*.

Modes can cause user frustration if poorly designed. However, while common in design, they do not inherently make a design unsuccessful. For instance, in a text editor's preview mode,

all editing tools are turned off to prevent content changes, with buttons grayed out. Similarly, a microwave oven might have different modes like defrost, heat, and bake, clearly indicated on the display, allowing users to select the appropriate mode. However, if there is no indication of modes, then pressing the start button of the microwave oven will lead to an ambiguous result that depends on the unrecognizable mode. A poorly designed mode cannot be clearly associated with any specific states.

In the proposed representation of interfaces, if an affordance graph cannot be constructed without implicit modes, the design may have potential ambiguities due to affordances' different functions. On the other hand, explicit modes, depending on their functionality, are not necessarily negative aspects of the design if they provide some additional information about the system and a user clearly recognizes the mode.

### 3.1.4 Abstract Graph $M$

While the action graph does not provide information about the interactional component but about the states and their transitions, the affordance graph contains information about how the user interacts with the interface, recognizable and performable actions, but not information about what the user interacts with, how actions change system states. The user interface contains information about states and an interactive component. Thus, the system's state represents a set of affordances (context) to the user. With a particular context, the user sees the system's state and can reason about possible actions. The implementation of an affordance performs some action on the states of the system (zero action is also an action!), and any action is performed by implementing some affordance. In terms of category of directed labeled multigraphs,

- For every vertex  $s \in V_A$  in the action graph, there exists at least one vertex  $c \in V_D$  in the affordance graph such that the state  $s$  provides a context  $c$ .
- For every vertex  $c \in V_D$  in the affordance graph, there exists at least one vertex  $s \in V_A$  in the action graph such that the context  $c$  allows interaction with the state  $s$ .
- For every edge  $e_A \in E_A$  in the action graph, there exists an edge  $e_D \in E_D$  in the affordance graph such that the function execution  $e_A$  is enabled by the realization of affordance  $e_D$ .
- For every edge  $e_D \in E_D$  in the affordance graph, there exists an edge  $e_A \in E_A$  in the action graph such that the affordance trigger  $e_D$  results in the state transition  $e_A$ .

Ideally, two graphs are independent up to a certain level; we can change a graphs to other ones and get the opportunity to manipulate a completely different system with the same affordances.

Or vice versa, the same system is manipulated with different affordances, depending on the user's needs. However, it happens that certain actions are context-dependent, or the outcome of specific actions depends on the state-context pair. In the next chapters, we will explore the issue of interface ambiguity and the corresponding characteristics within our framework.

In this section, we will combine previously described action and affordance graphs. An abstract graph acts as a foundation for gluing or indicates constraints set by morphisms from both the affordance and action graphs, which the resulting pullback graph must satisfy. This abstract graph represents a base of an interface. Abstract graph captures the relationships between the action and affordance graphs and outlines how they are combined in the pullback.

Let  $M = (V_M, E_M, \Lambda_M, \sigma : E_M \rightarrow V_M, \tau : E_M \rightarrow V_M, \lambda_M : E_M \rightarrow \Lambda_M)$  represent an abstract graph. We will use graph morphisms  $f$  and  $g$  (from the action and affordance graphs, respectively) to this abstract graph. The following holds for abstract graphs:

- Let  $c_1, \dots, c_r \in V_D$  be contexts, where  $|V_D| = m$ . Let  $s_1, \dots, s_k \in V_A$  be states, where  $|V_A| = n$ . If the states  $s_1, \dots, s_k$  are present in the contexts  $c_1, \dots, c_r$ , then there exists a vertex  $[v]$  in the abstract graph such that  $g(c_1) = \dots = g(c_r) = v$ , and  $f(s_1) = \dots = f(s_k) = [v]$ .
- Let  $\lambda_A^1, \dots, \lambda_A^l \in \Lambda_A$  be edge labels in the action graph, and let  $\lambda_D^1, \dots, \lambda_D^s \in \Lambda_D$  be edge labels in the affordance graph. If the transitions labeled  $\lambda_A^1, \dots, \lambda_A^l$  are realized through the affordances labeled  $\lambda_D^1, \dots, \lambda_D^s$ , and vice versa, then there exists a label  $[\lambda]$  in the abstract graph such that  $f(\lambda_A^1), \dots, f(\lambda_A^l) = \lambda$  and  $g(\lambda_D^1), \dots, g(\lambda_D^s) = [\lambda]$ .
- Let  $e_A^1, \dots, e_A^q \in E_A$  be edges of the action graph, and let  $e_D^1, \dots, e_D^w \in E_D$  be edges of the affordance graph. If  $f(\sigma(e_A^i)) = g(\sigma(e_D^j))$ ,  $f(\tau(e_A^i)) = g(\tau(e_D^j))$ , and  $f(\lambda(e_A^i)) = g(\lambda(e_D^j))$  for  $1 \leq i \leq q$  and  $1 \leq j \leq w$ , then there exists an edge  $[e]$  in the abstract graph such that  $f(e_A^i) = e$  and  $g(e_D^j) = [e]$ .

In general, we combine sets of vertices, labels, and edges into equivalence classes within the abstract graph, creating a more compact structure. Intuitively, each set of states belongs exclusively to a particular set of contexts, and each set of contexts corresponds only to those states. We then merge them using morphisms into a single vertex. Similarly, labels representing global function names and affordances are unified into a single label in the abstract graph if they are compatible. Edge mappings naturally follow from the mappings of vertices and labels. Notably, each vertex, label, or edge can only be homomorphically mapped to exactly one object.

**Example 3.1.9** ( Book Reader 1.1). In the section on action graph, we presented an example showing text-based states and transitions 3.1.2, while the affordance graph section illustrated a

context offering the user 4 potential actions 3.1.6. Let us combine these two graphs and present an abstract graph along with the corresponding morphisms. Since the affordance graph consists of only one vertex, each state in the action graph corresponds to a single context. Thus, the abstract graph ?? has a single vertex combining all the vertices of the action graph with the vertex from the affordance graph. The action graph contains the following transitions: next, previous, zoom in, and zoom out. The affordance graph represents four affordances - *l*, *r*, *in*, *out* - which are uniquely paired with these actions. Therefore, the abstract graph contains four loops and is isomorphic to the affordance graph up to relabeling.

| On Vertices                                         |                                        |
|-----------------------------------------------------|----------------------------------------|
| Source vertex $v$                                   | Target vertex $f^{vertex}(v)$          |
| <i>a text segment</i>                               | $\circ$                                |
| On Labels                                           |                                        |
| Source edge label $\lambda$                         | Target edge label $f^{label}(\lambda)$ |
| zoom in                                             | ++                                     |
| zoom out                                            | --                                     |
| previous                                            | +                                      |
| next                                                | -                                      |
| On Edges                                            |                                        |
| Mapping follows from vertex and edge label mappings |                                        |

Table 3.1: The homomorphism from the action graph to the abstract graph of a reader.

| On Vertices                                         |                                        |
|-----------------------------------------------------|----------------------------------------|
| Source vertex $v$                                   | Target vertex $g^{vertex}(v)$          |
| <i>l</i> , <i>r</i> , <i>in</i> , <i>out</i>        | $\circ$                                |
| On Labels                                           |                                        |
| Source edge label $\lambda$                         | Target edge label $g^{label}(\lambda)$ |
| <i>in</i>                                           | ++                                     |
| <i>out</i>                                          | --                                     |
| <i>l</i>                                            | +                                      |
| <i>r</i>                                            | -                                      |
| On Edges                                            |                                        |
| Mapping follows from vertex and edge label mappings |                                        |

Table 3.2: The homomorphism from the affordance graph to the abstract graph of a reader.

**Example 3.1.10** ( Book Reader 1.2). We previously presented a more detailed affordance graph 3.1.7, where, under specific conditions, certain actions become unavailable. Let us make an analysis in context of action graph 3.1.2 Any text segment starting with “Lorem ipsum” marks the beginning of the entire text, so we can link such states to a context that lacks the “left” affordance. If the full sentence is displayed, the zoom-out option becomes unavailable, as it has reached its limit. Conversely, with the smallest segment, zoom-in is unavailable. Any segment containing the end of the text will represent a context where affordance “right” is not present.

| On Vertices                                         |                                        |
|-----------------------------------------------------|----------------------------------------|
| Source vertex $v$                                   | Target vertex $g^{vertex}(v)$          |
| $\{l, r, in, out\}$                                 | $\{-, +, - -, ++\}$                    |
| $\{l, r, in\}$                                      | $\{-, +, - -\}$                        |
| $\{l, r, out\}$                                     | $\{-, +, ++\}$                         |
| $\{l, in, out\}$                                    | $\{-, - -, ++\}$                       |
| $\{r, in, out\}$                                    | $\{+, - -, ++\}$                       |
| $\{l, in\}$                                         | $\{-, - -\}$                           |
| $\{l, out\}$                                        | $\{-, ++\}$                            |
| $\{r, in\}$                                         | $\{+, - -\}$                           |
| $\{r, out\}$                                        | $\{+, ++\}$                            |
| On Labels                                           |                                        |
| Source edge label $\lambda$                         | Target edge label $g^{label}(\lambda)$ |
| in                                                  | - -                                    |
| out                                                 | ++                                     |
| l                                                   | -                                      |
| r                                                   | +                                      |
| On Edges                                            |                                        |
| Mapping follows from vertex and edge label mappings |                                        |

Table 3.3: The homomorphism from the affordance graph to the abstract graph of reader 1.2.

As in the previous example, the abstract graph is isomorphic (the same up to relabeling) to the affordance graph 3.1.7 since each state of the action graph is coupled only with a single context, and each action is uniquely coupled with some affordance. We will not provide an abstract graph here. Since the state names are long, we will use code names. Each state is categorized into distinct blocks—blue, green, and orange 3.1.2. Additionally, each state can signify either the start of the text, the end, or a segment in the middle. States will be encoded as  $x - y - n$ , where  $x \in \{b, g, o\}$  represents the color block (blue, green, or orange),  $y \in \{s, m, e\}$  indicates

| On Vertices                                         |                                        |
|-----------------------------------------------------|----------------------------------------|
| Source vertex $v$                                   | Target vertex $f^{vertex}(v)$          |
| $b - s$                                             | $\{+, --\}$                            |
| $b - e$                                             | $\{-, --\}$                            |
| $b - m - 1$                                         | $\{-, +, --\}$                         |
| $g - s$                                             | $\{+, --, ++\}$                        |
| $g - e$                                             | $\{-, --, ++\}$                        |
| $g - m - i, 1 \leq i \leq 4$                        | $\{-, +, --, ++\}$                     |
| $o - s$                                             | $\{+, ++\}$                            |
| $o - e$                                             | $\{-, ++\}$                            |
| $o - m - i, 1 \leq i \leq 10$                       | $\{-, +, ++\}$                         |
| On Labels                                           |                                        |
| Source edge label $\lambda$                         | Target edge label $f^{label}(\lambda)$ |
| zoom in                                             | ++                                     |
| zoom out                                            | -                                      |
| previous                                            | +                                      |
| next                                                | -                                      |
| On Edges                                            |                                        |
| Mapping follows from vertex and edge label mappings |                                        |

Table 3.4: The homomorphism from the action graph to the abstract graph, reader example 1.2.

the position in the text (start, middle, or end), and  $n$  is an optional number used to differentiate between different segments in the middle.

### 3.1.5 Pullback Graph $P$

Let  $P = (V_P, E_P, \Lambda_D, \sigma : E_P \rightarrow V_P, \tau : E_P \rightarrow V_P, \lambda_P : E_P \rightarrow \Lambda_P)$  together with morphisms  $\pi_A : P \rightarrow A$  and  $\pi_D : P \rightarrow D$  denote a pullback universal construction 2.2.5 on the diagram  $A \xrightarrow{f} M \xleftarrow{g} D$  in the category of directed labeled multigraphs.  $A$  is an action graph,  $D$  is an affordance graph,  $M$  is an abstract graph,  $f$  and  $g$  are graph homomorphisms. According to the definition of pullback, the calculated construction combines the action and affordance graphs, and their combination is determined by an abstract graph as a base.

**Proposition 9.** *A pullback  $P$  on the diagram  $A \xrightarrow{f} M \xleftarrow{g} D$  corresponds to a user interface.*

We propose combining action and affordance graphs into a single representation using a universal construction from category theory. This encoding approach enables a clear definition of the user

interface's purpose and its interactive elements. From this perspective, it becomes easier to analyze the interface in terms of accessibility, clarity, and potential ambiguity. It is important to note that this method does not account for the appearance or visual characteristics of the interface, but rather focuses on the user experience, assuming that the user can identify the interactive components and states of the object they are interacting with.

**Definition 3.1.9.** Let  $(s, c) \in V_P$  represent a vertex in the pullback graph  $P$ , where  $\pi_A(s, c) = s$  corresponds to a state of the user interface, and  $\pi_D(s, c) = c$  corresponds to a context of the user interface.

If a vertex  $(s, c)$  exists in the pullback  $P$ , it indicates that a state  $s$  represents a context  $c$  to the user. Clearly,  $s$  must exist in the action graph, and  $c$  must be a part of an affordance graph.

**Definition 3.1.10.** Let  $(e_A, e_D) \in E_P$  represent an edge in the pullback graph  $P$ , where  $\pi_A(e_A, e_D) = e_A$  corresponds to a state transition, and  $\pi_D(e_A, e_D) = e_D$  corresponds to an affordance realization.

The presence of an edges  $(e_A, e_D)$  in the graph  $P$  signifies that the transition  $\lambda(e_A)$  from one context-state  $\sigma(e_A, e_D)$  is linked to an affordance  $\lambda(e_D)$ , resulting in a new context-state  $\tau(e_A, e_D)$ . It also shows that transition from state  $\sigma(e_A)$  to state  $\tau(e_A)$  are feasible within the action graph, and that contexts  $\sigma(e_D)$  and  $\tau(e_D)$  are connected through the realization of the affordance  $\lambda(e_D)$ . Since the affordance graph illustrates user-interface interactions and the action graph outlines system transitions, displaying their connections is essential. Our objective is to model a user interface where each context represents at least one system state, and each state is represented by a context. Additionally, every transition between states is implemented through affordances, with each affordance facilitating at least one transition. The following definition arises.

**Definition 3.1.11.** A user interface (UI) represented as a pullback graph  $P$  on a diagram  $A \xrightarrow{f} M \xleftarrow{g} D$ , along with two morphisms  $\pi_A$  and  $\pi_D$ , is said to be **complete** if

- (a) For every  $v_A \in V_A$ , there exists  $v_P \in V_P$  such that  $\pi_A(v_P) = v_A$ .
- (b) For every  $v_D \in V_D$ , there exists  $v_P \in V_P$  such that  $\pi_D(v_P) = v_D$ .
- (c) For every  $e_A \in E_A$ , there exists  $e_P \in E_P$  such that  $\pi_A(e_P) = e_A$ .
- (d) For every  $e_D \in E_D$ , there exists  $e_P \in E_P$  such that  $\pi_D(e_P) = e_D$ .

Thus, the morphisms  $\pi_A$  and  $\pi_D$  must be surjective on both vertices and edges.

**Proposition 10.** A user interface (UI) represented as a pullback graph  $P$  over a diagram  $A \xrightarrow{f} M \xleftarrow{g} D$  with morphisms  $\pi_A$  and  $\pi_D$  is **complete** if and only if

- (1) For every  $v_A \in V_A$ , there exists  $v_D \in V_D$  such that  $f(v_A) = g(v_D)$ .
- (2) For every  $v_D \in V_D$ , there exists  $v_A \in V_A$  such that  $f(v_A) = g(v_D)$ .
- (3) For every  $e_A \in E_A$ , there exists  $e_D \in E_D$  such that  $f(e_A) = g(e_D)$ .
- (4) For every  $e_D \in E_D$ , there exists  $e_A \in E_A$  such that  $f(e_A) = g(e_D)$ .

*Proof.* We will demonstrate that the conditions stated are equivalent to those in the definition.

(a-b)  $\Leftarrow$  Consider any vertex  $v_D \in V_D$ . By (2), there exists a vertex  $v_A \in V_A$  such that  $f^{vertex}(v_A) = g^{vertex}(v_D)$ . According to the definition of a pullback, this leads to a vertex  $(v_A, v_D) \in V_P$ . Applying the projection  $\pi_D$  to this vertex yields  $v_D$ , satisfying condition (b) of completeness. Similarly, for any vertex  $v_A \in V_A$ , there exists  $v_D \in V_D$  such that  $f^{vertex}(v_A) = g^{vertex}(v_D)$ , and applying  $\pi_D$  gives  $v_D$ , satisfying condition (a).

(a-b)  $\Rightarrow$  Conversely, if there exists  $v_P = (v_A, v_D) \in V_P$  such that  $\pi_D(v_P) = v_D$ , then there must exist  $v_A \in V_A$  with  $f^{vertex}(v_A) = g^{vertex}(v_D)$ , according to the pullback definition. Likewise, if there exists  $v_P = (v_A, v_D) \in V_P$  such that  $\pi_A^{vertex}(v_P) = v_A$ , then there exists  $v_D \in V_D$  satisfying  $f^{vertex}(v_A) = g^{vertex}(v_D)$ .

(c-d)  $\Leftarrow$  Consider any edge  $e_A \in E_A$ . By (3), there exists an edge  $e_D \in E_D$  such that  $f^{edge}(e_A) = g^{edge}(e_D)$ . The pullback definition implies the presence of an edge  $(e_A, e_D) \in E_P$ . Applying  $\pi_A$  to this edge yields  $e_A$ , satisfying condition (c). Similarly, for any edge  $e_D \in E_D$ , by (4), there exists  $e_A \in E_A$  such that  $f^{edge}(e_A) = g^{edge}(e_D)$ , leading to  $e_D$  via  $\pi_D$ , thus satisfying condition (d).

(c-d)  $\Rightarrow$  Conversely, if there exists  $e_P = (e_A, e_D) \in E_P$  such that  $\pi_A^{edge}(e_P) = e_A$ , then there exists  $e_D \in E_D$  with  $f^{edge}(e_A) = g^{edge}(e_D)$ , as defined by the pullback. Likewise, if  $e_P = (e_A, e_D) \in E_P$  satisfies  $\pi_D^{edge}(e_P) = e_D$ , then there exists  $e_A \in E_A$  such that  $f^{edge}(e_A) = g^{edge}(e_D)$ .  $\square$

Next, we will focus solely on complete pullbacks. Now that the reader is familiar with all the graphs, we will consider the most effective choice of an abstract graph. This will be the minimal universal graph construction that identifies the relationship between the action and affordance graphs. By “minimal,” we mean a graph where, if any component is removed, the morphisms  $f$  and  $g$ , and thus the pullback, will no longer exist. Furthermore, adding any component will not change the pullback.

**Proposition 11.** *Let  $M$  be an abstract graph. A universal abstract graph is a pushout 2.2.6 on the diagram  $A \xleftarrow{\pi_A} P \xrightarrow{\pi_D} D$ , where  $A$  is an action graph,  $P$  is a complete pullback, and  $D$  is an affordance graph.*

*Proof.* Given the description of morphisms  $f$  and  $g$ , abstract graph aligns with a definition of pushout in 2.2.6. The role of pullback graph  $P$  is to define parts in graphs  $A$  and  $D$  to be glued with morphisms  $\pi_A$  and  $\pi_D$ . Pullback graph  $P$  containing vertices and edges implies that morphisms  $f$  and  $g$  map their left and right parts to the same equivalence classes agreeing with a process of constructing pushout given a diagram. Conversely, if  $f$  and  $g$  did not map objects to the same equivalence class, then they would not appear in the pullback. For formal proof, refer to the proposition 6.  $\square$

Given that an abstract graph provides the base for a pullback, considering pushout a universal basis is natural from a category theory perspective. The constructed pullback provides the relationships between all components that are equivalent in a specific manner, conversely being a good candidate for glueing base in pushout construction.

Using the implemented software, we compute the pullbacks for the previously discussed examples of a book reader.

**Example 3.1.11.** Returning to the example 3.1.9 of a text reader with a single context, where we explicitly defined the morphisms to an abstract graph. In the final step, we compute the pullback. The resulting pullback is illustrated in the figure 3.11. Notably, some affordances, while present, are not realized. For instance, the state-context pair “*Segment 2.1*, -  $\{r, l, in, out\}$ ” includes four affordances but only two outgoing edges, with none representing the left action  $l$ . Since this is the start of the text, the left action would not influence the state. To ensure the pullback is complete, by using loops, we can explicitly indicate that the available affordances do not impact the state in a specific context. However, this leads to an adjustment in the action graph, where we clearly indicate that certain actions do not affect some states.

**Example 3.1.12.** Example 3.1.10 provided us with morphisms to abstract graph, where affordance graph provided us with more possible contexts. The pullback is in the figure 3.12. The primary distinction from the previous example is that each context only includes those realized affordances that affect the state-context pair. In this case, there is no requirement to indicate the null effect of certain affordances, as they are simply absent. Various scenarios can also occur in real-world situations. If affordances are consistently accessible to an individual, this aligns with the first example. However, the second example becomes relevant if the design is arranged so that ineffective actions are unavailable.

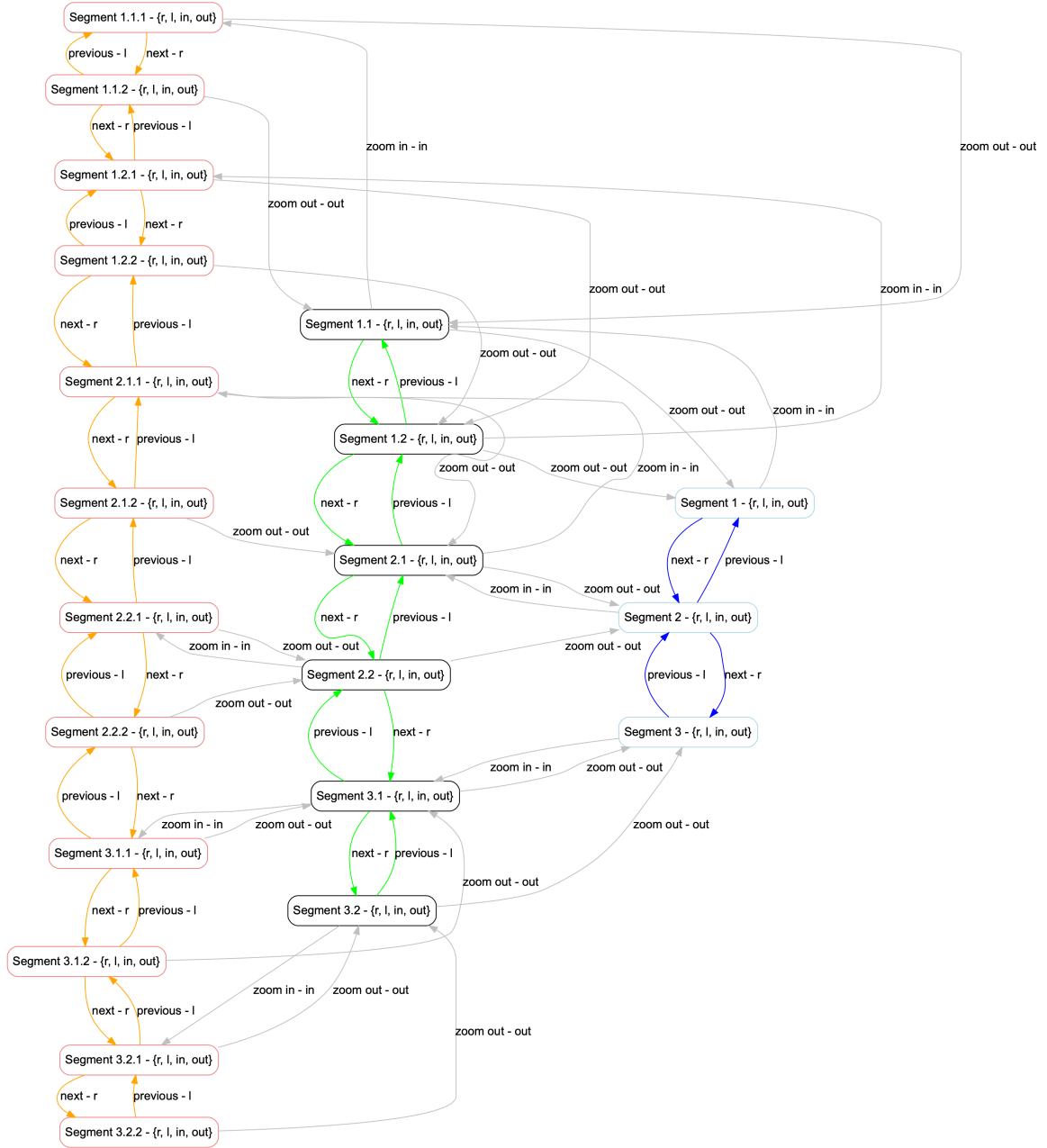


Figure 3.11: The pullback graph of a text reader 1.1.

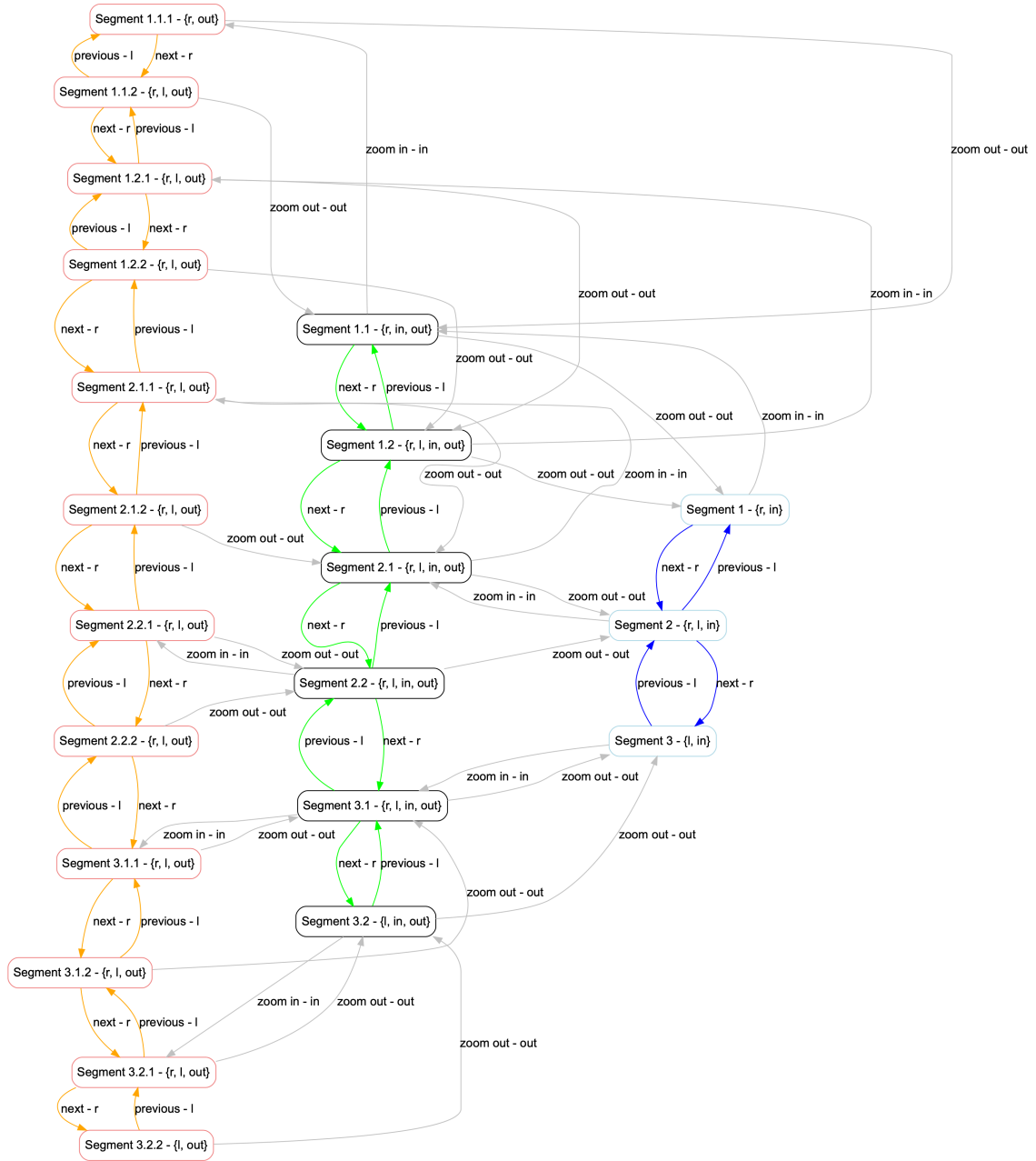


Figure 3.12: The pullback graph of a text reader 1.2.

### 3.1.6 Puzzles as Interacting Systems

Let us examine puzzles as user interfaces according to their logical and unambiguous structure, which lends itself to graphical representation. Notable examples to be described in this section include the famous 8-puzzle. While these puzzles were originally designed to foster and develop logical reasoning, achieving this objective requires that users are presented with a straightforward interface with affordances that enable intuitive interaction.

#### 3-Puzzle

**Action Graph.** Let us consider a full example of a smaller version of the 8-puzzle - the 3-puzzle. It consists of 3 tiles and one empty space. Let the state of the puzzle be represented as one of the

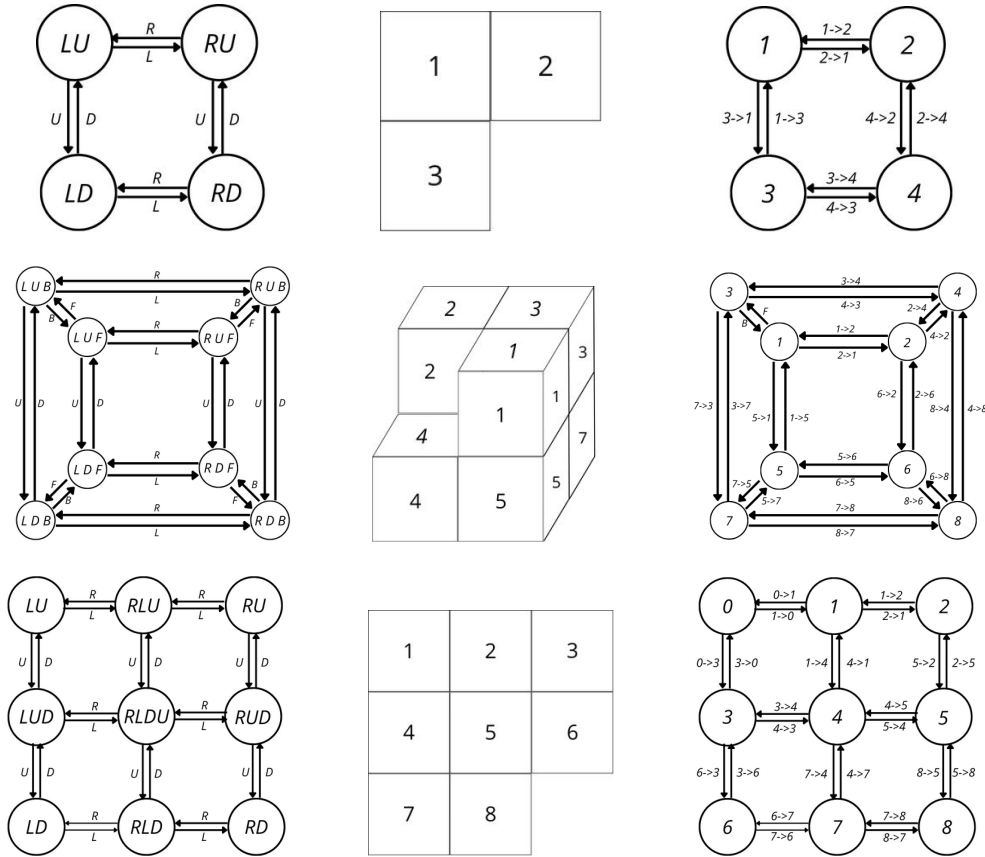


Figure 3.13: Visualization of a 3-puzzle (top), 7-puzzle (middle), 8-puzzle (bottom) in the center, together with their affordance graphs to the left and the abstract graphs to the right.

possible permutations of the numbers 1, 2, 3, and “\_,” which indicates a location of empty space. Let there be a transition from one state to another if there is an arrow of the form  $x \rightarrow y$  from one state, read as moving content from location  $x$  to location  $y$ . The “ $y$ ” defines where the “\_” is. As a result of the transition, location  $x$  becomes empty, and location “ $y$ ” contains what came from location  $x$ . Also note that from the first place, we cannot move to the fourth place and vice versa, and from the second, we cannot move to the third because they are arranged crosswise (see Figure 3.14).

**Affordance Graph.** The actions the user can perform depend on the location of the empty square. Each square contains two possibilities that encourage the user to move the squares. Therefore, there are transitions from each state to two others. Since there are four empty spaces that provide different sets of possible actions, we define four possible connected contexts (see Figure 3.13).

**Abstract Graph.** The abstract graph is constructed as follows. Each vertex exists by the location of the empty space. For example, the vertex with the number 1 represents states where the first place is empty. Also, since the first location is in the upper left corner, the vertex under this number represents the *LU* context. The labels of edges can be chosen arbitrarily, and we use labels from the action graph for simplicity (see Figure 3.13).

**Morphisms.** The morphisms are constructed as follows. A vertex from the action graph is mapped to the corresponding number of its empty space; the labels are mapped to the identical

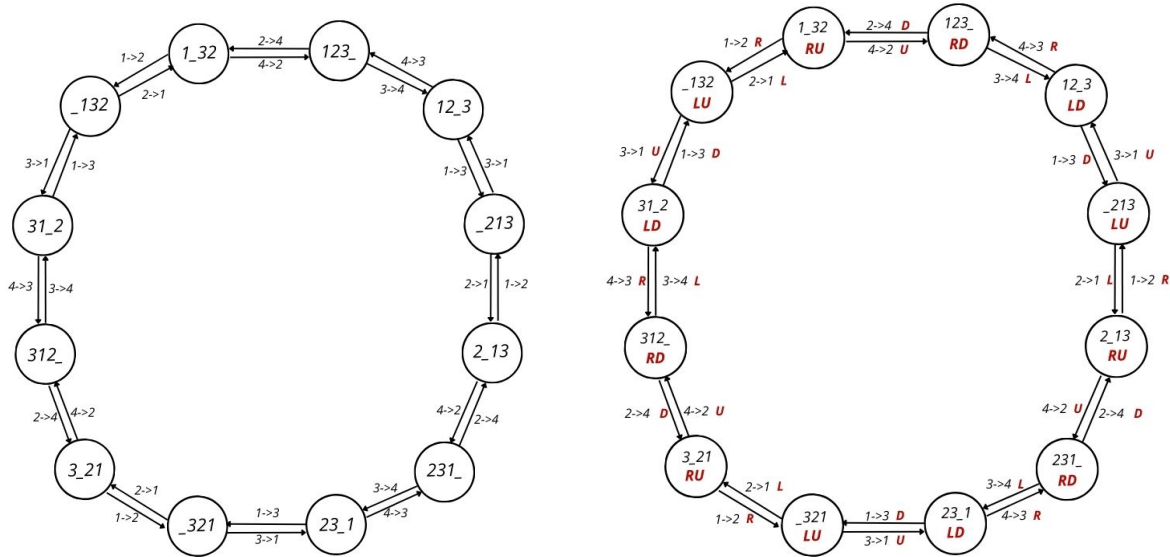


Figure 3.14: The action graph vs the pullback graph of a 3-puzzle game.

ones. From the affordance graph, the vertices are injectively mapped to the corresponding locations of the empty spaces, and the labels have no more than one choice to be mapped. If from the context  $RU$  we move to the context  $LU$  with affordance  $R$ , then  $R$  is mapped to the label  $1 \rightarrow 2$ , since only this transition is possible from the corresponding vertices in the abstract graph.

**Pullback Graph.** On the top level the Figure 3.13, we illustrate the 3-puzzle itself, as well as affordance and abstract graphs side by side, for comparison. The extended versions of the 3-puzzle are shown in the Figure 3.13 below 3-puzzle. Having defined three graphs and morphisms, we generate the pullback shown to the right of the Figure 3.14. It contains state-context information, and correspondence between transitions and affordances related to them.

### 3.1.7 Example - Generic Form

In this section provide an example of a user interfaces, modelled with directed labeled multi-graphs within a category theory framework. Let describe with directed labeled multigraphs a simple form consisting of a textbox and three buttons - **a**, **b**, **delete**. User Interface intends to provide the following functionality - user can typeset letters  $a$ ,  $b$ , and delete the last letter. We build graphs based on the assumption that the user is capable of recognizing and differentiating between states and perceiving provided affordances.

**Action Graph.** Before constructing an action graph, we define states. In our case essential part is a content in the text box, that is - a string that user perceives when interacting with a form. A capacity, etc. the longest string that can be stored, can be  $n$ . Therefore, possible states of the system are any strings containing  $a$ 's and  $b$ 's of length  $0 \leq m \leq n$ . A subgraph of the corresponding graph is shown in the figure 3.15.

**Affordance Graph.** It is essential to describe the set of affordances that user and interact with. In our simple form example, three actions are always available and non of the actions does not change context (a set of possible actions). Therefore a corresponding graph consists of one node  $\{a, b, \text{delete}\}$  and 3 edges - click  $a$ , click  $b$ , click  $delete$  (see figure 3.16).

**Abstract Graph.** Since all states must be glued together with a single context, and three distinct actions are observed, abstract graph is isomorphic to the affordance graph and is represented in the figure 3.17.

**Homomorphisms  $f$  and  $g$ .** A morphism from the action graph to the abstract graph is described below. A regex  $^{ab}^* \$$  indicates that all strings consisting of  $a$ 's and  $b$ 's are mapped to the single vertex in the abstract graph. It follows by the description of a morphism from the affordance graph to the abstract graph.

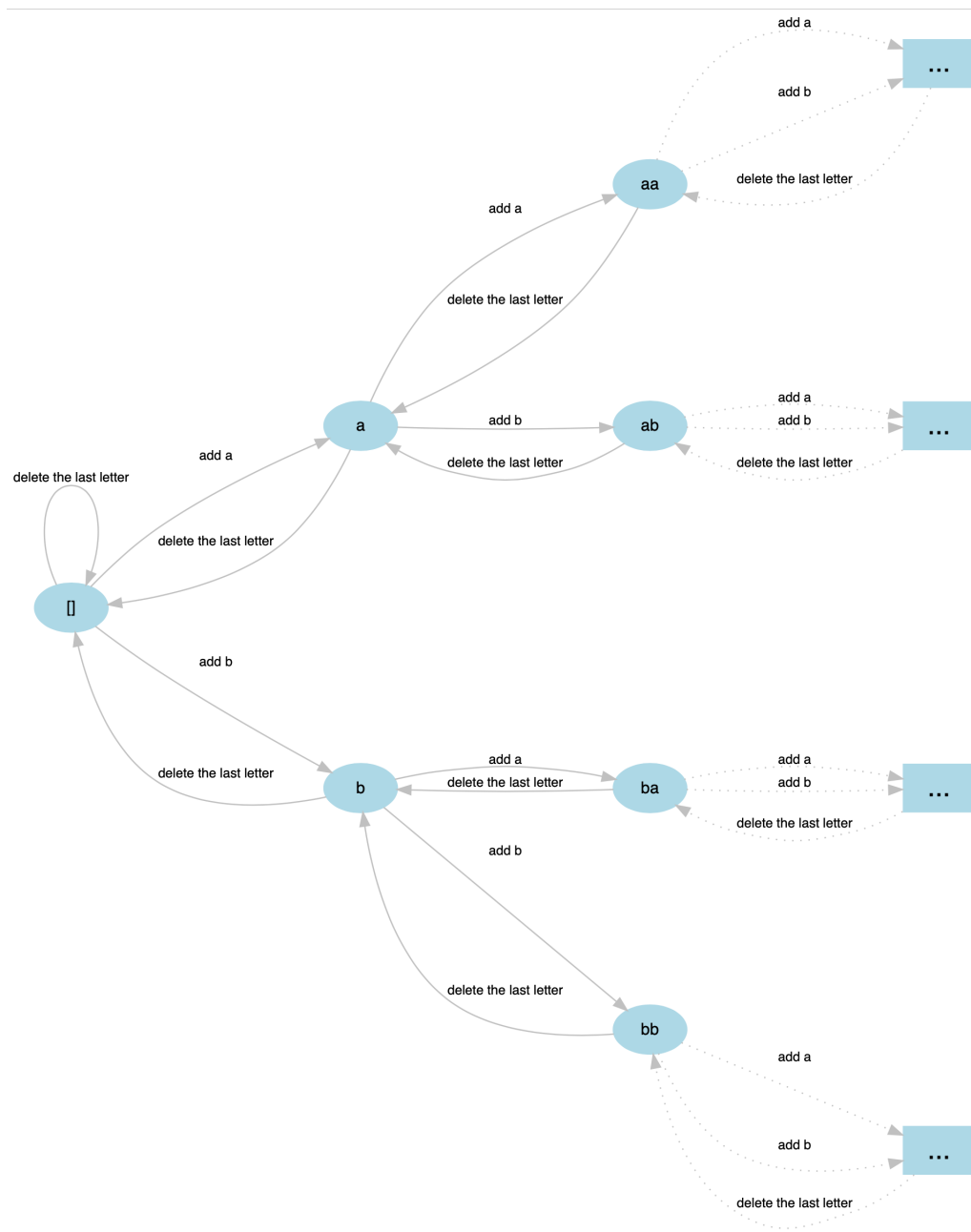


Figure 3.15: The action graph of a simple form with states defined by the content in the textbox.

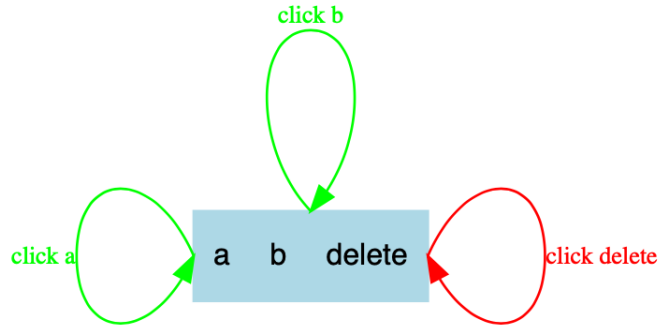


Figure 3.16: The affordance graph of a simple form that provides to the user a fixed set of possible actions (affordances) - **a**, **b**, and **delete**.

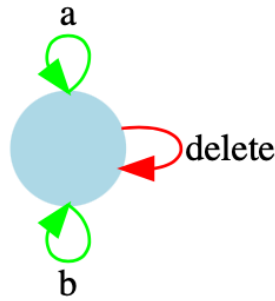


Figure 3.17: The abstract graph of a simple form.

| On Vertices                                         |                                        |
|-----------------------------------------------------|----------------------------------------|
| Source vertex $v$                                   | Target vertex $f^{vertex}(v)$          |
| $\hat{[ab]} * \$$                                   | $\circ$                                |
| On Labels                                           |                                        |
| Source edge label $\lambda$                         | Target edge label $f^{label}(\lambda)$ |
| add a                                               | a                                      |
| add b                                               | b                                      |
| delete the last symbol                              | delete                                 |
| On Edges                                            |                                        |
| Mapping follows from vertex and edge label mappings |                                        |

Table 3.5: The homomorphism from the action graph to the abstract graph of the generic form.

| On Vertices                                         |                                        |
|-----------------------------------------------------|----------------------------------------|
| Source vertex $v$                                   | Target vertex $f^{vertex}(v)$          |
| a, b, delete                                        | o                                      |
| On Labels                                           |                                        |
| Source edge label $\lambda$                         | Target edge label $f^{label}(\lambda)$ |
| click a                                             | a                                      |
| click b                                             | b                                      |
| click delete                                        | delete                                 |
| On Edges                                            |                                        |
| Mapping follows from vertex and edge label mappings |                                        |

Table 3.6: The homomorphism from the affordance graph to the abstract graph of the form.

**Pullback Graph.** A generated pullback is represented below. Notably, an appearance of the form is not encoded in the pullback itself, we provided an example just for visualization and better understanding. There can be many different UI’s designed from the pullback graph consisting of sets of possible actions together with system states and transitions between them.

## 3.2 Physical vs. Digital Interfaces

In describing the graphs, we did not specify interface types—physical or digital. With physical interfaces, affordances are natural, and there is typically a clear correspondence between states and affordances, making distinctions straightforward. In digital interfaces, however, the boundary between states and affordances is less defined, and their overlap requires careful consideration in choosing representation strategies. A favorite example is the contrast between physical and digital buttons. According to Norman, a physical button provides a real affordance, while a digital one is a perceived affordance. For digital buttons, the actual affordance is the screen (that allows us to touch it activating the digital button) or the mouse, which itself allows for clicking. The digital button serves primarily as a guide, showing where to click to achieve the expected outcome. In the context of digital interfaces, the state can include everything displayed on the screen, with the digital button becoming part of that state. A question arises for affordance graphs. Should a click on a digital button in the affordance graph be represented as a trigger for a button in a specific area of the screen or as a trigger for the screen/mouse itself?

Loeliger’s approach, which illustrates web pages using graphs, positioned states as titles with presented links and buttons on the page, while the affordance graph itself was not detailed [19]. However, we can deduce that it would rely on the device displaying the web page: for instance,

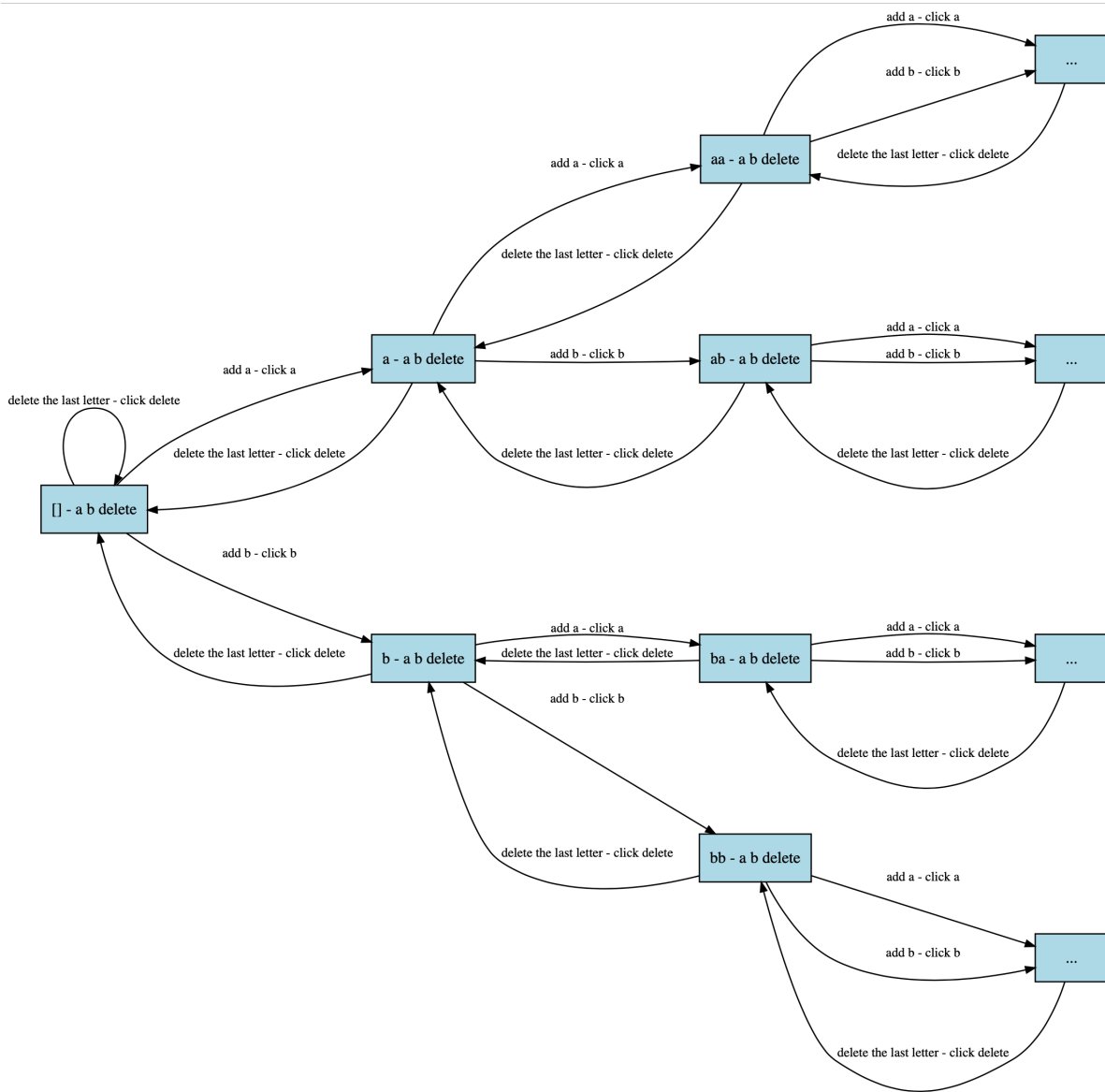


Figure 3.18: The pullback graph of a simple form. A form is encoded by a state of the textbox and buttons available. Edges describe resulting context-states pairs after actions realizations.

a laptop includes affordances like the screen, keyboard, and perhaps a mouse, while a tablet's affordances include the screen and possibly a stylus. Here, the affordance graph embodies the digital interface within the physical world, showing tools available to interact with information

on the screen. Her approach suggests that digital buttons and links are part of states rather than affordances. Our perspective differs in some sense. On one hand, regardless of screen content, a mouse click is always possible. However, analyzing the usability of the interface from the user's standpoint significantly relies on what is clicked. In this sense, the meaning behind pressing the button is significant, as is the user's interpretation of the action. Interface analysis, therefore, must consider two key factors: the embodied physical properties that enable interactions and the information they convey. So, how should a digital button press be depicted in the affordance graph? As a trigger for a physical affordance available to the user, or a trigger for a digital button on a physical device? Our answer is both. For instance, in an e-reader with physical and digital page-turn buttons, our representation should be consistent across both, as both of them afford us to do the same thing.

As Goguen points out, interfaces are mediators between abstract machines and users [10]. If digital interfaces are under consideration, more steps are needed. Let us show on the diagram,

**idea ↔ abstract machine (code) ↔ digital application ↔ hardware ↔ user**

Goguen considers each intermediate component in this chain as a semiotic morphism, whereas we interpret it as a pullback construction. The code embodies both its underlying concept and its implementation within the application. The application, in turn, contains the code used to create it and serves as the realization of that code on the computer on which it operates. Finally, the computer embodies the application, which is then processed and interpreted by a user. Each internal connection acts as a mediator between the adjacent components.

Returning to the topic of affordances, we identify two points in the system chain where affordances are present, within the digital interface and the hardware. Therefore, we consider both parts essential. The digital interface comprises perceived affordances and signifiers, bounded by a person's cultural and mental framework, while the hardware primarily offers physical affordances. Users are typically unaware of the other components; they neither see the code nor know the underlying idea. Focusing on the components that enable user actions, the construction of action and affordance graphs depends on problem interpretation. Loeliger structured graphs around the three right components of the chain, with two graphs containing affordances [19].

**digital application ↔ hardware ↔ user**

This approach was ideal in evaluation of application accessibility for different user profiles. While we acknowledge the importance of including affordances within the action graph, we think

an affordance graph should directly map the affordances a person perceives, both perceived and real. In the chapter 5, we will examine the function of affordances, their role within interfaces, and their influence on ambiguity. For this purpose, it is essential to include the component that links code to these affordances. Therefore, we propose the following approach for designing pullback components.

**abstract machine (code)  $\leftrightarrow$  digital application running on a physical machine  $\leftrightarrow$  user**

Coming back to the example of the form we illustrated, if it is digital, the current diagram omits the hardware component, which could be included by constructing another pullback structure. If puzzles are physical, they inherently encode an embodied connection with the user; however, if the puzzles are abstract, we would need to add an additional link that accounts for the user along with a digital graph containing physical affordances or work under the assumption that these affordances exist and are fully accessible to the user. This diagram aligns closely with the analysis of physical interfaces like a coffee machine, a light control system, a projector, and similar devices, that serve as mediators between the controls and a user.

**controls  $\leftrightarrow$  device  $\leftrightarrow$  user**

# Chapter 4

## Profiles

With directed labeled multigraphs, we can represent interfaces with varying levels of detail and user perception. Previous research highlighted the potential for creating user profiles, each corresponding to a unique way of encoding the interface as a pullback [19]. This encoding can be relevant to one group of users but not the others. With this in mind, the next steps focus on developing a theory centered on user profiles.

### 4.1 Pullback Relation to User Perception

Affordances are the actionable properties of objects in relation to an individual and describe what actions are possible within the user's mental, physical, and perceivable capacities [24]. Affordances contribute to mental models by allowing users to deduce possible actions without explicit instructions. However, when considering perceived affordances, the outcome of such deduction mostly relies on cultural conventions, like cursor shapes or clickable icons, which are learned symbols rather than actual affordances. These conventions help navigate screen-based systems and are related to the user's conceptual interaction model. Physical constraints restrict actions by people's physical capabilities, while logical and cultural constraints emerged as a result of user's way of reasoning or societal norms. It led to the motivation for developing a user profile, unique setup that outlines a person's physical, cultural, and cognitive constraints and how they influence their interface perception.

Let us give an example. Designers who know all about the entity they created can easily recognize its states and work with its features. They understand the results of their actions and how to reach the desired outcomes. They can also recognize all the indicators of possible

functions and assign meaning to them. However, the same system may be less intuitive for a first-time user without a technical background who is simply trying to solve a problem. They may not distinguish between states and physical limits could restrict their access to certain functions. So, the graph's representation changes based on the user's background. We suggest adjusting pullbacks or diagrams to fit specific user profiles. Based on this approach, the main actions for modifying pullbacks is to create an interface focused on a specific group of users.

**Observation 1.** *The pullback construction reflects the users' profile, distinguishing between all the states and the affordances and knowing the function of the affordances in a given context. There is no graph representation that covers all user profiles. Each profile has its own version of the pullback graph.*

**Proposition 12.** *Let  $P$  be a complete pullback above a diagram  $A \xrightarrow{f} M \xleftarrow{g} D$ . Let  $P'$  be a graph associated with the user profile obtained by modifying pullback graph in the following ways:*

1. *Collapsing states (from an action graph) by forgetting their features, or uniting them into equivalence classes.*
2. *Forgetting some details (mode, method) associated with affordance implementation from edges in an affordance graph.*
3. *Forgetting affordances by removing them from contexts and removing edges the represent the affordance trigger.*

Note that these pullback modifications may make the graph no longer a categorically universal construction for any diagram. In some cases, modifying the pullback results in a corresponding change in the diagram, making the new graph a universal construction. However, in other cases, modifying the graphs results in the inability to create a pullback. Therefore, the universal way to produce a graph associated to the user profile is to construct the pullback and then apply the necessary modifications. Let us consider in detail each of the ways to change the pullback.

**Forgetting state features.** By omitting certain state features, we reduce the level of detail, collapsing states previously differentiated by those features. For instance, two vectors,  $[0,0]$  and  $[1,0]$ , become a single state  $[0]$  when the first component is ignored. Importantly, none of the edges in the pullback graph are lost. This is an essential way of associating with a certain group of people who may not distinguish the states by characteristics or remember the difference between states, considering them as one.

**Example 4.1.1.** Let us revisit the example of text segments shown to the user in the book reader. Individuals with limited vision may struggle to read text below a certain size threshold. For

a particular hypothetical user profile, the text divided into three sections becomes unreadable. Therefore, we can categorize the three states into a single equivalence class called “inaccessible text,” glueing corresponding vertices of the action graph.

**Simplifying affordance details.** Affordance labels in a graph can include information such as the name of the affordance, its mode, and its method of implementation. We can omit details like the mode and method by simplifying the labels. This abstraction keeps the edges, as we are not removing affordances but reducing their realization level of detail. When there are modes or different implementation methods, it adds to the mental workload, which can sometimes be forgotten. A person may forget what mode the interface is in or that single and double-clicking the same button produces different results. It is critical to analyze the interface from the user’s perspective, unaware of these details.

**Example 4.1.2.** Consider a projector where a single button controls slide navigation. A single click advances to the next slide, while a double click returns to the previous one. For a profile of the user familiar with this setup, different labels on the edges will represent the button’s actions in distinct ways. However, for a user’s profile unaware of this difference, the same label will apply to both edges, ignoring the details that indicate how the affordance is activated.

**Forgetting affordances.** When affordances are forgotten, all contexts containing them lose an element, and the edges that represent the implementation of these affordances are removed from the graph. People with disabilities may have restricted access to certain affordances. Examining the interface from their perspective can provide understanding of the interface’s adaptability to different user profiles.

**Example 4.1.3.** If users are not provided with the possibility of choosing a language they understand, then a site does not present the majority of perceived affordances if they depend on the text.

At this stage, the concept of creating and categorizing profiles is still an introductory idea rather than a fully developed theory. It requires comprehensive study, which could stand as an independent direction within this research. Nevertheless, we believe it provides an important connection between interfaces from both technical and human perspectives and can act as a helpful formalism in HCI.

# Chapter 5

## Unambiguity of User Interfaces

Clear design is essential for a successful interface. As Donald Norman emphasizes, a user interface must make actions easy to recognize, reducing confusion and keeping things simple. It must also help users to avoid major mistakes while providing the flexibility needed to perform essential tasks. Therefore, ambiguity is the presence of multiple possible meanings, interpretations, or outcomes that can cause uncertainty or confusion.

In this chapter, we introduce conditions of unambiguity of the pullback and equivalent conditions on the diagram. The focus is on analyzing the unambiguity of the actions available to the user in some state and context, always from the perspective of the specific user profile. We focus on the local affects of specific affordances on the current states rather than on the study of the global affordance consistency among all states. It is important to mention that the definitions provided below are derived from observations of physical interfaces rather than digital ones. In digital interfaces, affordances may need to be included as a part of the state and it may be necessary to adjust the conditions of interface unambiguity, depending on the strategy for graphs construction.

### 5.1 Unambiguity

Let  $f$  denote a graph morphism from an action graph  $A$  to an abstract graph  $M$ , and let  $g$  denote a graph morphism from an affordance graph  $D$  to an abstract graph  $M$ . Let  $P$  denote a pullback over the diagram  $A \xrightarrow{f} M \xleftarrow{g} D$  together with two morphisms  $\pi_A$  and  $\pi_D$ .

**Definition 5.1.1.**  $E_\Gamma^{[v, -]} = \{e : \sigma_\Gamma(e) = v, v \in V_\Gamma \text{ \& } e \in E_\Gamma\}$  denotes a set of all edges in the directed labeled multigraph  $\Gamma$  where each edge  $e$  has the same source  $v$ .

**Definition 5.1.2** (Local Injectivity). Let  $f : \Gamma \rightarrow \Phi$  be a directed labeled multigraph morphism. We say  $f$  is *locally injective* with respect to edge labels if the following condition holds:

$$\forall v \in V_\Gamma \quad \forall e_1, e_2 \in E_\Gamma^{[v, -]} : \quad \lambda_\Gamma(e_1) \neq \lambda_\Gamma(e_2) \implies f^{label}(\lambda_\Gamma(e_1)) \neq f^{label}(\lambda_\Gamma(e_2))$$

**Definition 5.1.3.** Let  $P$  be a pullback over a diagram  $A \xrightarrow{f} M \xleftarrow{g} D$ . Let

$$V_P^s = \{v \in V_P : \pi_A^{vertex}(v) = s \in V_A\}$$

be a set of all representations of some state  $s$  within an interface  $P$ .

**Definition 5.1.4** (Determinism of DLMG). Let  $\Gamma$  be directed labeled multigraph. We say  $\Gamma$  is *deterministic* if the following holds:

$$\forall v \in V_\Gamma \quad \forall e_1, e_2 \in E_\Gamma^{[v, -]} : \quad \lambda_\Gamma(e_1) \neq \lambda_\Gamma(e_2) \vee \tau_\Gamma(e_1) = \tau_\Gamma(e_2)$$

**Definition 5.1.5.** [Unambiguity of a UI pullback] A user interface (UI) represented as a pullback graph  $P$  over a diagram  $A \xrightarrow{f} M \xleftarrow{g} D$  together with two morphisms  $\pi_A$  and  $\pi_D$  is *unambiguous* if the following holds:

$$\forall e_1, e_2 \in E_P \text{ such that } \sigma_P(e_1), \sigma_P(e_2) \in V_P^s \text{ for some state } s \in V_A,$$

$$\pi_D^{label}(\lambda_P(e_1)) = \pi_D^{label}(\lambda_P(e_2)) \implies \tau_P(e_1), \tau_P(e_2) \in V_P^{s'} \text{ for some state } s' \in V_A.$$

Moreover, if  $\sigma_P(e_1) = \sigma_P(e_2)$  then

$$\pi_D^{label}(\lambda_P(e_1)) = \pi_D^{label}(\lambda_P(e_2)) \implies \tau_P(e_1) = \tau_P(e_2)$$

Suppose the interface presents multiple representations of the same state  $V_P^s$ , with state  $s$  associated with different contexts  $c_1, c_2, \dots, c_n$ . In that case, any affordance  $a = \pi_D^{label}(\lambda(e))$  common to these contexts,  $a \in c_i \cap c_j$ ,  $1 \leq i, j \leq n$ , must consistently affect the state, regardless of the context. In other words, the effect of an affordance on a state representation does not lead to representations of different states.

**Theorem 5.1.1** (Unambiguity of an Interface Pullback). *Let  $P$  together with morphisms  $\pi_A$  and  $\pi_D$  be a **complete** pullback above the diagram  $A \xrightarrow{f} M \xleftarrow{g} D$ , where  $A$  is an action graph,  $D$  is an affordance graph, and  $M$  is an abstract graph.  $P$  is unambiguous if and only if the **diagram conditions** hold:*

- \* (Uniqueness) *For any edge  $e_A \in E_A$  there exist no more than one edge  $e_D \in E_D$  such that  $\sigma_D(e_D) = v_D$ ,  $\lambda_D(e_D) = l_D$  for some vertex  $v_D$  and label  $l_D$ , where  $f^{edge}(e_A) = g^{edge}(e_D)$ .*

\* (*Splitting Coherence*) For some vertex  $v_A$  in the action graph, if there exist  $e_A^1, e_A^2 \in E_A^{[v_A, -]}$ , then

$$\tau_A(e_A^1) \neq \tau_A(e_A^2) \implies f^{edge}(e_A^1) \neq f^{edge}(e_A^2) \wedge \lambda_D(e_D^1) \neq \lambda_D(e_D^2)$$

for some  $e_D^1, e_D^2 \in E_D$  such that  $f^{edge}(e_A^1) = g^{edge}(e_D^1)$  and  $f^{edge}(e_A^2) = g^{edge}(e_D^2)$ .

*Proof.* We will show that definition of unambiguity (U) implies the conditions (W), stated in the theorem, hold, and vice versa.

$U \implies W$  : Assume that a pullback  $P$  is unambiguous by definition, but diagram conditions do not hold. Then one of the conditions do not hold.

**Case 1:** Suppose the *uniqueness* condition fails regardless of the *splitting coherence* condition. Then for some edge  $e_A$  in the action graph  $A$  there exist at least *two* edges  $e_D^1, e_D^2$  in the affordance graph such that  $\sigma_D(e_D^1) = \sigma_D(e_D^2)$ ,  $\lambda_D(e_D^1) = \lambda_D(e_D^2)$ , and  $f^{edge}(e_A) = g^{edge}(e_D^1) = g^{edge}(e_D^2)$ .

Then the pullback  $P$  contains two edges,  $e_P^1 = (e_A, e_D^1)$  and  $e_P^2 = (e_A, e_D^2)$  such that  $\sigma_P(e_P^1) = \sigma_P(e_P^2)$ ,  $\lambda_D(\pi_D(e_P^1)) = \lambda_D(\pi_D(e_P^2))$ , but  $\tau_P(e_P^1) \neq \tau_P(e_P^2)$  because two edges from the affordance graph lead to different target contexts, otherwise  $e_D^1 = e_D^2$ . As a result, condition on the pullback unambiguity is violated and we come to a contradiction.

**Case 2:** Suppose, the *splitting coherence* condition fails regardless of the *uniqueness* condition. Then there exist two edges,  $e_A^1, e_A^2$ , in the action graph, and two edges,  $e_D^1, e_D^2$ , in the affordance graph such that  $\sigma_A(e_A^1) = \sigma_A(e_A^2)$ ,  $\tau_A(e_A^1) \neq \tau_A(e_A^2)$ ,  $f^{edge}(e_A^1) = g^{edge}(e_D^1)$ , and  $f^{edge}(e_A^2) = g^{edge}(e_D^2)$ .

1. If  $f^{edge}(e_A^1) = f^{edge}(e_A^2)$ , then  $g^{edge}(e_D^1) = f^{edge}(e_A^1) = f^{edge}(e_A^2) = g^{edge}(e_D^2)$ . Pick  $e_D^1$ . Then the pullback  $P$  contains two edges,  $e_P^1 = (e_A^1, e_D^1)$  and  $e_P^2 = (e_A^2, e_D^1)$ , such that  $\sigma_P(e_P^1), \sigma_P(e_P^2) \in V_P^s$  for state  $s = \sigma(e_A^1) = \sigma(e_A^2)$ . Also,  $\lambda_D(\pi_D^{edge}(e_P^1)) = \lambda_D(\pi_D^{edge}(e_P^2))$ , but  $\tau_P(e_P^1), \tau_P(e_P^2) \notin V_P^{s'}$  for some state  $s'$ , because targets in the action graph are not equal. As a result, condition on the pullback unambiguity is violated and we come to contradiction.
2. If  $\lambda_D(e_D^1) = \lambda_D(e_D^2)$  it follows that there are two edges in the pullback graph  $e_P^1 = (e_A^1, e_D^1)$  and  $e_P^2 = (e_A^2, e_D^2)$ , such that  $\sigma_P(e_P^1), \sigma_P(e_P^2) \in V_P^s$  for state  $s = \sigma_A(e_A^1) = \sigma_A(e_A^2)$ ,  $\lambda_D(e_D^1) = \lambda_D(e_D^2)$ , but  $\tau_P(e_P^1), \tau_P(e_P^2) \notin V_P^{s'}$ , contradicting an assumption about pullback unambiguity by definition.

$W \implies U$  : Suppose the diagram conditions hold, but pullback is ambiguous following definition. Then there exist two edges in the pullback graph such that  $\sigma_P(e_P^1), \sigma_P(e_P^2) \in V_P^s$  for state  $s = \sigma_A(e_A^1) = \sigma_A(e_A^2)$ ,  $\lambda_D(\pi_D^{edge}(e_P^1)) = \lambda_D(\pi_D^{edge}(e_P^2))$ , but  $\tau_P(e_P^1), \tau_P(e_P^2) \notin V_P^{s'}$  for some state  $s'$ .

**Case 1:** Suppose  $\pi_A^{edge}(e_P^1) = \pi_A^{edge}(e_P^2) = e_A$  (it is the same edge in the action graph). Then  $\pi_D^{edge}(e_P^1) \neq \pi_D^{edge}(e_P^2)$ . If  $\sigma_D(e_D^1) \neq \sigma_D(e_D^2)$  then pullback is not ambiguous, so we assume that  $\sigma_D(e_D^1) = \sigma_D(e_D^2)$ . Therefore, edges  $e_D^1, e_D^2$  differ by their targets,  $\tau_D(e_D^1) \neq \tau_D(e_D^2)$ . By the definition of pullback as a universal construction on the diagram,  $g^{edge}(\pi_D^{edge}(e_P^1)) = f^{edge}(\pi_A^{edge}(e_P^1)) = e_M = f^{edge}(\pi_A^{edge}(e_P^2)) = g^{edge}(\pi_D^{edge}(e_P^2))$ . Since for  $e_A$  there exist two edges  $e_D^1, e_D^2$  with the same label and source but different target, such that morphisms  $f$  and  $g$  map them to the same edge  $e_M$  in the abstract graph, uniqueness condition is violated and we come to contradiction.

**Case 2:** Suppose  $\pi_A^{edge}(e_P^1) \neq \pi_A^{edge}(e_P^2)$ .

1. If  $\tau_A(e_A^1) = \tau_A(e_A^2)$ , then  $g^{vertex}(\tau_D(e_D^1)) = f^{vertex}(\tau_A(e_A^1)) = f^{vertex}(\tau_A(e_A^2)) = g^{vertex}(\tau_D(e_D^2))$ . Since  $\sigma_A(e_A^1) = \sigma_A(e_A^2)$  (by definition), then  $g^{vertex}(\sigma_D(e_D^1)) = f^{vertex}(\sigma_A(e_A^1)) = f^{vertex}(\sigma_A(e_A^2)) = g^{vertex}(\sigma_D(e_D^2))$ . Moreover, by assumption pullback is ambiguous, and sources of affordance edges must be the same,  $\sigma_D(e_D^1) = \sigma_D(e_D^2)$ . The last but not least, labels of the affordance edges are the same, and all of this implies that  $g^{edge}(e_D^1) = g^{edge}(e_D^2)$ . Moreover, by assumption pullback is ambiguous, and sources of affordance edges must be the same  $\sigma_D(e_D^1) = \sigma_D(e_D^2)$ . Since for  $e_A^1$  (and  $e_A^2$ ) there exist two edges  $e_D^1, e_D^2$  with the same label and source, such that morphisms  $f$  and  $g$  map them to the same edge in the abstract graph, uniqueness condition is violated and we come to contradiction.
2. If  $\tau_A(e_A^1) \neq \tau_A(e_A^2)$ , it trivially shows that  $\lambda_D(e_D^1) \neq \lambda_D(e_D^2)$  since they are the same by assumption, and splitting coherence condition is violated. We come to contradiction.

We conclude that definition of a pullback unambiguity implies that uniqueness and splitting coherence conditions hold, and vice versa, uniqueness and splitting coherence conditions guarantee that pullback is unambiguous.  $\square$

In user-computer interaction terms, the uniqueness condition states that two realizations of the same affordance in the same context must not implement the same transition from the same state to the target. For example, a button that, when pressed, provides more than one result. This adds the feeling that user cannot control the interface and a randomness is present. Note that the effect of randomness is not necessarily related to the random selection of an action; it may

indicate an insufficient number of factors that would allow the distinction of two different states of the interface.

The splitting coherence condition tells us that if it is possible to move to different states from a certain state of the interface from a set of its representations, then the transition must be implemented by different affordances. If one affordance is responsible for different transitions, this creates ambiguity in defining its role and in building mental maps for the user.

However, it is not true the other way around and is often observed in different interfaces. To avoid unambiguity, one affordance should not be coupled with different state transitions, but one state transition can be realized with many affordances. For example, exit from the page can be performed by pressing different buttons or using a voice command, which can be available to the user at the same time. In this case, parallel arrows between the contexts indicate different affordances that perform the same action. Notably, if two affordance realizations are mapped by the morphism  $g$  into one certain arrow of the abstract graph, if considered to be the same. Formally,

**Definition 5.1.6.** Let  $e_A \in E_A$  represent a state transition, and let  $e_D^1, e_D^2 \in E_D$  denote two realizations of distinct affordances such that  $\lambda_D(e_D^1) \neq \lambda_D(e_D^2)$ . If  $f^{\text{edge}}(e_A) = g^{\text{edge}}(e_D^1) = g^{\text{edge}}(e_D^2)$ , then we define  $e_D^1$  and  $e_D^2$  as equivalent.

Next, we provide some results related to interface ambiguity in the context of the category of directed labeled multigraphs.

**Proposition 13.** *Let  $V_P^s$  be a set of all representations of a state  $s$  in a pullback  $P$ . If for any  $v_1$  and  $v_2$  in  $V_P^s$   $\pi_D(v_1) \cap \pi_D(v_2) = \emptyset$ , affordance graph  $D$  is deterministic, and splitting coherence condition holds, then the pullback  $P$  is unambiguous.*

*Proof.* Since sets of affordances are disjoint, there are no two edges  $e_1, e_2 \in E_P$  in the pullback graph such that  $\sigma_P(e_1), \sigma_P(e_2) \in V_P^s$  and  $\sigma_P(e_1) \neq \sigma_P(e_2)$  (since label of an edge in the affordance graph corresponds to the realization of one of the affordances from the context, a set of possible actions). Therefore, we need to review only the case where  $\sigma_P(e_1) = \sigma_P(e_2)$ . Since *splitting coherence* and affordance graph determinism holds, it guarantees that there are no two edges in the pullback such that  $\sigma_P(e_1) = \sigma_P(e_2)$ ,  $\pi_D^{\text{label}}(\lambda(e_1)) = \pi_D^{\text{label}}(\lambda(e_2))$  and  $\tau_P(e_1) = \tau_P(e_2)$ . Proof is similar to the proof of the theorem 5.1.1.  $\square$

**Proposition 14.** *The **diagram conditions** hold for a pullback  $P$  only if the pullback is both unambiguous and complete.*

*Proof.* Let us choose action and affordance graphs, along with morphisms  $f$  and  $g$ , such that the diagram conditions fail. Let action graph contain contain two edges  $e_1$  and  $e_2$  such that

$\sigma_A(e_1) = \sigma_A(e_2)$ . Consider an abstract graph consisting of two disconnected vertices, each with a self-loop. The action graph maps to one vertex with a loop, while the affordance graph maps to the other vertex with a loop. In this scenario, the pullback graph is empty graph, and is not unambiguous. However, a splitting coherence is violated, since  $f^{edge}(e_1) = f^{edge}(e_2)$ .  $\square$

**Proposition 15.** *Let  $P$  be a complete pullback graph above a diagram  $A \xrightarrow{f} M \xleftarrow{g} D$ .  $P$  is ambiguous if affordance graph  $D$  represents a single context, and action graph  $A$  is non-deterministic.*

*Proof.* Since  $D$  represents a single context, then  $|V_D| = 1$ , and all vertices in the pullback have the same affordance component, implying that for all vertices in the action graph  $f^{vertex}(v_A) = g^{vertex}(v_D)$ . If action graph is non-deterministic, there are two edges  $e_A^1, e_A^2 \in E_A$  for which the following is true,

$$\sigma_A(e_A^1) = \sigma_A(e_A^2), \quad \lambda_A(e_A^1) = \lambda_A(e_A^2), \quad \tau_A(e_A^1) \neq \tau_A(e_A^2).$$

Since  $f^{vertex}(\sigma_A(e_A^1)) = f^{vertex}(\tau_A(e_A^1)) = f^{vertex}(\tau_A(e_A^2))$ , and  $f^{label}(\lambda_A(e_A^1)) = f^{label}(\lambda_A(e_A^2))$ , we get  $f^{edge}(e_A^1) = f^{edge}(e_A^2)$ . However, it violates a *splitting coherence* condition on the diagram, and we conclude that pullback  $P$  is ambiguous.  $\square$

### 5.1.1 Strict Unambiguity

We consider an interface unambiguity represented by a pullback where one system state is represented by a single context to the user. There can be many contexts, but no more than one representation exists for each state. There are many real-world examples where UI only gives the user a single set of possible actions, such as a light control system, tap water regulator, coffee machine, or microwave oven. Therefore, we propose considering this section as a particular case of the user interface and will study a type of single-context user interface. We will adapt the unambiguity of the user interface to a single-context interface, as a *strict unambiguity* since we *restrict* system states to a single context. Notably, a theory on arbitrary interfaces applies to single-context interfaces as well.

**Definition 5.1.7** (Single-Context Interface). Let  $P$  be a pullback graph above a diagram  $A \xrightarrow{f} M \xleftarrow{g} D$  together with two morphisms  $\pi_A : P \rightarrow A$  and  $\pi_D : P \rightarrow D$ . We call  $P$  a *single-context interface* graph if the following holds:

$$\forall v_1, v_2 \in V_P : \pi_A^{vertex}(v_1) = \pi_A^{vertex}(v_2) \implies \pi_D^{vertex}(v_1) = \pi_D^{vertex}(v_2).$$

**Proposition 16** (Single-Context Interface). *Let  $P$  be a complete pullback graph above a diagram  $A \xrightarrow{f} M \xleftarrow{g} D$  together with two morphisms  $\pi_A : P \rightarrow A$  and  $\pi_D : P \rightarrow D$ .  $P$  is a single-context interface graph if and only if  $g$  is injective on vertices.*

*Proof.*  $\Rightarrow$  Suppose  $g$  is not injective on vertices. Then, there are  $v_1, v_2 \in V_D$ , such that  $g(v_1) = g(v_2)$ . Since pullback is complete, there exist some  $v_a \in V_A$  such that  $f(v_a) = g(v_1) = g(v_2)$ . It follows that pullback contains two vertices  $(v_a, v_1)$  and  $(v_a, v_2)$ . However, it implies that  $P$  is not a single-context interface and we come to contradiction.

$\Leftarrow$  Suppose pullback  $P$  is not a single-context interface representation. Then, there are two vertices  $v_1, v_2$  in the pullback such that  $\pi_A^{vertex}(v_1) = \pi_A^{vertex}(v_2)$  and  $\pi_D^{vertex}(v_1) \neq \pi_D^{vertex}(v_2)$ . From the definition of pullback as a categorical universal construction,  $f^{vertex}(\pi_A^{vertex}(v_1)) = g^{vertex}(\pi_D^{vertex}(v_1))$  and  $f^{vertex}(\pi_A^{vertex}(v_2)) = g^{vertex}(\pi_D^{vertex}(v_2))$ , since “action” parts are the same  $f^{vertex}(\pi_A^{vertex}(v_1)) = f^{vertex}(\pi_A^{vertex}(v_2))$  it follows that  $g^{vertex}(\pi_D^{vertex}(v_1)) = g^{vertex}(\pi_D^{vertex}(v_2))$  where  $\pi_D^{vertex}(v_1) \neq \pi_D^{vertex}(v_2)$ . We conclude that  $g$  is not injective on vertices and come to contradiction.  $\square$

**Proposition 17.** *If pullback  $P$  is a single-context interface pullback, then  $\pi_A$  is injective on vertices.*

*Proof.* Suppose  $\pi_A$  is not injective on vertices. Let  $(v_A^1, v_D^1)$  and  $(v_A^2, v_D^2)$  be two vertices in the pullback graph. Assume  $\pi_A^{vertex}(v_A^1, v_D^1) = v_A^1 = v_A^2 = \pi_A(v_A^2, v_D^2)$ . However, then there are two different representations  $(v_A^1, v_D^1)$  and  $(v_A^2, v_D^2)$  of the state  $v_A^1 = v_A^2$  contradicting a statement.  $\square$

**Proposition 18.** *If pullback  $P$  is a single-context interface pullback, then for any vertex  $s \in V_A$  the set of its representations in  $P$  contains a single element,  $|V_P^s| = 1$ .*

*Proof.* It implies from injectivity of  $\pi_A$  when pullback  $P$  is a single-context interface pullback. Uniformly arguing, a state  $s$  cannot be represented with more than one context.  $\square$

**Definition 5.1.8.** [Strict Unambiguity of a Single-Context Interface Pullback] A single-context user interface (UI) represented as a pullback graph  $P$  over a diagram  $A \xrightarrow{f} M \xleftarrow{g} D$  together with two morphisms  $\pi_A$  and  $\pi_D$  is *strictly unambiguous* if the following holds:

$\forall e_1, e_2 \in E_P^{[v, -]}$  for some vertex  $v \in V_P$ :

$$\pi_D^{label}(\lambda_P(e_1)) = \pi_D^{label}(\lambda_P(e_2)) \implies \tau_P(e_1) = \tau_P(e_2).$$

Similar to the general case, unambiguity implies the faithful action of any affordance with a name  $\pi_D^{label}(\lambda(e))$ . Its action on a certain state should not ambiguously lead to different resulting states.

**Theorem 5.1.2** (Strict Unambiguity of a Single-Context Interface Pullback). *Let  $P$  together with morphisms  $\pi_A$  and  $\pi_D$  be a **complete** pullback above the diagram  $A \xrightarrow{f} M \xleftarrow{g} D$ , where  $A$  is an action graph,  $D$  is an affordance graph, and  $M$  is an abstract graph.  $P$  is strictly unambiguous if the **splitting coherence condition** holds:*

*For some vertex  $v_A$  in the action graph and vertex  $v_D$  in the affordance graph, if there exist  $e_A^1, e_A^2 \in E_A^{[v_A, -]}$  and  $e_D^1, e_D^2 \in E_D^{[v_D, -]}$ , such that  $f^{edge}(e_A^1) = g^{edge}(e_D^1)$  and  $f^{edge}(e_A^2) = g^{edge}(e_D^2)$ , then*

$$\tau_A(e_A^1) \neq \tau_A(e_A^2) \implies f^{edge}(e_A^1) \neq f^{edge}(e_A^2) \wedge \lambda_D(e_D^1) \neq \lambda_D(e_D^2).$$

*Proof.*  $\Rightarrow$  Suppose a single-context pullback  $P$  is strictly unambiguous 5.1.8, but it does not follow that splitting coherence condition holds. Then there exist  $e_A^1, e_A^2 \in E_A^{[v_A, -]}$  and  $e_D^1, e_D^2 \in E_D^{[v_D, -]}$  such that  $f^{edge}(e_A^1) = g^{edge}(e_D^1)$  and  $f^{edge}(e_A^2) = g^{edge}(e_D^2)$ , where

$$\tau_A(e_A^1) \neq \tau_A(e_A^2) \wedge (f^{edge}(e_A^1) = f^{edge}(e_A^2) \vee \lambda_D(e_D^1) = \lambda_D(e_D^2)).$$

Consider the following cases:

**Case 1:**  $\lambda_D(e_D^1) = \lambda_D(e_D^2)$ . The pullback contains two edges  $e_1 = (e_A^1, e_D^1)$  and  $e_2 = (e_A^2, e_D^2)$ . Since  $\sigma_D(e_D^1) = \sigma_D(e_D^2)$  and  $\lambda_D(e_D^1) = \lambda_D(e_D^2)$ , then  $\sigma_P(e_1) = \sigma_P(e_2)$ , and  $\pi_D^{label}(\lambda_P(e_1)) = \pi_D^{label}(\lambda_P(e_2))$ . However,  $\tau_P(e_1) \neq \tau_P(e_2)$  since  $\pi_A^{vertex}(\tau_P(e_1)) \neq \pi_A^{vertex}(\tau_P(e_2))$ . Therefore, two edges coming from the same source and having the same affordance label lead to different targets, and *splitting coherence* condition of pullback unambiguity fails. We come to contradiction.

**Case 2:**  $f^{edge}(e_A^1) = f^{edge}(e_A^2)$ . We assume that  $g$  is injective on vertices, otherwise a contradiction follows. Since  $f^{edge}(e_A^1) = f^{edge}(e_A^2)$ , then  $g^{edge}(e_D^1) = g^{edge}(e_D^2)$ . Notably,  $f^{edge}(e_A^1) = f^{edge}(e_A^2) = g^{edge}(e_D^1) = g^{edge}(e_D^2)$ . Since  $g^{edge}(e_D^1) = f^{edge}(e_A^1) = f^{edge}(e_A^2)$ , there are two edges in the pullback  $e_1 = (e_A^1, e_D^1)$  and  $e_2 = (e_A^2, e_D^1)$ , where  $\tau_A(e_A^1) \neq \tau_A(e_A^2)$ , while  $\lambda_D(\pi_D^{edge}(e_1)) = \lambda_D(\pi_D^{edge}(e_2))$  since they are the same edge in the affordance graph. Thus, *splitting coherence* condition is violated, leading to a contradiction.

$\Leftarrow$  Suppose, a diagram condition is met but it does not follow that pullback is unambiguous. Then, there are two edges  $e_1, e_2$  such that  $\sigma_P(e_1) = \sigma_P(e_2)$ ,  $\pi_D^{label}(\lambda_P(e_1)) = \pi_D^{label}(\lambda_P(e_2))$ , but  $\tau_P(e_1) \neq \tau_P(e_2)$ . Let us review the cases where  $\tau_P(e_1) \neq \tau_P(e_2)$ .

**Case 1:** Suppose  $\pi_D^{vertex}(\tau_P(e_1)) \neq \pi_D^{vertex}(\tau_P(e_2))$  and  $\pi_A^{vertex}(\tau_P(e_1)) = \pi_A^{vertex}(\tau_P(e_2))$ . However, it implies that  $P$  is not a single-context interface pullback as stated in the theorem.

**Case 2:** Suppose  $\pi_A^{vertex}(\tau_P(e_1)) \neq \pi_A^{vertex}(\tau_P(e_2))$  regardless of equality or inequality of the affordance component. Since these two edges exist in the pullback, then  $f^{edge}(\pi_A^{edge}(e_1)) = g^{edge}(\pi_D^{edge}(e_1))$  and  $f^{edge}(\pi_A^{edge}(e_2)) = g^{edge}(\pi_D^{edge}(e_2))$ . However  $\tau_A(\pi_A^{edge}(e_1)) \neq \tau_A(\pi_A^{edge}(e_2))$  does not imply that  $\lambda_D(\pi_D^{edge}(e_1)) \neq \lambda_D(\pi_D^{edge}(e_2))$  since they must be equal by definition. We come to contradiction.

We conclude that conditions are equivalent.  $\square$

**Proposition 19.** *If pullback  $P$  is strictly unambiguous then it is unambiguous.*

*Proof.* We will prove that strict unambiguity implies unambiguity by contradiction. Suppose a pullback  $P$  is strictly unambiguous but not generally unambiguous. Then there exist two edges  $e_1, e_2 \in E_P$  such that  $\sigma_P(e_1), \sigma_P(e_2) \in V_P^s$  for some state  $s$ . If  $|V_P^s| > 1$ , then the state  $s$  is represented by more than context, there exist at least two vertices  $v_1, v_2$  such that  $\pi_A^{vertex}(v_1) = \pi_A^{vertex}(v_2)$  and  $\pi_D^{vertex}(v_1) \neq \pi_D^{vertex}(v_2)$ . Then, strict unambiguity conditions do not apply. We suppose that for any state  $s$   $|V_P^s| = 1$ . Then being not unambiguous implies that there exist two edges  $e_1, e_2 \in E_P$  where  $\sigma_P(e_1) = \sigma_P(e_2)$ , and  $\pi_D^{label}(\lambda_P(e_1)) = \pi_D^{label}(\lambda_P(e_2)) \not\Rightarrow \tau_P(e_1) = \tau_P(e_2)$ . However, condition of *splitting coherence* is then trivially violated, since splitting coherence conditions become equivalent. We come to contradiction and conclude that strictly unambiguous pullback interface is generally unambiguous.  $\square$

**Proposition 20.** *Let  $P$  be an unambiguous pullback. If for any state  $s \in V_A$  a set of its representatives contains at most one vertex  $|V_P^s| = 1$ , then it is strictly unambiguous.*

*Proof.* Since for any state  $s$ ,  $|V_P^s| = 1$ , then pullback  $P$  trivially becomes a single-context interface pullback. Condition on the unambiguity decreases to the following one:

If  $\sigma_P(e_1) = \sigma_P(e_2)$  then

$$\pi_D^{label}(\lambda_P(e_1)) = \pi_D^{label}(\lambda_P(e_2)) \implies \tau_P(e_1) = \tau_P(e_2),$$

which is equivalent to the condition on strict unambiguity. Therefore if it is satisfied and pullback is relaxed unambiguous by assumption, then strong unambiguity follows.  $\square$

**Proposition 21.** *Let  $q$  be a morphism from a single-context pullback  $P$  to a graph  $P'$  such that,*

1.  $\forall v \in V_P : q^{vertex}(v) = \pi_A^{vertex}(v),$
2.  $\forall \lambda \in \Lambda_P : q^{label}(\lambda) = \pi_D^{label}(\lambda),$

$$3. \forall e \in E_P : q^{vertex}(\sigma_P(e)) = \sigma_{P'}(q^{edge}(e)), q^{vertex}(\tau_P(e)) = \tau_{P'}(q^{edge}(e)), q^{label}(\lambda_P(e)) = \lambda_{P'}(q^{edge}(e)).$$

*Pullback  $P$  is strictly unambiguous if and only if the graph  $P'$  is deterministic.*

*Proof.*  $\Rightarrow$  Suppose, pullback  $P$  is strictly unambiguous, but  $P'$  is non-deterministic. Since all edge labels are replaced by their “affordance” part, then non-determinism of graph  $P'$  indicates that in the original pullback  $P$  there are two edges  $e_1, e_2$  such that  $\sigma_P(e_1) = \sigma_P(e_2)$ ,  $\tau_P(e_1) \neq \tau_P(e_2)$  (since morphism  $q$  maps vertices to their state parts that differ), and their labels are the same due to non-determinism of  $P'$ . It implies, that there is some affordance that leads to different states, contradicting assumption about pullback  $P$  unambiguity.

$\Leftarrow$  Suppose, a graph  $P'$  is deterministic, but  $P$  is ambiguous. Ambiguity of  $P$  implies that there are two edges  $e_1, e_2 \in E_P^{[v, -]}$  for some vertex  $v \in V_P$  such that  $\pi_D^{label}(\lambda_P(e_1)) = \pi_D^{label}(\lambda_P(e_2))$ , and  $\tau_P(e_1) \neq \tau_P(e_2)$ . Then applying  $q$  to edges  $e_1, e_2$  results in two edges  $q^{edge}(e_1), q^{edge}(e_2)$ , where

$$q^{vertex}(\sigma_P(e_1)) = q^{vertex}(\sigma_P(e_2)) \implies \pi_A^{vertex}(\sigma_P(e_1)) = \pi_A^{vertex}(\sigma_P(e_2)),$$

$$q^{label}(\lambda_P(e_1)) \neq q^{label}(\lambda_P(e_2)) \implies \pi_D^{label}(\lambda_P(e_1)) = \pi_D^{label}(\lambda_P(e_2)).$$

Since  $P$  represents single-context interface, if vertices are not equal, then their “action” components differ. Thus,

$$q^{vertex}(\tau_P(e_1)) \neq q^{vertex}(\tau_P(e_2)) \implies \pi_A^{vertex}(\tau_P(e_1)) \neq \pi_A^{vertex}(\tau_P(e_2)).$$

The, the resulting graph  $P'$  is non-deterministic, contradicting our assumption.  $\square$

## 5.1.2 Profiles and Unambiguity

In the previous section, we indicated that the pullback with the highest level of details can be modified in three proposed ways: forgetting state features or uniting states, forgetting affordance implementation details, and forgetting affordances. We will study how these modifications affect the pullback’s state of ambiguity.

**Observation 2.** *By merging states in the pullback  $P$ , the resulting graph may transition from unambiguous to ambiguous, or conversely, from ambiguous to unambiguous.*

For example, consider part of the graph below. The first version of the graph is unambiguous. Then, we merge the two vertices  $(s_0, c_0), (s'_0, c'_0)$  into one, and the edges leading from them to different targets have the same labels, so the ambiguity arises. However, then merging the targets  $(s_1, c_1), (s'_1, c'_1)$  removes it.

$$\begin{array}{ccccc}
(s'_0, c'_0) & (s_0, c_0) & & [s_0, c_0] & [s_0, c_0] \\
(l_A, l_D) \downarrow & \downarrow (l_A, l_D) & \longrightarrow & (l_A, l_D) \swarrow \searrow (l_A, l_D) & \longrightarrow \downarrow (l_A, l_D) \\
(s'_1, c'_1) & (s_1, c_1) & & (s'_1, c'_1) & (s_1, c_1) & [s_1, c_1]
\end{array}$$

**Observation 3.** *The pullback graph  $P$  from unambiguous can become ambiguous by forgetting the affordance trigger details.*

For example, consider part of the graph below. The first version of the graph is unique because the affordance labels  $l_D^1, l_D^2$  are different. However, forgetting the details, we consider them the same as shown on the right, causing ambiguity.

$$\begin{array}{ccc}
[s_0, c_0] & & [s_0, c_0] \\
(l_A, l_D^1) \swarrow \searrow (l_A, l_D^2) & \longrightarrow & (l_A, [l_D]) \swarrow \searrow (l_A, [l_D]) \\
(s'_1, c'_1) & (s_1, c_1) & (s'_1, c'_1) & (s_1, c_1)
\end{array}$$

**Observation 4.** *The pullback graph from ambiguous can become unambiguous by forgetting the affordances.*

For example, consider part of the graph on the previous page. The first version of the graph is ambiguous. However, removing the edge associated with the affordance will also lead to ambiguity. At the same time, the interface for this user profile loses its functional component and becomes incomplete.

$$\begin{array}{ccc}
[s_0, c_0] & & [s_0, c_0] \\
(l_A, l_D) \swarrow \searrow (l_A, l_D) & \longrightarrow & \downarrow (l_A, l_D) \\
(s'_1, c'_1) & (s_1, c_1) & (s'_1, c'_1) & (s_1, c_1)
\end{array}$$

### 5.1.3 When Ambiguity is Detected

Having determined the conditions of unambiguity within the categorical representation of interfaces, in this section we will discuss the reasons for the vulnerability of interfaces, and recommendations for their improvement. Each of the above reasons applies to a specific user profile. That is, for one profile the action can be unambiguous, while for another it is not.

**Unrecognizable states.** The first source of ambiguity we will examine is the user's inability to differentiate between states where the same affordance has different effects. For example, a button that both turns a device on and off. If an important condition, referred to here as a state feature, is required to activate the device, and the user cannot detect whether this condition is met, starting the device becomes ambiguous. The outcome then depends on the presence or absence of the feature, which the user may not be aware of. In abstract terms, if a pullback is ambiguous and two edges emerge from a vertex where the affordance is identical but lead to different resulting states, the cause could be the merging of two states that should trigger distinct actions for the same affordance. However, due to the inability to distinguish between these states, ambiguity arises. In this case, it is necessary to introduce additional elements into the design to enable the distinction between the states.

**Hidden complexities.** A less obvious example of ambiguity is the presence of internal dependencies between states and contexts. This issue is closely tied to the concept of implicit modes discussed earlier. When one action only takes effect if another has already been performed, it introduces dependencies that are not always easy to identify or remember. Additionally, if affordances are poorly designed, their appearance might not indicate the existence of these dependencies. If identical light switches are set up such that one depends on the other, it becomes challenging for the user to understand the dependency and even more challenging to capture the meaning or purpose behind it. However, complexities are not always a negative aspect of design. For instance, we do not want the start button of a coffee maker to work if there is no water or coffee beans. In this case, the button's functionality depends on the water and coffee levels, which is an example of effective design. On the other hand, this dependency is not hidden and is intuitive to people who want to make coffee. Therefore, we recommend making all implicit modes explicit, and remove hidden complexities where necessary.

**Limited number of affordances.** Another source of ambiguity arises from the multiple functions assigned to individual affordances. When a single affordance controls several actions, it complicates the user's understanding of the relationship between affordances and their functions within the interface. This is often due to explicit or implicit modes, which allow an affordance to perform different actions in the same state. While this can be an efficient engineering solution, enabling a wide range of functionality with limited affordances, it poses challenges from a design perspective. Users must continually be aware of the current mode and the specific function an affordance performs in that mode, placing an additional cognitive load on them. According to Norman, additional mental effort does not indicate effective design. A recommended approach would be introducing more affordances or representations for some affordances. Recall that from our point of view, modes should provide hints about the state of the interface, but not change the functional role of affordances.

Next, we will present overviews of ambiguous interfaces, each being an example of the aforementioned reasons for ambiguity. We will construct the graphs based on the interface descriptions and use implemented software on the .Net platform, to verify the pullback ambiguity. Our implementation of unambiguity checking can be accessed at [unambiguity repository](https://github.com/mzheplin/UIGraphs)<sup>1</sup>.

## 5.2 Case Study of a Generic Form

Previously, we gave an example of a generic form with the following possible actions - pressing buttons a, b, and delete, and changing active position. The deletion was only possible for the last letter of the text in the text box. Let us consider the use case of a form in which it is possible to delete letters in any position without restrictions, but it is impossible to determine the position. This scenario is based on the real queries of people who could not observe a cursor in some interactive systems, such as command line prompt in Windows 10. There can be many interpretations here - a certain group of people (profile) cannot see it with certain settings, or the error is on the part of the software, and recognition of the active position for removing and adding symbols is basically impossible.

### Affordance Graph

The following affordances are available on the form: three buttons - a, b, delete - and two arrows for changing the active position. When interacting with any of these affordances, the context does not change. We get an affordance graph with one vertex and five arrows, each of which is associated with one of the affordances. For visualization, see figure below.

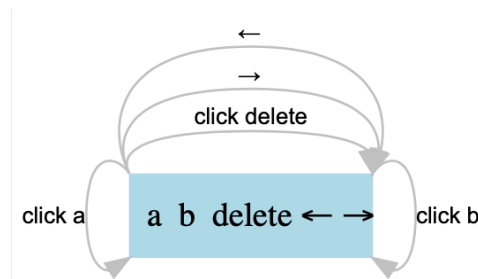


Figure 5.1: The affordance graph of the generic form, where five possible actions are available.

<sup>1</sup><https://github.com/mzheplin/UIGraphs>

## Action Graph

The text in the text box defines the states of the interface. Since the active position is unknown, the action of changing the position does not visually change the state. The action of deleting or

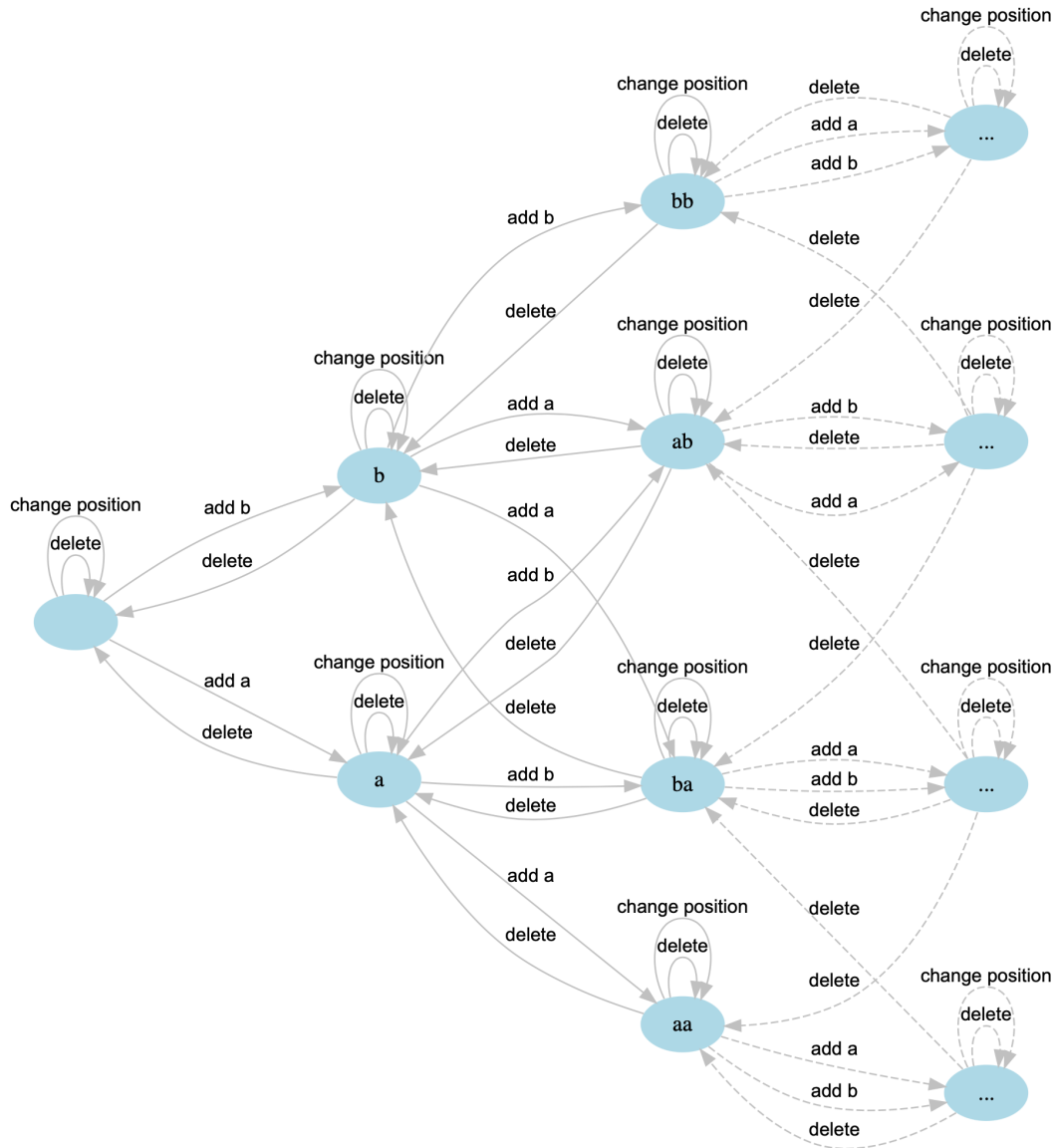


Figure 5.2: The action graph of the generic form, where a state is defined by the textbox content.

adding characters, in many cases, is non-deterministic because the result of their action depends on the position of the cursor (or any other indicator). For example, deleting a symbol where the state is “ab” may result in states “a,” “b,” or “ab.” Note that this action graph can be obtained from the graph where the cursor position is available by forgetting this feature (see section 4.1). For vizualization, see figure 5.2.

## Abstract Graph and Morphisms

Since the same context represents all states in the action graph, the morphism maps them to a single vertex in the abstract graph, where the single affordance context is also mapped. Each unique edge in the action graph corresponds to an edge in the abstract graph. The affordance edges are mapped to their counterparts in the abstract graph according to the transitions each affordance triggers. Note that the left and right arrows represent a position change transition, so they map to the same label and edge in the abstract graph. The resulting abstract graph has one vertex and four edges. Below, we provide the reader with the details of the morphisms  $f$  and  $g$  from the affordance and action graphs to the abstract graph.

## Pullback Graph

The calculated pullback graph integrates the information from both graphs. The next page provides a pullback graph of some interface providing text manipulation. Each state of the interacting system is connected to other states with the corresponding arrows, that contain information about transition name and about affordance trigger.

## Interface Ambiguity

The interface is naturally ambiguous. In this scenario, *Splitting Coherence Condition* of interface unambiguity fails many times. Half of the affordances behave non-deterministically, while the

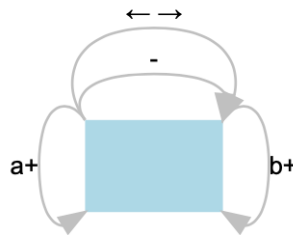


Figure 5.3: The abstract graph of the generic form.

| On Vertices                                         |                                        |
|-----------------------------------------------------|----------------------------------------|
| Source vertex $v$                                   | Target vertex $g^{vertex}(v)$          |
| a, b, delete                                        | o                                      |
| On Labels                                           |                                        |
| Source edge label $\lambda$                         | Target edge label $g^{label}(\lambda)$ |
| click a                                             | a+                                     |
| click b                                             | b+                                     |
| click delete                                        | -                                      |
| $\leftarrow$                                        | $\leftarrow \rightarrow$               |
| $\rightarrow$                                       | $\leftarrow \rightarrow$               |
| On Edges                                            |                                        |
| Mapping follows from vertex and edge label mappings |                                        |

Table 5.1: The homomorphism from the affordance graph to abstract graph, failed form example.

| On Vertices                                         |                                        |
|-----------------------------------------------------|----------------------------------------|
| Source vertex $v$                                   | Target vertex $f^{vertex}(v)$          |
| $\wedge [ab] \$$                                    | o                                      |
| On Labels                                           |                                        |
| Source edge label $\lambda$                         | Target edge label $f^{label}(\lambda)$ |
| add a                                               | a+                                     |
| add b                                               | b+                                     |
| delete                                              | -                                      |
| change position                                     | $\leftarrow \rightarrow$               |
| On Edges                                            |                                        |
| Mapping follows from vertex and edge label mappings |                                        |

Table 5.2: The homomorphism from the action graph to abstract graph, failed form example.

rest do not affect either state or context. As mentioned earlier, this ambiguity can arise from the inability to distinguish between states, leading to non-determinism. Note that we could achieve such a graph by constructing a complete set of different states and then applying the forgetting morphism to the vertices of the pullback, where they will be mapped to the corresponding one without the cursor position (see section 4.1). To resolve ambiguity and make the interface unambiguous, we need to add details that allow more explicit interaction with the text in the textbox. For instance, providing a visual representation of the active cursor position increases the number of states in the action graph. For example, “ab” could expand to “/ab,” “a/b,” or “ab/,” and adding or removing characters would directly impact these states. Deletion and adding symbols are not

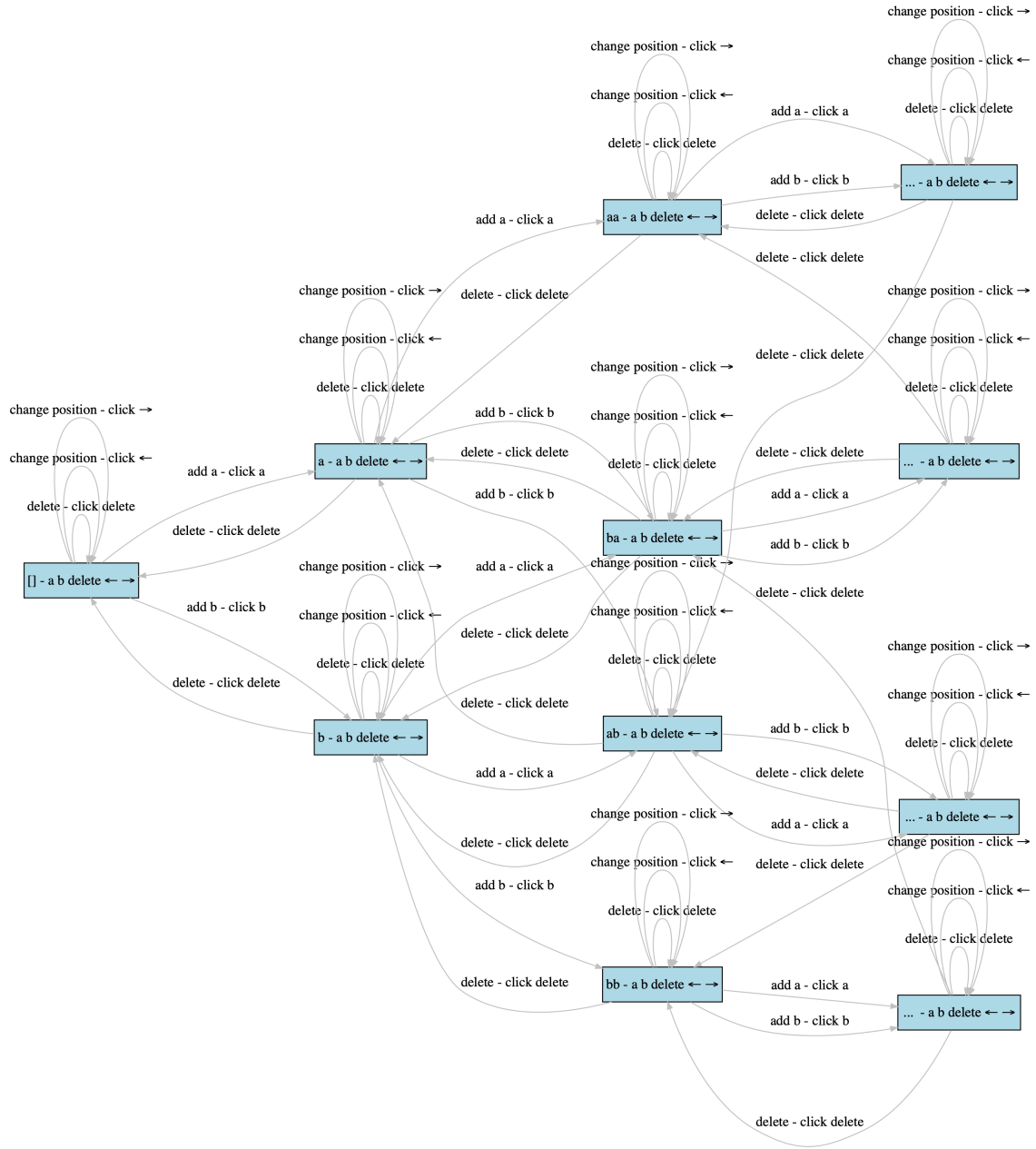


Figure 5.4: The pullback graph of the generic form that enables user interaction with the text.

inherently non-deterministic actions on the state `[ab]`, even though they may seem that way. The

key issue is that multiple states share the same representation, “ab,” which creates ambiguity. As a result, the detection, depending on the interpretation of the ambiguity, may indicate that the interface itself does not have enough visual clues, the addition of which will get rid of the ambiguity.

### 5.3 Case Study of a Generic Form - 2

Let us consider another case of the form where ambiguity is present. The idea of creating this example occurred based on a real text editor. Let the space for affordances be limited, and the designer solves this issue by creating modes. The following affordances are available - a, d, and two radio buttons. When one of the radio buttons is pressed, it becomes active, and the other becomes inactive, indicating a mode change. As in the previous example, button a adds the letter a to the text box, but button d in one mode adds d, and in the other, it removes the last letter of the text.

#### Affordance Graph

The affordance graph contains two vertices, each representing two versions of similar contexts with a set of affordances *a*, *d*, and *switch radio buttons*. Its active radio button characterizes each of the contexts. The implementation of the affordances *switch radio buttons* switches the mode; the implementation of affordances *a* and *d* leaves it unchanged. We will add information about the mode in which the affordance is implemented on the edge labels. This will allow us to correctly combine the names of affordances and modes with the actions they implement. The resulting affordance graph is shown in the figure 5.5.

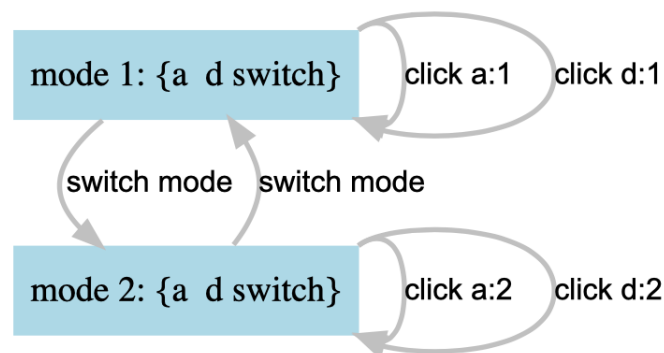


Figure 5.5: The affordance graph of a generic form, where two modes are present.

## Action Graph

The content of the textbox determines the interface's current state. The available transitions between states are adding "a," adding "d," and deleting the last character. Additionally, there are unnamed loops on the state, representing potential actions that do not change the state but will be necessary for context switches. Next page contains a visualization of the action graph. Note that in this example, the action graph is completely deterministic.

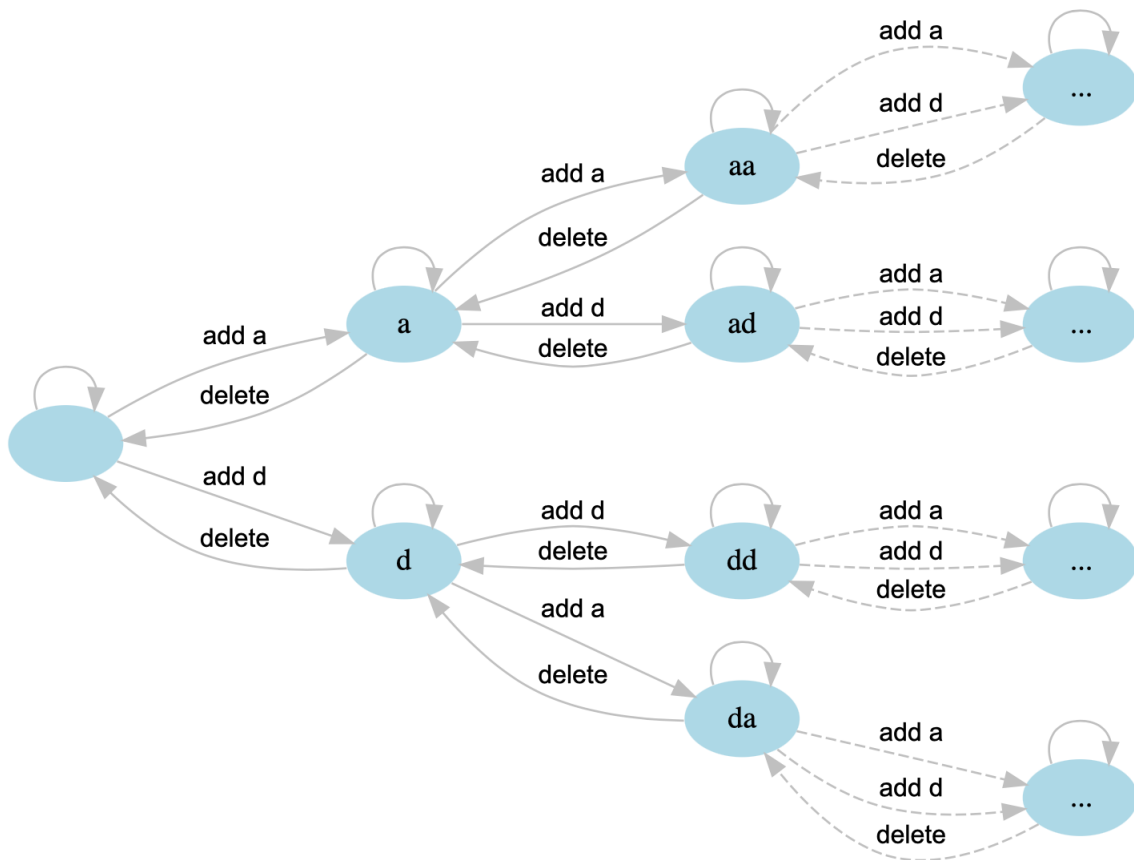


Figure 5.6: The action graph of a generic form, where a state is defined by a text in the text box.

## Abstract Graph and Morphisms

Since each state can be in each mode, the abstract graph will contain only one vertex, to which all the vertices of both, the affordance of the graph, the action of the graph, map. Mapping on

| On Vertices                                         |                                        |
|-----------------------------------------------------|----------------------------------------|
| Source vertex $v$                                   | Target vertex $f^{vertex}(v)$          |
| $\wedge [ad] \$$                                    | $\circ$                                |
| On Labels                                           |                                        |
| Source edge label $\lambda$                         | Target edge label $f^{label}(\lambda)$ |
| add a                                               | a                                      |
| add d                                               | d:1                                    |
| delete                                              | d:2                                    |
| $\rightarrow$                                       | switch                                 |
| On Edges                                            |                                        |
| Mapping follows from vertex and edge label mappings |                                        |

Table 5.3: The homomorphism from the action graph to abstract graph of failed form 1.2.

| On Vertices                                         |                                        |
|-----------------------------------------------------|----------------------------------------|
| Source vertex $v$                                   | Target vertex $g^{vertex}(v)$          |
| mode 1: a, b, delete                                | $\circ$                                |
| mode 2: a, b, delete                                | $\circ$                                |
| On Labels                                           |                                        |
| Source edge label $\lambda$                         | Target edge label $g^{label}(\lambda)$ |
| click a:1, click a:2                                | a                                      |
| click d:1                                           | d:1                                    |
| click d:2                                           | d:2                                    |
| switch                                              | switch                                 |
| On Edges                                            |                                        |
| Mapping follows from vertex and edge label mappings |                                        |

Table 5.4: The homomorphism from the affordance graph to abstract graph of failed form 1.2.

labels and the edges themselves is even more interesting. A mode switch and an unnamed action on a state are one and the same, so there is one edge and one label in the abstract graph for this pair. Moreover, control  $a$  adds  $a$  within any mode, so  $a - a$  pair also has an edge label and an edge. However, affordance  $d$  in mode 1 adds  $d$ , and in mode 2 removes the last character, so each case has a corresponding edges. Notably, without information about the mode on the labels, we could not build the pullback precisely because of the distribution of the affordance  $d$  on different actions. The resulting abstract graph contains one vertex and four edges with unique labels (see figure 5.7). Details on graph morphisms are given above.

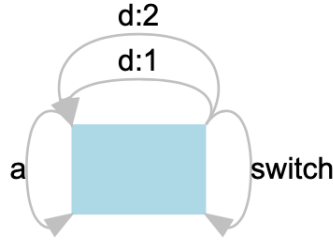


Figure 5.7: The abstract graph of the failed form 1.2.

### Pullback Graph

We obtained a pullback where the form with a single state appears twice, corresponding to each mode. Below, we provide the graph that results from specific modifications to the pullback. As explained in the section on profiles, an interface modification corresponds to a particular user profile. Let us consider the profile of a typical user encountering the interface for the first time. For this user, the position of the active radio button is not informative. The goal is to examine how the buttons influence the textbox’s content. As mentioned in the section on profiles, we can ignore the mode information on the edge labels. As a result, “d:1” and “d:2” will simply be labeled as “d.” The modified graph is provided on the next page.

### Ambiguity

The ambiguity of the interface is present, and it is caused by the different effects of affordance on the same state in different representations. For example, let us apply the realization of the affordance  $d$  to the state “a”. In one representation, the result will be “ad”, and in the other empty string. Therefore, the interface is ambiguous by definition. We recall that an interface is considered unambiguous, where the action of one affordance in any context translates one state in any representation into a single state in a certain representation. We observe that for each state of the action graph, there are three transitions leading to three other states, realized only with two affordances. Ambiguity can be avoided by adding affordance or replacing affordance  $d$  with another in one of the modes.

## 5.4 Case Study of Light Control System (E5 Seminar Room)

This section presents studies of a real-world example of an unintuitive user interface for controlling room lighting. We demonstrate how the proposed method can formally model the interface

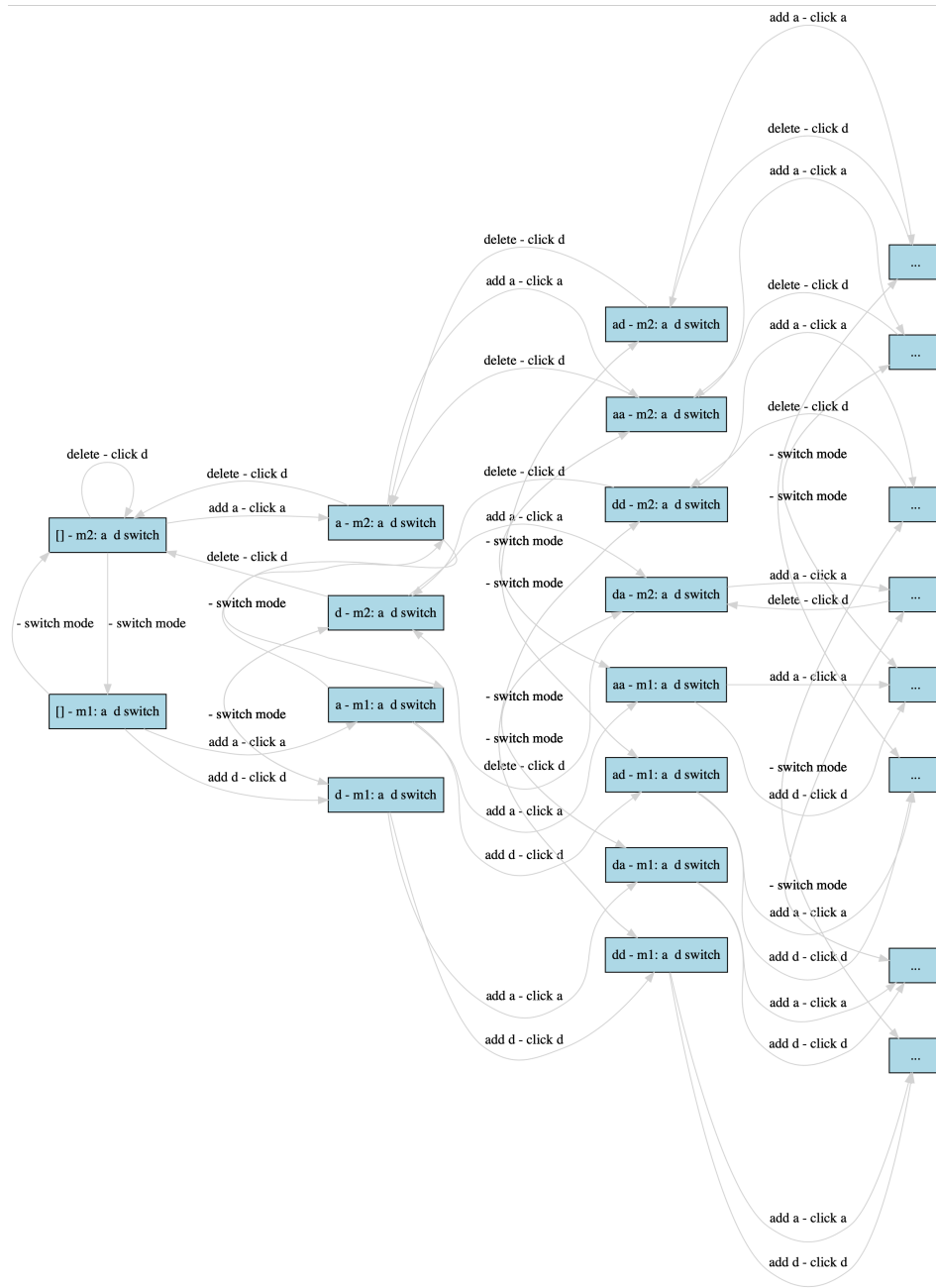


Figure 5.8: The pullback graph for a generic form that enables user interaction with the text within two modes.

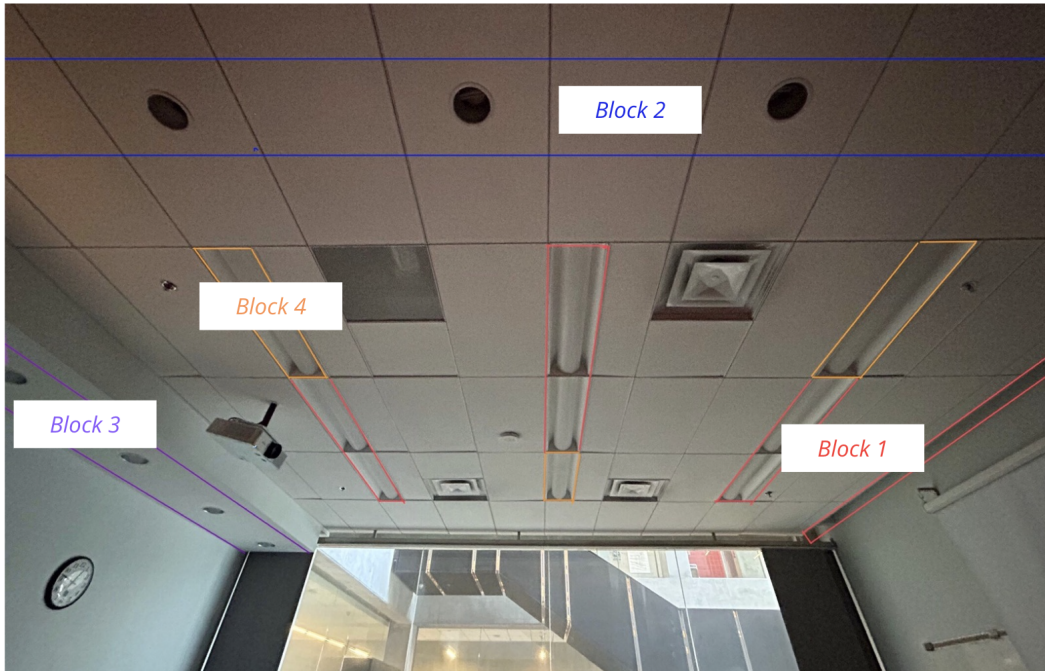


Figure 5.9: Four light blocks in the seminar room correspond to the buttons and are associated with specific colors (1 - red; 2 - blue; 3 - purple; 4 - orange).

and identify its flaws. In this example, we assume that the user knows the buttons' functions and understands the connection between them and the light blocks they control. Thus, our focus is solely on the feedback the user receives upon interaction. The feedback consists of two elements: the change in the button's state and the corresponding change in the room's lighting.

**Details on the room lighting control.** The room's bulb sets are divided into 4 separate blocks, each connected to specific buttons as illustrated in the picture below. Each block corresponds to a number associated with buttons controlling it and distinct colors to visually represent the room's lighting state. The room light is managed through 8 buttons: 4 for turning the light on and 4 for turning it off. Firstly, we number the buttons from left to right. The upper buttons also have the suffix  $t$ , while the lower buttons have the  $b$ . A formal representation of the light control is provided in the figure 5.10.

**Description of light control functionality.** The following are the results of the interaction with the lighting system. Pressing button  $4t$  turns on the lights in block 4, while button  $4b$  turns off this light when pressed. When activated, buttons  $1t$  and  $2t$  light up their backlights. However, they

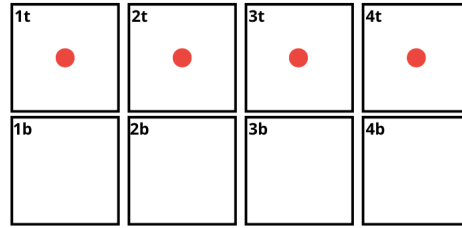


Figure 5.10: Formal representation of light control consisting of 8 buttons. The upper and lower buttons are enumerated from 1 to 4, and suffixed with ‘t’ and ‘b’, respectively.

only illuminate the room under certain conditions. If the room is not lit or block 4’s light is on, pressing these buttons will not light the corresponding blocks. These buttons change the context without affecting the room lighting. Users can press buttons *1b* and *2b* to turn off the backlights of buttons *1t* and *2t*. The third set of buttons (*3t* and *3b*) adds more complexity. These buttons control the light in block 3. Pressing *3t* turns on the light in block 3, and any blocks linked to *1t* and *2t* illuminated buttons. Pressing *3b* turns off block 3 and any linked blocks but does not turn off the backlights, which must be deactivated by pressing *1b* and *2b*. Lights in blocks 1 and 2 can only be turned on if block 3 is already lit, creating a dependency. When block 3’s light is on, pressing buttons *1t* and *2t* activates lights in units 1 and 2, while pressing *1b* and *2b* turns them off. Therefore, units 1 and 2 can only be illuminated if both their buttons are on and block 3’s light is active. Buttons *1t* and *2t* do not light up when pressed, confusing users trying to associate the button with the lights. Also, the backlights on buttons *1t* and *2t* serve more of an engineering function, which may cause further confusion as they do not provide visual feedback, making the system more complicated to understand.

### Affordance Graph

First, only two buttons, *1t*, and *2t*, are illuminated. Accordingly, affordances in the form of buttons, being in different representations, form explicit modes. A visual change in affordance prompts the user to assume a change in state or functionality. We encode each mode through a pair of  $x$  numbers, where each can have a value of 1 or 0, where  $x$  encodes the illumination of the first button, and  $y$  is for the illumination of the second. Therefore, affordance graph contains four explicit modes. However, the effect of buttons *1t* and *2t* on the state of illumination depends on the illumination of block 3. Block 3 is illuminated by pressing button *3t*. As a result, implicit modes are formed and connected with realizations of affordances *3t*, *3b*. As a result, the affordance graph contains two copies of each mode. Notably, pressing buttons *4t* and *4b* in all contexts, and others in specific contexts, does not change the context/mode, so self-loops in the affordance graph represent them, although not visualized.

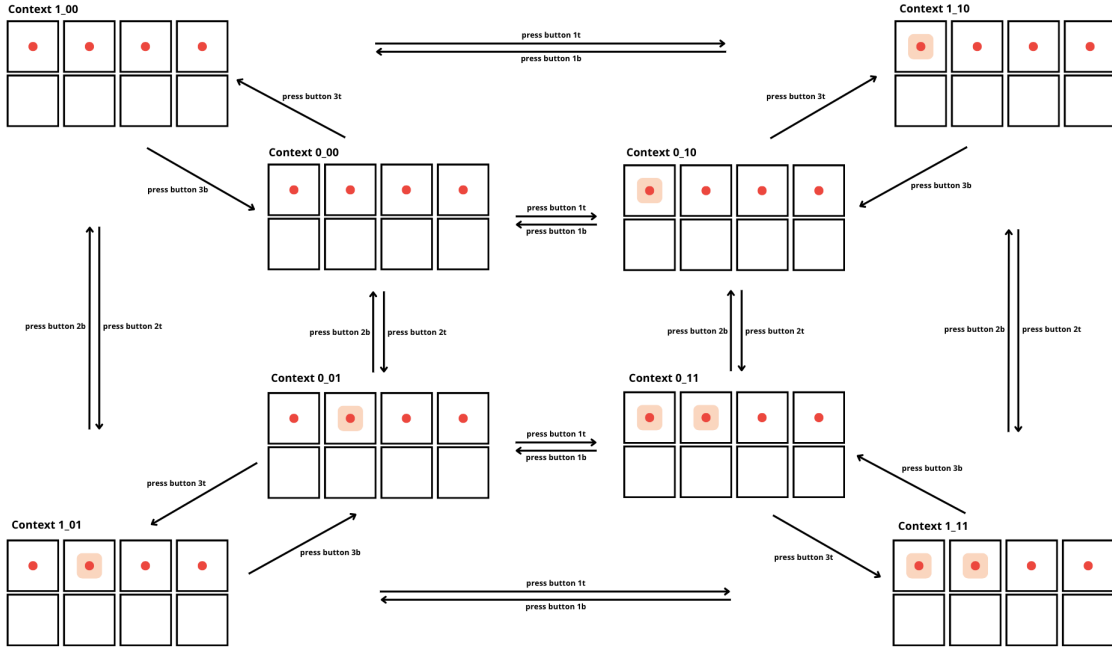


Figure 5.11: The affordance graph of the light contexts in the seminar room.

## Action Graph

Let us build an action graph. As mentioned earlier, we assume that the target user knows the correspondence between the buttons and the light blocks that turn on. Let us depict the room's lighting conditions as vectors consisting of 4 digits, each of which is associated with a block of light and can have a value of 0 - unlit and 1 - lit. For example, 0001 encodes the state of the room, in which only the fourth block is lit. Note that the action graph only states what the user can observe. For example, state 1000 is impossible due to the dependence of the first block of light on the third. A transition exists between states if an action can be performed that changes the first state to the last state. The edge label indicates which blocks have changed state. The corresponding graph is in the figure 5.12.

## Abstract Graph and Homomorphisms

Let us build an abstract graph. At this stage, we will match the action and affordance graphs, answering the question of how the components combine. First, if the room is dark or only the light of the 4<sup>th</sup> unit is on, then changing modes will not change the states, so they are composed of internal copies of all modes. When pressing 3t, we proceed into a state with a precise association with only one of the modes. Therefore, the abstract graph has five vertices, one for

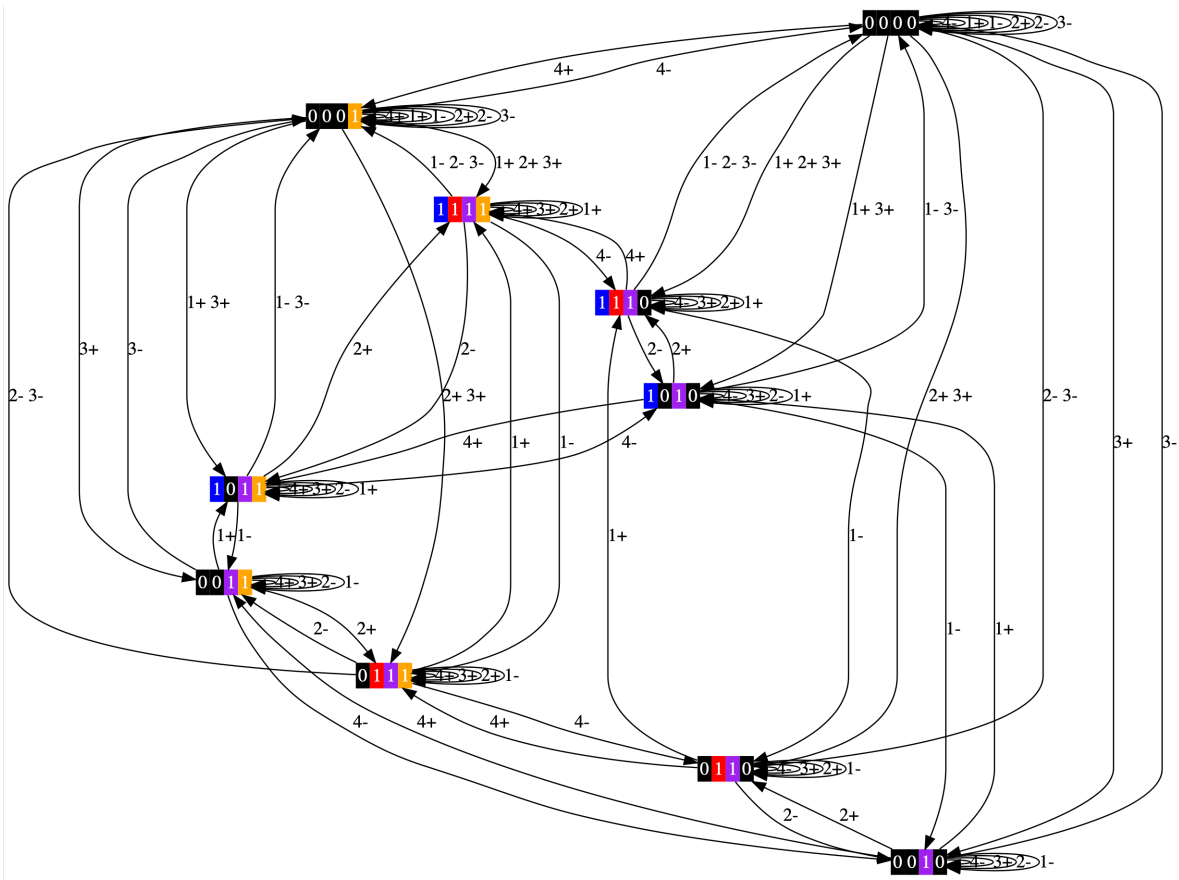


Figure 5.12: The action graph of the light states in the seminar room.

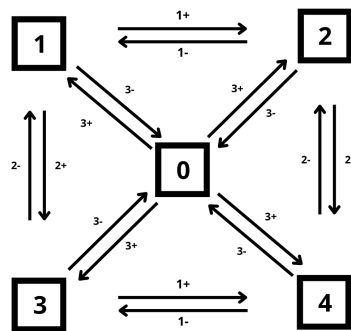


Figure 5.13: The abstract graph of the light control in the seminar room.

| On Vertices                                         |                                        |
|-----------------------------------------------------|----------------------------------------|
| Source vertex $v$                                   | Target vertex $g^{vertex}(v)$          |
| 0_00, 0_01, 0_10, 0_11                              | 0                                      |
| 1_00                                                | 1                                      |
| 1_10                                                | 2                                      |
| 1_01                                                | 3                                      |
| 1_11                                                | 4                                      |
| On Labels                                           |                                        |
| Source edge label $\lambda$                         | Target edge label $g^{label}(\lambda)$ |
| $\{x\}t$                                            | $\{x\}+$                               |
| $\{x\}b$                                            | $\{x\}-$                               |
| $x \in \{1, 2, 3, 4\}$                              |                                        |
| On Edges                                            |                                        |
| Mapping follows from vertex and edge label mappings |                                        |

Table 5.5: The homomorphism from the affordance graph to the abstract graph, light control example.

states without association with modes and four more for those that do. We have not illustrated self-loops similar to those in the affordance graph. However, information about them is given in the tables describing the morphisms. Also, the edge labels are paired one-to-one, except for the  $3t$  label, which corresponds to several functions at once. A detailed description of the graphs homomorphisms from the action and the affordance graphs to the abstract graph and the abstract graph are given in the figure 5.13.

### Pullback Construction

Using a software implementation, we constructed a pullback over the proposed diagram  $A \xrightarrow{f} M \xleftarrow{g} D$ , where  $A$  is the action graph,  $M$  is the abstract graph,  $D$  is the affordance graph,  $f$  is a homomorphism from action to abstract graph, and  $g$  is a homomorphism from affordance to abstract graph. The resulting visualization is shown in the Figure 5.14. Each vertex in the graph consists of two components and is represented by digits,  $xyzw - ab_c$ , where  $x, y, z, w, a, b, c \in \{0, 1\}$ . The left component corresponds to the action graph, while the right component corresponds to the affordance graph. Additionally, we rendered the vertices with four colored segments, each indicating whether the corresponding block's light is on. If a block is not lit, it is colored black. The pullback edge labels also include both the action and affordance labels from the graph. However, for clarity and to conserve space, we display only the affordance label, which identifies the button ID among the eight possible options.

| On Vertices                                         |                                        |
|-----------------------------------------------------|----------------------------------------|
| Source vertex $v$                                   | Target vertex $g^{vertex}(v)$          |
| 0000, 0001                                          | 0                                      |
| 0010, 0011                                          | 1                                      |
| 1010, 1011                                          | 2                                      |
| 0110, 0111                                          | 3                                      |
| 1110, 1111                                          | 4                                      |
| On Labels                                           |                                        |
| Source edge label $\lambda$                         | Target edge label $g^{label}(\lambda)$ |
| $\{x\}+$                                            | $\{x\}t$                               |
| $\{x\}-$                                            | $\{x\}b$                               |
| $x \in \{1, 2, 4\}$                                 |                                        |
| $3+, 1+3+, 2+3+, 1+2+3+$                            | $3t$                                   |
| $3-, 1-3-, 2-3-, 1-2-3-$                            | $3b$                                   |
| On Edges                                            |                                        |
| Mapping follows from vertex and edge label mappings |                                        |

Table 5.6: The homomorphism from the action graph to abstract graph, light control example.

### Ambiguous?

The pullback analysis with our software shows some key issues: pressing buttons  $3t$  and  $3b$  leads to different outcomes depending on the context, violating the condition on pullback unambiguity. While the same room state might exist in other contexts, buttons  $1t$ ,  $1b$ ,  $2t$ , and  $2b$  only act as context switches when the light in unit 3 is off. When the light in unit 3 is on, these buttons directly affect the room's lighting. We will analyze the potential actions to remove ambiguity. Our first suggestion is to introduce more states. In this example, ambiguity arises in complete darkness and when only the orange block's light is on. To resolve this, these states should be split into additional states based on the different actions of the affordances. For instance, the dark state remains unchanged when buttons 1 and 2 are pressed, as blocks 1 and 2 stay unaffected. Adding more states can create four possible combinations: 1000, 0100, 1100, and 0000. We can assign a unique action to button  $3t$  with these four distinct states, removing unambiguity. Also, button  $3t$  and  $3b$  cause implicit modes. We can change the state of one of the buttons to create an explicit mode. In the example of the first two buttons, pressing them leads to illumination. It will be informative to know that the block 3 related to this button is enabled. The main reason for the ambiguity is the action of the  $3t$  button. The result of its pressure can be the activation of one, two, or three blocks of light. Being inattentive, users may not understand the reasons for

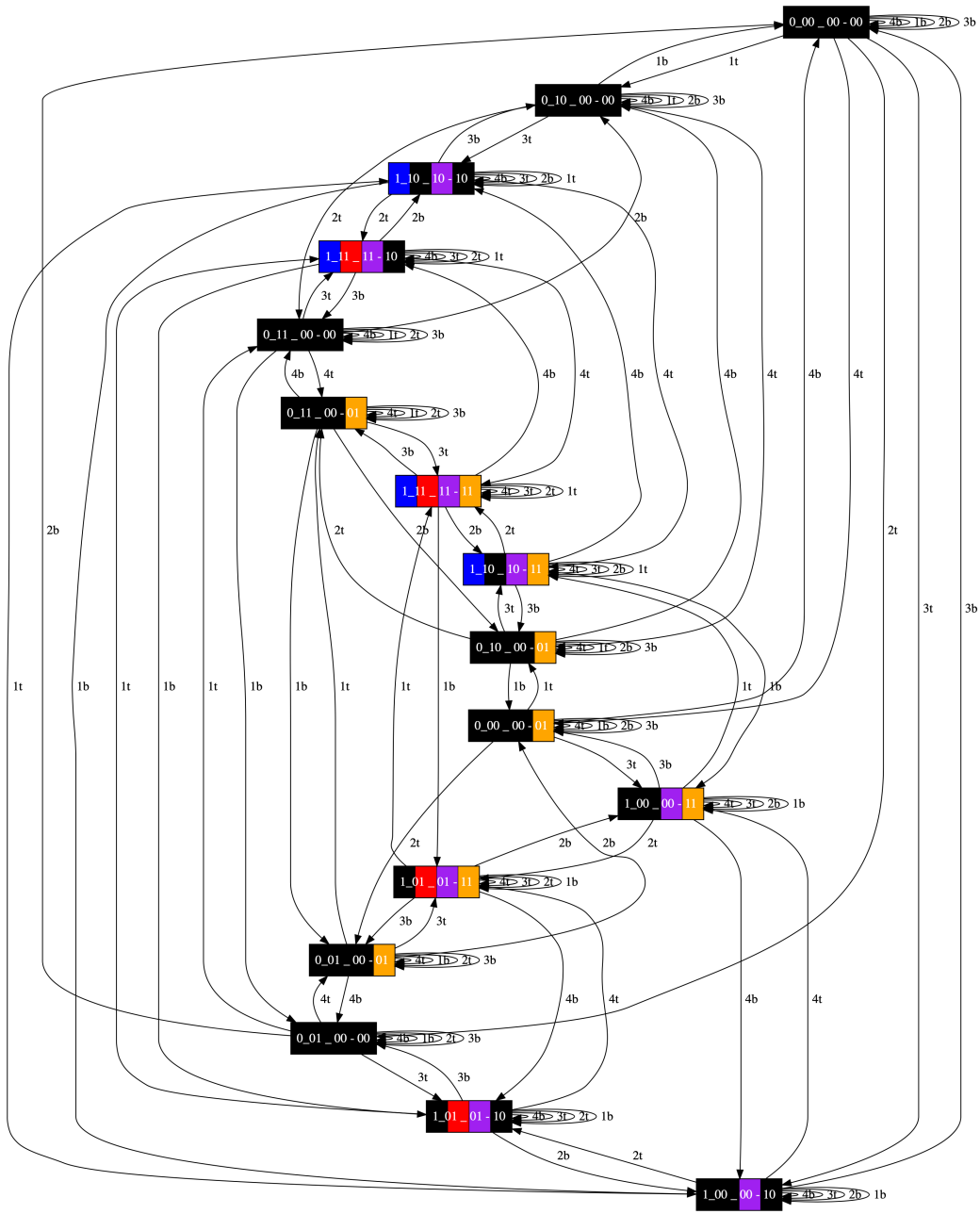


Figure 5.14: The pullback graph of the light control interface.

this. However, we want to point out another potential problem that is beyond the scope of this example with this choice of graph construction strategy. As mentioned earlier, buttons  $1t$  and  $2t$  work depending on the state. If a person does not see this dependence, their action either has an effect or does not. The first thought of a person is that the button does not work, although, in fact, there is an intricate control behind it. In order to analyze the interface from this point of view, we will choose a different strategy, relying more on the user unawareness. We will show this strategy in the next smaller example, with two blocks of light bulbs and, this time, two buttons.

### 5.4.1 Strategy “Feedback”

Let us return to the example of room lighting control. The user does not see the difference between the buttons or know which one controls the light. Feedback is significant to the user. Also, the user cannot distinguish buttons with the same appearance. In this example, we apply the concrete user profile from the previous example to the lighting control system. As we mentioned, modify our pullback graph without a need to build a new diagram and redefine three graphs with morphisms to satisfy the description. We will change pullback graph in the following way:

**Vertices:**  $xyzw - ab \mapsto s - t$ ;

**Edge labels:**  $\lambda \mapsto \{\text{click not illuminated, click illuminated, click bottom}\}$ ;

**Edges** are collapsed, if between two vertices there are multiple edges with the same edge labels after modifications;

Notably, provided map is not a homomorphism. Each state is represented in the form  $xyzw - ab$ , where all digits are either 0 or 1. We map this to a specific state  $s - t$ , where  $s$  represents the number of ones on the left side, and  $t$  represents the number of ones on the right. Consequently,  $s$  corresponds to the number of lit blocks, and  $t$  reflects the number of illuminated buttons from the top row. The edge labels correspond to one of three possible actions: click not illuminated, click illuminated, click a bottom button. Since all buttons look the same and serve the single function of turning on the light, this transformation in the feedback-focused graph is appropriate and allows for ambiguity analysis. The generated graph is provided in the Figure 5.15.

Considering the labels of the affordance graph as representations of the appearance of the buttons involved in the interaction, we notice that the uniqueness condition of the graph is violated multiple times. The action involving illuminated buttons is ambiguous, as it does not alter the states. Pressing non-illuminated buttons leads to ambiguous changes in states and contexts, sometimes affecting them and sometimes not. In this scenario, constructing a clear action-consequence conceptual model within the control system becomes challenging.

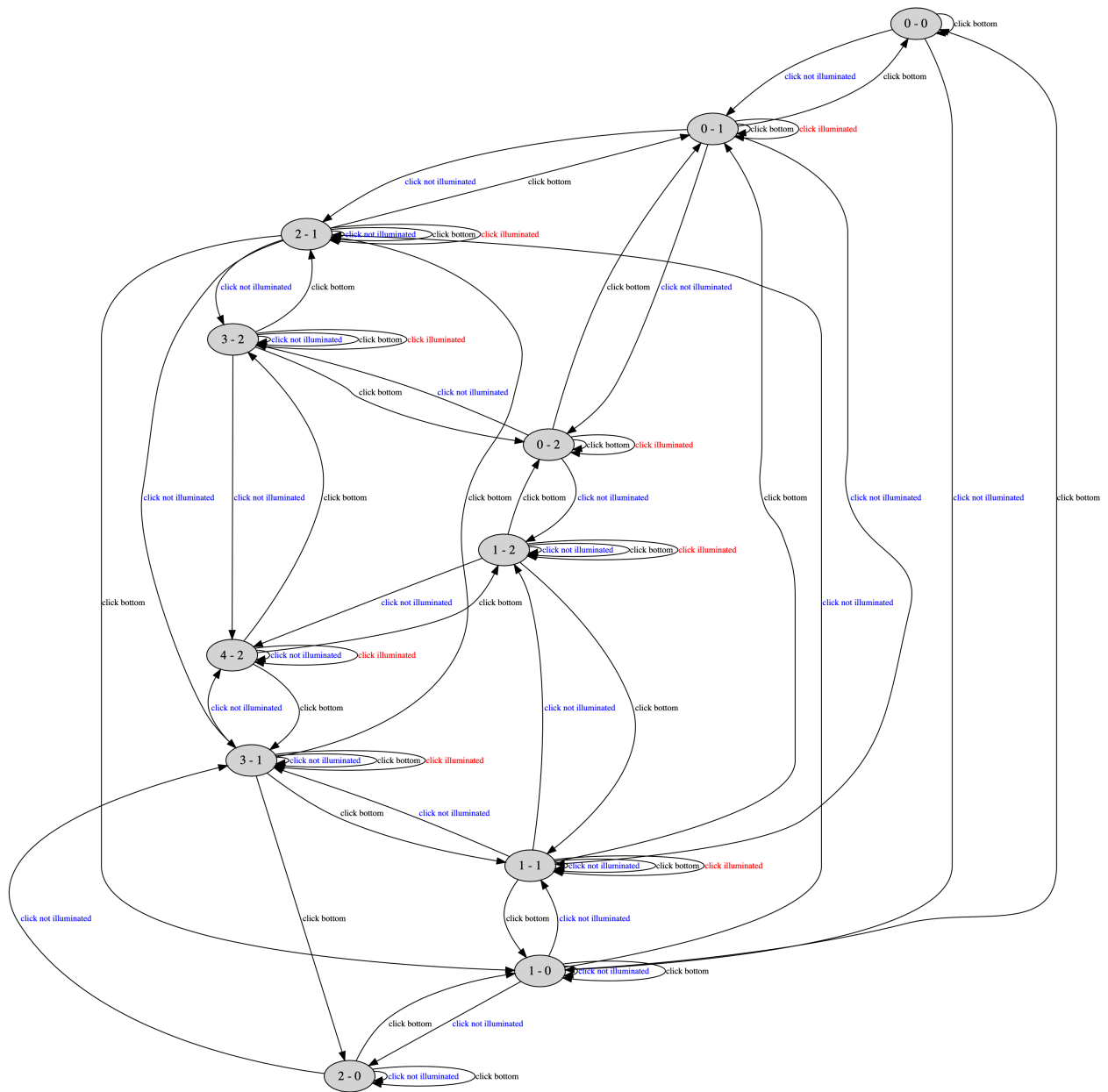


Figure 5.15: Modified pullback graph of the light control interface with a focus on feedback.

## 5.5 Discussion

Don Norman distinguishes four core principles of a good design: visibility, a good conceptual model, feedback, and good mappings [25]. Visibility ensures that users can quickly determine the device’s state and available actions. In our context, the lack of visible distinctions between states can create ambiguity in decision-making. We provided an example demonstrating how insufficient detail prevents users from distinguishing between states, leading to confusion in building conceptual models. Additionally, the previously defined action graph reflects the human perception of states, not the states encoded within the system. A good conceptual model provides users with an understanding of operations and results. The proposed pullback construction, its unambiguity directly indicates the particular effect of affordances on states. If a user achieves different states under different conditions with one affordance, it is challenging to build a good conceptual model. Good mappings establish clear relationships between actions, controls, and their effects. Our definition of pullback graph ambiguity reflects actions’ impact on the system states. Also, with different user profiles, it is possible to analyze mappings at different levels of perception. However, we did not include an analysis of the global effect of affordance on the state of the interface, focusing more on local mappings. Feedback provides users with information about the results of their actions. In the categorical framework, feedback consists of changing the context-state pairs. However, we do not encode supporting feedback, such as sound or textile sensations, which serves as confirmation of the successful interaction, but focus exclusively on changes in the encoded states.

In this chapter, we considered the issue of the ambiguity of the interface and the conditions of unambiguity within the framework. Unambiguity terms are subject to varying levels of detail and pullback modifications with a focus on user profiles. We described the sources of ambiguity and potential recommendations for correction. However, the next step is to provide an algorithm for ambiguity detection and analysis and changes to the graph to correct the ambiguity. Most assumptions about ambiguity are based on physical interfaces, with a small overview of virtual interfaces. In focusing on formal interfaces, we encounter the challenge of defining states, given the complexity and multi-component structure of these interfaces. We believe applicability is feasible if analysis is conducted at the component level, avoiding mixing different detail levels. For instance, evaluating a form’s performance is one level, while navigating across pages is another. Our conditions do not align with Esther Loeliger’s proposed approach [19], as her states are represented by links and event listeners on pages, which we view as perceived affordances that must be a part of the affordance graph.

# Chapter 6

## Generalizations

We introduced the general concept of using a universal construction from category theory to represent interfaces with labeled multigraphs. This chapter explores potential extensions of this theory to include a wider range of interfaces. We will present preliminary ideas and examples to illustrate these ideas.

### 6.1 Event-Driven Interfaces

In this section, we expand the framework for describing user interfaces to encompass a broader range of events beyond those triggered only by user actions. Previously, we focused on analyzing user experience and the direct outcomes of user-initiated actions, like clicking a button or following a URL, which causes specific functions to execute. However, user-driven events represent only a subset of what influences system behavior.

Events can also occur automatically, independent of user input. For instance, time progresses without user control or awareness, and its effects on the system are essential to consider. Many interfaces also run automated processes in the background, making it crucial to understand system behavior fully. To address this, we will generalize our graphs to include all types of events, not just those initiated by users.

In earlier sections, we focused exclusively on the interactive component of interfaces, assuming state changes were entirely under user control. Here, we generalize that scenario to account for state changes triggered by other means. The primary modification needed is the affordance graph. In the generalized case, the new version of the affordance graph will differ in the following way. The edges connecting contexts represent events that trigger transitions, including affordance triggers, but not only. We categorize events into three types: *affordance triggers*,

which are initiated by a user; *automatic triggers*, which are initiated by the system or software; and *hybrid triggers*, which can be initiated either by a user or the system. This classification allows for a more comprehensive understanding of how different types of events drive changes in the interface, accounting for both user-driven and automated system behaviors.

Some events may be accompanied by an affordance that *informs* the user of an action. Absence of supporting affordances can negatively affect the user experience. For example, I lived in a house with a thermostat that would sometimes ignore changes to the desired temperature and revert to the previous setting without any indication. This happened when the temperature difference was too large, but the lack of feedback was too frustrating.

### 6.1.1 Clock Example

Let us model a primary clock that displays the current time to the user and allows the user to set an alarm for a specific time. For simplicity, we can limit ourselves to only one alarm at a time, but generalizing to  $n$  alarms will not be difficult. The time display depends on the passage of time, which we cannot influence in any way.

**Affordance Graph.** The new affordance graph comprises the set of affordances available to the user, as initially described. The arrows representing changes in the context are generalized to cover all event-driven actions, both within and beyond user control. The affordance graph highlights two distinct contexts: a quiet context and a ringing context. Adding an alarm (a user action) does not immediately change the context. However, when the alarm is activated (an automatic action), the context changes, and the sound is heard. To return to the quiet context, the alarm can be stopped manually (within user control) or will revert automatically after a certain time (beyond user control). An illustration of this concept is depicted in the Figure 6.1.

**Action Graph.** The action graph describes the system states and the transitions between them, with specific conditions for each state transition depending on the current state. This graph serves as a high-level conceptual diagram rather than a detailed list of observed states. We highlight three main states: the clock state, which includes a displayed time  $t$  and an alarm setup. The clock state can have no alarm  $[]$ , a set alarm  $[a]$ , and a special case of an activated alarm  $[a!]$ . Our goal is to facilitate adding an alarm  $[a]$ , activating it when the set alarm time matches the current time  $t$ , stopping the alarm  $[a]$ , or postponing it. Since time  $t$  changes, each state includes self-loops. It's important to note that transitions do not necessarily lead to states with the same time because some actions take time and are not completed immediately. However, they always lead to states within the same category. This basic clock logic can be extended by adding more functionality and differentiating additional states based on the actions taken. Representation of the graph we have described is in the Figure 6.2.

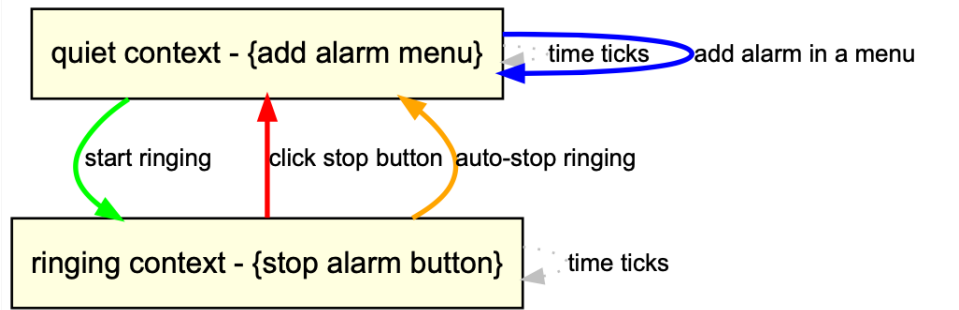


Figure 6.1: The affordance graph demonstrates the transitions between a quiet context and a ringing context. The quiet context allows adding an alarm, which does not change the state immediately. Transition to the ringing context occurs when the alarm starts, either automatically or manually. The ringing context can revert to quiet through manual stopping or automatic cessation. Time progression within each context is represented by self-loops.

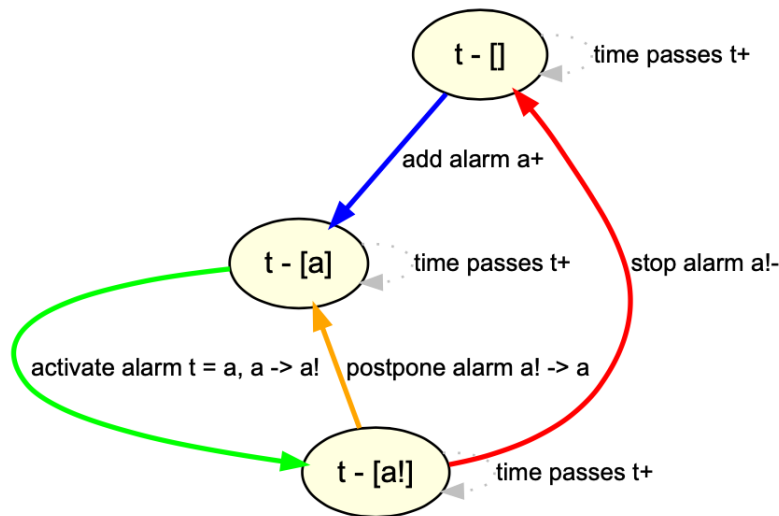


Figure 6.2: The action graph representing the system states and transitions. The graph highlights three main states: no alarm, set alarm, and activated alarm, with transitions based on actions like adding, activating, stopping, or postponing the alarm. Time changes are represented with self-loops, reflecting the evolving state of the clock logic.

**Abstract Graph.** We designed the abstract graph that is isomorphic to the affordance graph and emphasizes state dynamics without detailing specific affordances. This abstraction focuses on the essential state transitions without detailing the specific states, contexts, or transitions.

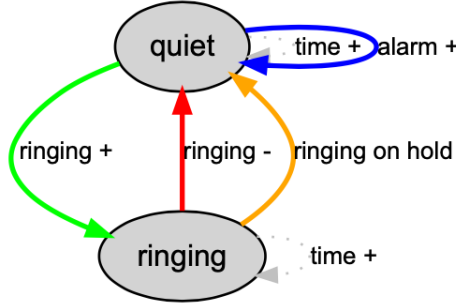


Figure 6.3: The abstract graph shows two main states: quiet and ringing. Transitions in the quiet state are triggered by time and alarms, while the ringing state handles time progression, stopping, and holding the alarm.

**Homomorphisms.** Homomorphisms between the action and affordance graphs are detailed in the tables below. The states of the clock without any alarm and with an inactive alarm are combined into the quiet context. In contrast, the state with an active alarm is associated with the ringing context, indicating that the activation of the alarm is accompanied by a ringing sound. Since the affordance graph captures all actions distinctly, each action-affordance pair is mapped to a corresponding element in the abstract graph.

| On Vertices                                         |                                        |
|-----------------------------------------------------|----------------------------------------|
| Source vertex $v$                                   | Target vertex $f^{vertex}(v)$          |
| $t - []$                                            | quiet                                  |
| $t - [a]$                                           | quiet                                  |
| $t - [a!]$                                          | ringing                                |
| On Labels                                           |                                        |
| Source edge label $\lambda$                         | Target edge label $f^{label}(\lambda)$ |
| add alarm $a+$                                      | alarm +                                |
| activate alarm $t = a, a \rightarrow a!$            | ringing +                              |
| stop alarm $a!-$                                    | ringing -                              |
| postpone alarm $a! \rightarrow a$                   | ringing on hold                        |
| time passes $t+$                                    | time +                                 |
| On Edges                                            |                                        |
| Mapping follows from vertex and edge label mappings |                                        |

Table 6.1: The homomorphism from the action graph to the abstract graph of the clock.

| On Vertices                                         |                                        |
|-----------------------------------------------------|----------------------------------------|
| Source vertex $v$                                   | Target vertex $f^{vertex}(v)$          |
| quiet context - {add alarm menu}                    | quiet                                  |
| ringing context - {stop alarm button}               | ringing                                |
| On Labels                                           |                                        |
| Source edge label $\lambda$                         | Target edge label $f^{label}(\lambda)$ |
| add alarm in a menu                                 | alarm +                                |
| start ringing                                       | ringing +                              |
| click stop button                                   | ringing -                              |
| auto-stop ringing                                   | ringing on hold                        |
| time ticks                                          | time +                                 |
| On Edges                                            |                                        |
| Mapping follows from vertex and edge label mappings |                                        |

Table 6.2: The homomorphism from the affordance graph to the abstract graph of the clock.

**Pullback Graph.** The generated pullback graph provides a template for the clock interface, encoding states and possible actions. Since all actions are distinctly defined, the graph is isomorphic to the action graph. The pullback graph is shown in the image below.

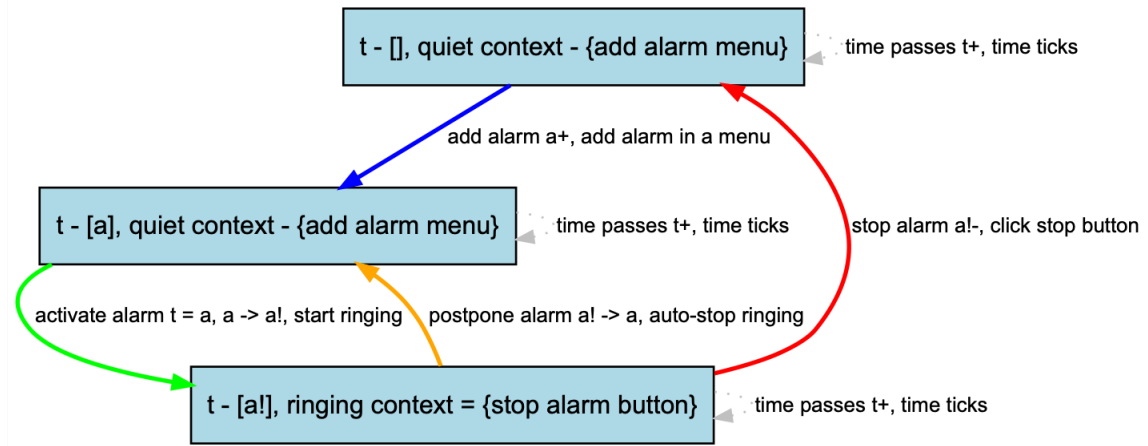


Figure 6.4: The pullback graph illustrates the relationship between three states: a quiet context without any active alarm, a quiet context with an active alarm, and a ringing context with a stop alarm button. Transitions between these states include adding an alarm, activating it, stopping it, and postponing it. The graph also shows self-loops for time progression in each state.

The pullback graph can become more complex with additional logic. For instance, we might include rules such as a clock that can be automatically postponed up to three times before stopping instead of postponing further. We could also account for multiple alarms, where stopping one alarm would only lead to an empty state if it was the only alarm; otherwise, the system would remain in a state with remaining alarms. The graph differentiates between automatic actions, which occur under certain conditions, and user-initiated actions. For example, adding an alarm is a user action (blue arrow), activation is an automatic action (green arrow), while deactivation can be either user-initiated (red arrow) or automatic after some time (orange arrow).

### 6.1.2 Event-Driven Interfaces as Pushouts

We discussed the generalization of the affordance graph into an event graph. However, this approach shifts the focus away from the user and the affordances provided to the user for interaction. A possibly more effective approach is distinguishing between actions the user controls and those outside their control. Let the pullback  $P$  represent a universal construction centered on the user, as this thesis explores. Let  $T$  be a graph representing state transitions not controlled by the user. Then, there exists a universal pushout construction representing a fully event-driven system, isomorphic to the defined pullback for a certain diagram.

**Proposition 22.** *Let  $P_E$  denote an event-driven pullback graph. The graph  $P_E$  is a pushout universal construction from the diagram  $P \xleftarrow{p} B \xrightarrow{t} T$ , where  $P$  represents the user interface graph modeling user-system interactions, and  $T$  represents state transitions outside the user's control. The graph  $B$  consists of state-context pairs from the user interface, with morphisms  $p$  and  $t$  defining the equivalence of states in graphs  $P$  and  $T$ .*

A key aspect of category theory is the duality of universal constructions. Abstractly, a pullback can be seen as a generalization of multiplication, while a pushout represents a generalized addition of objects. Previously, we observed that pushouts are well-suited for combining systems by defining common components. In the context of event-driven interactive systems, such systems combine user-controlled and non-controlled components. Hypothetically, we can imagine dividing the system into two parts and merging them. Given that, in directed multigraphs, the common features are states while the differences lie in the edges, the proposal is to merge them at the states, keeping the edges distinct. However, this implies that both graphs must contain information about states and contexts, even if only one graph directly relies on the contexts, to make the merging possible.

## 6.2 Multi-User Interfaces

Some interfaces are designed to involve more than one person interacting with the system. Previously, we briefly introduced puzzles as interfaces, and similarly, any game can be modeled as an interactive system. Notably, games inherently involve multiple players. Not only games require interactions among multiple users. For instance, many management systems involve several users with access, where changes one user makes impact on the interfaces of others. We understand it from the perspective of communication within a multi-user interactive system, where users' actions influence the shared interface. In this section, we provide an example of how an arbitrary pullback graph, representing an interactive system, can be generalized to support multiple users. Additionally, we discuss why these graphs can be understood as event-driven interfaces.

Let the pullback graph  $P$  represent a specific interactive system. Each graph component includes two parts: one derived from the action graph and the other from the affordance graph. Suppose we want to incorporate additional information about the player. Let  $n$  users interact with one another, and their interaction pattern is represented by a graph defined in the following way.

**Definition 6.2.1** (User Graph). Let  $U$  be a graph on  $n$  vertices. We define  $U$  as a *user graph*, where each vertex corresponds to an active player, and each edge indicates the users' move and the transition to the next active user. The label on the edge indicates the player initiating the move.

**Example 6.2.1.** For example, consider the simplest form of communication between two users. One user makes a move and passes the turn to the other. Both players interact with the same game interface, and the system state is shared between players. In this scenario, a simple graph  $U$  can represent the relationship between the players (or agents), where the nodes correspond to the players, and the arrows denote the actions that transition the turn from one player to another.

Now, we can define a graph that encodes both the interface and the players interaction patterns.

**Proposition 23.** Let  $P$  be a pullback above the diagram  $A \xrightarrow{f} M \xleftarrow{g} D$  representing an interacting system. Let  $U$  be a player graph. Let  $P_U$  be a multi-user pullback graph above a diagram  $P \xrightarrow{\phi} B \xleftarrow{\psi} U$  where  $B$  is a base, and  $\phi$  and  $\psi$  are graph morphisms.

The base graph is the core for connecting various graphs. Like an abstract graph that combines the action and affordance graphs, this base graph determines what states each player can access and which affordances they control. In the most straightforward case, all users can occupy any state, and since they share a common interface, the same affordances are accessible to everyone. In this scenario, the pullback graph is a product of two graphs.

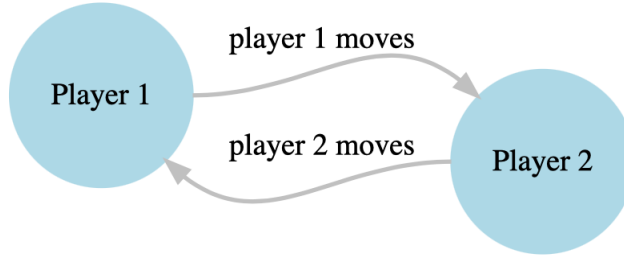


Figure 6.5: The user graph represents interaction between two users when they take turns.

The resulting pullback generally differs from the product universal construction in more complex scenarios, where the base graph governs dependencies and restrictions. The base graph reflects an additional structure that limits how states and affordances are shared among players, making the pullback more specific than the standard product.

Consider a multi-user interface represented by a pullback graph. If we fix one user and treat the actions of other users as external events, we can derive an event graph similar to the one described in the previous section.

**Proposition 24.** *Let  $P_U$  be a multi-user pullback graph above a diagram  $P \xrightarrow{\phi} B \xleftarrow{\psi} U$ . For  $P_U$  and a fixed user  $u$  there exists an event-driven pullback graph  $P_E$  where actions triggered by other users are events.*

This method allows for analyzing the interface from the perspective of a single user. Consequently, any theory or analysis technique applicable to event-driven graphs can extend to multi-user graphs. It can potentially allow the evaluation of the user’s experience of interacting with the interface when control is limited.

## 6.3 Discussion

Although event-driven interfaces were not the primary focus of this research, they are widespread in today’s automated world. This raises important questions: Can the techniques presented in this thesis be effectively applied to event-driven interfaces, or do they require further development and additional research? Moreover, what defines ambiguity in a generalized context, and how do users perceive automated state transitions? Addressing these questions calls for a deeper exploration of diverse examples and interfaces, along with a thorough analysis.

## Chapter 7

# Summary of Contributions and Future Directions

In this thesis, we developed a method within a categorical framework for studying user interfaces, implemented computational methods for constructing pullbacks and other universal objects in the category of directed labeled multigraphs and for verifying interface ambiguity conditions. We tested our assumptions on case studies, including two book reader interfaces, puzzles as interactive objects, three text-editing forms, a video recorder, a light control system, and a clock with alarm settings. We demonstrated how to represent interfaces as graphs, defining states and identifying affordances. Our approach emphasizes designing graphs that prioritize user perception of the interface, with flexibility for customization to suit different user profiles.

In Chapter 2 on category theory, we formulated and proved the validity of universal constructions within the category of directed labeled multigraphs. While existing literature provided constructions for other categories of graphs, there was no proof of the pullback universal construction, which is the main object of study in this thesis. Lastly, we developed software to compute the constructions proposed within the category of directed multigraphs.

In Chapter 3 on user interfaces graph representation, we described a representation of interactive systems from two distinct viewpoints. In this thesis, we developed and detailed a diagrammatic approach for constructing graphs that represent the interactive system as accurately as possible, supported by relevant examples. We addressed the issue of modes representing identical contexts but resulting in different interaction outcomes. We distinguished between explicit and implicit modes and explored how these can be formally represented within a categorical framework in an affordance graph. Lastly, we discussed the difference between physical and digital interfaces and provided strategies in the graphs construction.

In Chapter 4 on user profiles, we emphasized the importance of considering user profiles rather than generalizing for all groups. The level of perception of target users can vary significantly. As a result, we concluded that no universal representation of an interface can be captured solely by a pullback construction; instead, the construction must account for the user’s ability to interact with the interface. We proposed modifying the pullback graph to reflect different user profiles and outlined three key modification methods in this context, supported by examples.

In Chapter 5 on interfaces ambiguity, we studied the idea of ambiguity in user interfaces, where some actions lead to unexpected or counterintuitive state transitions. We chose a “local” strategy, focusing on affordance consistency within local states. We defined the conditions for unambiguity and defined criteria for a diagram that ensures unambiguity of the pullback, proving the equivalence of these conditions. We highlighted a class of interfaces where each state is accompanied by at most one context and analyzed the factor of ambiguity in the diagram for various cases, providing case studies of mock and real interfaces containing ambiguity and demonstrating how an interface can be unambiguous for one user profile but not for another.

In Chapter 6 on event-driven interfaces, we expanded the discussion to generalize graphs by focusing on user interaction without considering potential changes to the interface outside user control. We introduced event-driven graphs, where state transitions occur based on events, including user-driven events. We also demonstrated that event-driven graphs can be constructed using another universal construct, the pushout, which provides a framework for extending functionality by defining a common part in the objects. Furthermore, we discussed another form of extension: generalization to multi-user interfaces, where the actions of several users must be identified. We argued that such interfaces are event-driven, as a single user perceives actions by others as events beyond their control.

For the future development of this research, we will emphasize three main directions. First, while we analyzed a range of case studies, most interfaces were physical, where the distinction between states and affordances is clearer than in digital interfaces. Future research should focus on digital interfaces, including software for graph construction that incorporates parsers for specific interface domains. A potential challenge will be defining interface states in ways that align with user perceptions and tracking their changes based on affordances presented. Second, the concept of user profiles remains high-level at this stage, and developing concrete profiles requires further investigation, potentially involving studies with real participants to refine assumptions. Computationally implementing profiles as settings that adapt graph structures would be highly valuable for this research. Third, while our local approach to detecting interface ambiguity was effective for studies cases of user interfaces, we did not explore the global influence of each affordance across all states. Ideally, a control should consistently modify the same feature of any state. If consistent, it could impact the user’s understanding of the affordance’s function. Extending local ambiguity conditions to global ones is a possible direction for future research.

# References

- [1] Asaad Ahmed, Kamal Eldahshan, and Mahmoud Embabi. Conditional Activity-Based Model for Describing User Experience Using Category Theory. *International Journal of Computer Trends and Technology (IJCTT)*, 18, 12 2014.
- [2] A.F. Blackwell, C. Britton, A. Cox, T.R.G. Green, C. Gurr, G. Kadoda, M.S. Kutar, M. Loomes, C.L. Nehaniv, M. Petre, C. Roast, C. Roes, A. Wong, and R.M. Young. Cognitive Dimensions of Notations: Design Tools for Cognitive Technology. In *Cognitive Technology 2001 (LNAI 2117)*, pages 325–341. Springer, 2001.
- [3] Eugenia Cheng. *The Joy of Abstraction: An Exploration of Math, Category Theory, and Life*. Cambridge University Press, 2022.
- [4] Luca de Alfaro and Thomas A. Henzinger. Interface Automata. In *Proceedings of the 9th Annual Symposium on Foundations of Software Engineering*, pages 109–120. ACM, 2001.
- [5] Samuel Eilenberg and Saunders MacLane. General Theory of Natural Equivalences. *Transactions of the American Mathematical Society*, 58(2):231–294, 1945.
- [6] David England. A Category Theory Approach to HCI. In *Computing and Philosophy*, pages 73–84. IOS Press, 2007.
- [7] Gilles Fauconnier and Mark Turner. *The Way We Think: Conceptual Blending and the Mind’s Hidden Complexities*. Basic Books, 2002.
- [8] Krzysztof Z. Gajos, Jacob O. Wobbrock, and Daniel S. Weld. Improving the Performance of Motor-Impaired Users with Automatically-Generated, Ability-Based Interfaces. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI ’08, page 1257–1266, New York, NY, USA, 2008. Association for Computing Machinery.
- [9] James J. Gibson. *The Ecological Approach to Visual Perception: Classic Edition*. Houghton Mifflin, 1979.

- [10] Joseph Goguen. An Introduction to Algebraic Semiotics, with Application to User Interface Design. In Chrystopher L. Nehaniv, editor, *Computation for Metaphors, Analogy, and Agents*, volume 1562, pages 242–291. Springer Lecture Notes in Computer Science, 1999.
- [11] Joseph Goguen. What Is a Concept? In Frithjof Dau, Marie-Laure Mugnier, and Gerd Stumme, editors, *Conceptual Structures: Common Semantics for Sharing Knowledge*, pages 52–77, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.
- [12] Joseph Goguen and Fox Harrell. Value-Driven Design with Algebraic Semiotics. In *First International Workshop on Value-Based Software Engineering*. Springer, 2004.
- [13] Joseph A. Goguen. Algebraic Semiotics: A New Perspective on the Relationships Between Semiotics and Information Systems. *Proceedings of the 1999 International Conference on Information Systems*, pages 82–90, 1999.
- [14] Ronald Harris. *Semiotic Perspectives in Psychology*. Psychology Press, 1996.
- [15] Muhammad Nazrul Islam and Harry Bouwman. Towards User-Intuitive Web Interface Sign Design and Evaluation: A Semiotic Framework. *International Journal of Human-Computer Studies*, 86:121–137, 2016.
- [16] Muhammad Nazrul Islam, Harry Bouwman, and A. K. M. Najmul Islam. Evaluating Web and Mobile User Interfaces With Semiotics: An Empirical Study. *IEEE Access*, 8:84396–84414, 2020.
- [17] Wei Jia, Qingyi Hua, Minjun Zhang, Bo Wang, Rui Chen, and Xiang Ji. Category Theory-Based Mobile User Interface Pattern Recommendation Method. *IEEE Access*, 2018.
- [18] Slobodan Kovacevic. UML and User Interface Modeling. In Jean Bézivin and Pierre-Alain Muller, editors, *The Unified Modeling Language: UML’98 Beyond the Notation*, volume 1619 of *Lecture Notes in Computer Science*, pages 20–30. Springer, 1999.
- [19] Esther-Anna Loeliger and Chrystopher L. Nehaniv. Affordance Graphs for User Interface Design: Application of Category-theoretic Constructs. In *Proceedings of the 13th International Conference on Automata and Formal Languages*, pages 392–394, 2011.
- [20] Grant Malcolm and Joseph A. Goguen. Signs and Representations: Semiotics for User Interface Design. In Ray Paton and Irene Neilson, editors, *Visual Representations and Interpretations*, pages 363–384. Springer-Verlag London, London, 1999.
- [21] Mark Minas and Hans Jürgen Schneider. *Graph Transformation by Computational Category Theory*, pages 33–58. Springer, 2010.

- [22] Yaniv Mordecai and Avner Engel. Multimodal System Design Integration Using Category Theory. In *Proceedings of the 2nd International Workshop on Model-Driven Requirements Engineering*, pages 47–56. IEEE, 2014.
- [23] Chrystopher Nehaniv, Fariba Karimi, Daniel Schreckling, Nicoline den Breems, Ágnes Bonivárt, Maria Schilstra, Alastair Munro, and Attila Egri-Nagy. *BIOMICS Deliverable D2.1 Category Theoretic Framework: Functors and Adjoints for Discrete and Continuous Dynamical Cellular Systems Symmetries*. European Commission, March 2014.
- [24] Donald Norman. Affordance, Conventions, and Design. *The Nielsen Norman Group*, 1999.
- [25] Donald Norman. *The Design of Everyday Things*. Basic Books, New York, NY, revised and expanded edition, 2013.
- [26] Philippe A. Palanque, Rémi Bastide, Louis Dourte, and Christophe Sibertin-Blanc. Design of user-driven interfaces using Petri nets and objects. In *Proceedings of the IFIP TC2/WG2.7 Working Conference on Engineering for Human-Computer Interaction*, pages 203–220. North-Holland Publishing Co., 1993.
- [27] T. L. Riché, R. Gonçalves, B. Marker, and D. Batory. Pushouts in Software Architecture Design. In *2015 12th Working IEEE/IFIP Conference on Software Architecture*, pages 51–60. IEEE, 2015.
- [28] Simone Santini. Notes for the Conceptual Design of Interfaces. In David W. Embley, Antoni Olivé, and Sudha Ram, editors, *Conceptual Modeling - ER 2006*, pages 413–423, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.
- [29] Marco Schorlemmer and Enric Plaza. A Uniform Model of Computational Conceptual Blending. *Cognitive Systems Research*, 65:118–137, 2021.
- [30] Kamran Sedig and Paul Parsons. Interaction Design for Complex Cognitive Activities with Visual Representations: A Pattern-Based Approach. *ACM Transactions on Computer-Human Interaction (TOCHI)*, 20(5):26, 2013.
- [31] Ben Shneiderman and Catherine Plaisant. *Designing the User Interface: Strategies for Effective Human-Computer Interaction*. Pearson/Addison Wesley, 2004.
- [32] Paulo Pinheiro da Silva and Norman W. Paton. User Interface Modelling with UML. In *Proceedings of the 5th International Conference on the UML*, pages 51–65. Springer, 2001.
- [33] L. Zouhaier, Y. Ben Daly Hlaoui, and L.B. Ayed. Adaptive User Interface Based on Accessibility Context. *Multimedia Tools and Applications*, 82:35621–35650, 2023.

# APPENDICES

# Appendix A

## Universal Constructions Implementation

We developed algorithms for constructing universal constructs with the C# programming language in the .NET platform (can be accessed here <https://github.com/mzheplin/Categories>). Architecture of the project aligns with category theoretical levels of abstrac-

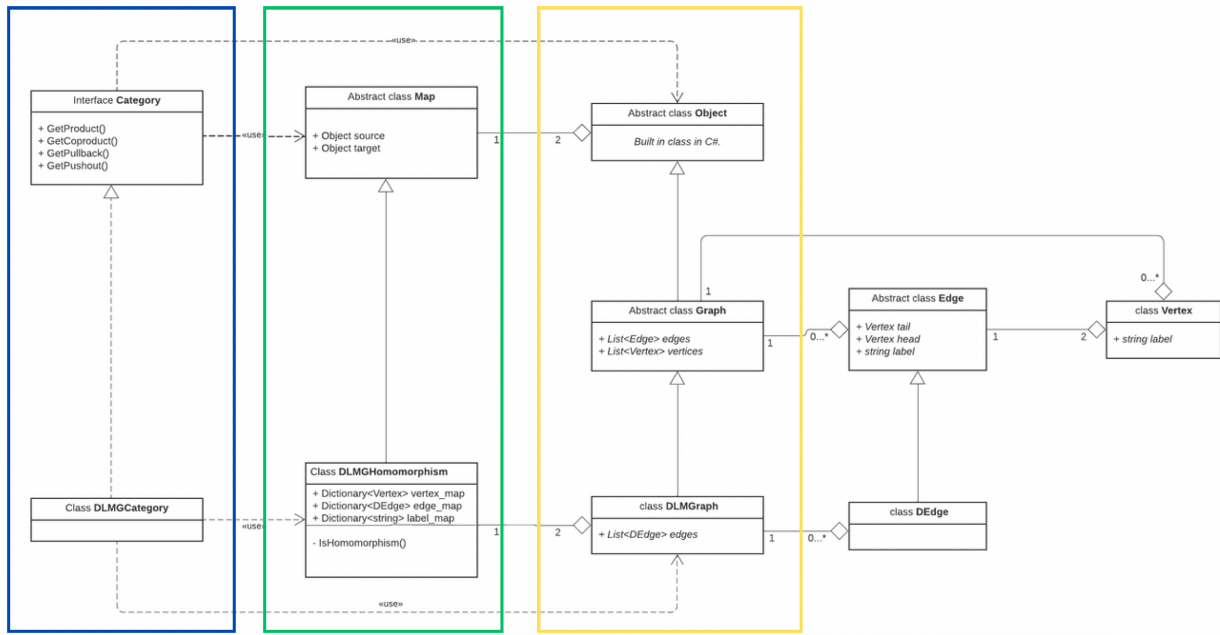


Figure A.1: UML diagram of the DLMG project.

tion - category, map, and objects. Implementations of this abstract classes *DLMGCategory*, *DLMGHomomorphism*, *DLMGGraph* classes represent a concrete implementation of the category of directed labeled multigraphs.

**DLMG Homomorphism.** The implementation of homomorphisms between directed labeled multigraphs is based on the pseudocode provided below. The method consists of three stages: verification of the mapping on vertices, followed by verification on labels, and finally, verification on edges, completed if the first two stages are successful.

---

**Algorithm 1:** Pseudocode for checking a graph homomorphism.

---

```

1 function CheckHomomorphism (source, target, v_map, l_map);
   Input : source and target are directed labelled multigraphs, v_map and l_map are vertex and
           label maps respectively, represented as lists of tuples.
   Output: boolean value.
2 foreach vertex in source do
3   if v_map doesn't contain any tuple (vertex,w), where w is in the target then
4   |   return false;
5   else if v_map contains more than one tuple (vertex,w), where w is in the target then
6   |   return false;
7   else if v_map contains a tuple (vertex,w), where w is NOT the in target then
8   |   return false;
9 end
10 foreach edge label in source do
11   if l_map doesn't contain any tuple (label,w), where w is in the target then
12   |   return false;
13   else if l_map contains more than one tuple (label,w), where w is in the target then
14   |   return false;
15   else if l_map contains a tuple (label,w), where w is NOT the in target then
16   |   return false;
17 end
18 foreach edge in source do
19   newTail  $\leftarrow$  v_map(edge.tail) ;
20   newHead  $\leftarrow$  v_map(edge.head) ;
21   newLabel  $\leftarrow$  l_map(edge.label) ;
22   newEdge  $\leftarrow$  new edge with newTail, newHead, newLabel;
23   if target does not contain newEdge then
24   |   return false;
25   end
26 end
27 return true;

```

---

**Coproduct.** The coproduct in the category of directed labeled multigraphs corresponds to the disjoint union of two graphs. The following pseudocode outlines the implementation of a function that computes the coproduct of directed labeled multigraphs, with morphisms  $f$  and  $g$ .

---

**Algorithm 2:** Algorithm for finding a coproduct of two directed labelled multigraphs.

---

```

1 function Coproduct (graph_a, graph_b);
   Input : Two directed labelled multigraphs graph_a and graph_b
   Output: triple (coproduct,  $f$ ,  $g$ )
2 Let coproduct be a directed labelled multigraph;
3 Let  $f$  be a homomorphism from graph_a to graph coproduct;
4 Let  $g$  be a homomorphism from graph_b to graph coproduct;
5 foreach vertex in graph_a do
6   |  $newLabel \leftarrow vertex\ label + " - 0"$ ;
7   |  $newVertex \leftarrow vertex\ with\ newLabel$ ;
8   | add tuple (vertex, newVertex) to  $f$ ;
9   | add newVertex to coproduct.
10 end
11 foreach vertex in graph_b do
12   |  $newLabel \leftarrow vertex\ label + " - 1"$ ;
13   |  $newVertex \leftarrow vertex\ with\ newLabel$ ;
14   | add tuple (vertex, newVertex) to  $g$ ;
15   | add newVertex to coproduct.
16 end
17 foreach edge in graph_a do
18   |  $newLabel \leftarrow edge\ label + " - 0"$ ;
19   |  $newTail \leftarrow f(edge\ tail)$ ;
20   |  $newHead \leftarrow f(edge\ head)$ ;
21   |  $newEdge \leftarrow edge\ with\ newLabel, newTail, newHead$ ;
22   | add tuple (edge label, newLabel) to  $f$ ;
23   | add newEdge to coproduct.
24 end
25 foreach edge in graph_b do
26   |  $newLabel \leftarrow edge\ label + " - 1"$ ;
27   |  $newTail \leftarrow f(edge\ tail)$ ;
28   |  $newHead \leftarrow f(edge\ head)$ ;
29   |  $newEdge \leftarrow edge\ with\ newLabel, newTail, newHead$ ;
30   | add tuple (edge label, newLabel) to  $g$ ;
31   | add newEdge to coproduct.
32 end

```

---

**Product.** The product in the *DLMG* category is distinct from the Cartesian product. An edge  $e = (e_1, e_2)$  with  $\sigma(e) = (\sigma(e_a, \sigma(e_2)), \tau(e) = (\tau(e_a, \tau(e_2)), \lambda(e) = (\lambda(e_a, \lambda(e_2)))$  is present in the product graph if an edge  $e_1$  exists in the first graph and an edge  $e_2$  exists in the second graph. Below is the pseudocode for implementing a function that computes the product of two directed labeled multigraphs, along with the two morphisms to the individual graphs.

---

**Algorithm 3:** Algorithm for finding a product of two directed labelled multigraphs.

---

```

1  function Product (graph_a, graph_b);
   Input : Two directed labelled multigraphs graph_a and graph_b
   Output: triple (product, f, g)
2  Let product be a directed labelled multigraph;
3  Let f be a homomorphism from product to graph_a;
4  Let g be a homomorphism from product to graph_b;
5  foreach vertex a in graph_a do
6      foreach vertex b in graph_b do
7          newVertex  $\leftarrow$  (vertex a, vertex b);
8          add tuple (newVertex, a) to f;
9          add tuple (newVertex, b) to g;
10         add newVertex to product.
11     end
12 end
13 foreach edge a in graph_a do
14     foreach edge b in graph_b do
15         newLabel  $\leftarrow$  (edge a label , edge b label);
16         newTail  $\leftarrow$  (edge a tail , edge b tail);
17         newHead  $\leftarrow$  (edge a head , edge b head);
18         newEdge  $\leftarrow$  edge with newLabel, newTail, newHead;
19         add tuple (newEdge, a) to f;
20         add tuple (newEdge, b) to g;
21         add newEdge to product.
22     end
23 end

```

---

**Pullback.** The pullback over the diagram  $A \xrightarrow{f} C \xleftarrow{g} B$  forms a subgraph of the product of graphs  $A$  and  $B$ . The pseudocode for the function that computes the pullback given diagram is provided below. The difference with the pseudo-code of the product is in lines 12 and 22, where addition occurs only under the condition of equality of morphisms.

---

**Algorithm 4:** Algorithm for finding a pullback of two directed labelled multigraphs.

---

```
1 function Pullback ( $s, t$ );  
   Input : Two homomorphisms  $s$  and  $t$  of directed labelled multigraphs  
   Output: triple ( $pullback, f, g$ )  
2 if if target of  $s$  and  $t$  are different then  
3   | return null;  
4 end  
5  $graph\_a \leftarrow$  source of  $s$ ;  
6  $graph\_b \leftarrow$  source of  $t$ ;  
7 Let  $pullback$  be a directed labelled multigraph;  
8 Let  $f$  be a homomorphism from  $pullback$  to  $graph\_a$ ;  
9 Let  $g$  be a homomorphism from  $pullback$  to  $graph\_b$ ;  
10 foreach vertex  $a$  in  $graph\_a$  do  
11   | foreach vertex  $b$  in  $graph\_b$  do  
12     | if  $s(a)$  and  $t(b)$  are the same then  
13       |  $newVertex \leftarrow$  (vertex  $a$ , vertex  $b$ );  
14       | add tuple ( $newVertex, a$ ) to  $f$ ;  
15       | add tuple ( $newVertex, b$ ) to  $g$ ;  
16       | add  $newVertex$  to  $product$ .  
17     | end  
18   | end  
19 end  
20 foreach edge  $a$  in  $graph\_a$  do  
21   | foreach edge  $b$  in  $graph\_b$  do  
22     | if  $s(a)$  and  $t(b)$  are the same then  
23       |  $newLabel \leftarrow$  (edge  $a$  label , edge  $b$  label);  
24       |  $newTail \leftarrow$  (edge  $a$  tail , edge  $b$  tail);  
25       |  $newHead \leftarrow$  (edge  $a$  head , edge  $b$  head);  
26       |  $newEdge \leftarrow$  edge with  $newLabel, newTail, newHead$ ;  
27       | add tuple ( $newEdge, a$ ) to  $f$ ;  
28       | add tuple ( $newEdge, b$ ) to  $g$ ;  
29       | add  $newEdge$  to  $product$ .  
30     | end  
31   | end  
32 end
```

---

**Pushout.** Below is the pseudocode of function that finds a pushout given diagram. Pushout construction involves the generation of equivalence classes for vertices and edge labels. An edge

exists between such equivalence classes that contain edge vertices from the source graph. Below, we provide pseudocode 5 for generating equivalence classes of vertices and edge labels, and pseudocode for calculating a pushout given diagram 6.

---

**Algorithm 5:** Algorithm for finding a vertex equivalence classes  $v\_classes$ .

---

```

1  function Generate_classes ( $k, e$ );
   Input : Two homomorphisms  $k : c \rightarrow a$  and  $e : c \rightarrow b$  of directed labelled multigraphs.
   Output:  $v\_classes$ 
2  if if source or target of  $k$  and  $e$  are different then
3    | return null;
4  end
5   $graph\_c \leftarrow$  source of  $k$ (or  $e$ );
6   $graph\_a \leftarrow$  target of  $k$ ;
7   $graph\_b \leftarrow$  target of  $e$ ;
8  foreach vertex in  $graph\_c$  do
9    |  $a \leftarrow k(vertex)$ ;
10   |  $b \leftarrow e(vertex)$ ;
11   | if  $a$  is in some equivalence class  $[a']$  then
12     | add  $b$  to  $[a']$ ;
13     | go to the next vertex;
14   | end
15   | if  $b$  is in some equivalence class  $[b']$  then
16     | add  $a$  to  $[b']$ ;
17     | go to the next vertex;
18   | end
19   | create a new equivalence class  $[a]$  (or  $[b]$ );
20   | add  $a$  and  $b$  to the equivalence class  $[a]$  (or  $[b]$ );
21   | add  $[a]$  to  $v\_classes$  ;
22 end
23 foreach vertex  $v$  in  $graph\_a \sqcup graph\_b$  do
24   | if  $v$  is in some equivalence class then
25     | go to the next vertex;
26   | else
27     | create a new equivalence class  $[v]$ ;
28     | add  $vertex$  to the equivalence class  $[v]$  and make it a representative;
29     | add  $[v]$  to  $v\_classes$  ;
30   | end
31 end

```

---

---

**Algorithm 6:** Algorithm for calculating a pushout given diagram.

---

```

1  function Pushout ( $s, t$ );
   Input : Two homomorphisms  $s$  and  $t$  of directed labelled multigraphs
   Output: triple ( $pushout, f, g$ )
2  if source of  $s$  and  $t$  are different then
3  |   return null;
4  end
5   $graph\_a \leftarrow$  target of  $s$ ;
6   $graph\_b \leftarrow$  target of  $t$ ;
7  Let  $pushout$  be a directed labelled multigraph;
8  Let  $f$  be a homomorphism from  $graph\_a$  to  $pushout$ ;
9  Let  $g$  be a homomorphism from  $graph\_b$  to  $pushout$ ;
10 (1) Generate vertex equivalence classes  $v\_classes$  from  $s^{vertex}$  and  $e^{vertex}$ ;
11 (2) Generate label equivalence classes  $l\_classes$  from  $s^{label}$  and  $t^{label}$ ;
12 foreach vertex  $v$  in  $graph\_a \sqcup graph\_b$  do
13 |   Find equivalence class  $[v]$  in  $v\_classes$ ;
14 |   Add  $[v]$  to vertices of  $pushout$ ;
15 |   if  $v$  is from  $graph\_a$  then
16 |   |   Add tuple  $(v, [v])$  to homomorphism  $f$ ;
17 |   else
18 |   |   Add tuple  $(v, [v])$  to homomorphism  $g$ ;
19 |   end
20 end
21 foreach edge  $e$  in  $graph\_a \sqcup graph\_b$  do
22 |   Find equivalence class  $[label(e)]$  in  $l\_classes$ ;
23 |   Find equivalence class  $[head(e)]$  in  $v\_classes$ ;
24 |   Find equivalence class  $[tail(e)]$  in  $v\_classes$ ;
25 |   Add  $[label(e)]$  to labels of  $pushout$ ;
26 |   Add  $[e]$  to edges of  $pushout$ ;
27 |   if  $e$  is from  $graph\_a$  then
28 |   |   Add tuple  $(label(e), [label(e)])$  to homomorphism  $f$ ;
29 |   |   Add tuple  $(e, [e])$  to homomorphism  $f$ ;
30 |   else
31 |   |   Add tuple  $(label(e), [label(e)])$  to homomorphism  $g$ ;
32 |   |   Add tuple  $(e, [e])$  to homomorphism  $g$ ;
33 |   end
34 end

```

---