ABSTRACT

Title of Dissertation: SEQUENTIAL, HIERARCHICAL, AND ANALOGICAL PLAN TRANSFER IN ROBOTICS

Angeline Aguinaldo
Doctor of Philosophy, 2025

Dissertation Directed by: Professor William Regli
Department of Computer Science

As robotic systems encounter increasingly complex domains, efficient plan transfer—within and across distinct planning contexts—becomes essential to achieving adaptability and operational efficiency. This dissertation formalizes task-level plan transfer by introducing a conceptual and mathematical framework based on category theory, defining three types of transfer: sequential, hierarchical, and analogical plan transfer. For sequential and hierarchical transfers, symmetric monoidal categories (SMCs) and string diagrams are used to provide a rigorous framework for coherent task composition. To enable analogical transfers, a new planning representation language that integrates structured knowledge through C-sets and double-pushout (DPO) rewriting is introduced. Functorial data migrations are then used to align and preserve semantic structure during analogical plan transfer, circumventing the need to re-plan. The efficacy of this framework is demonstrated through case studies across several applications, from industrial automation to service robotics, illustrating the broad applicability of these methods. In doing so, this work contributes to the growing body of research demonstrating how category theory unifies heterogeneous representations across computer science. Ultimately, this work advances AI planning and robotics by establishing a principled foundation for task-level plan transfer that enhances the safety, flexibility, and interpretability of robotic task planning in complex and knowledge-rich environments.

SEQUENTIAL, HIERARCHICAL, AND ANALOGICAL PLAN TRANSFER IN ROBOTICS

by

Angeline Aguinaldo

Dissertation submitted to the Faculty of the Graduate School of the
University of Maryland, College Park in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
2025

Advisory Committee:
Professor William Regli, Chair/Advisor
Professor Huan Xu, Dean's Representative
Professor John Yiannis Aloimonos
Professor Dinesh Manocha
Professor Emeritus Dana Nau
Professor Mark Fuge, ETH Zurich
Professor Eswaran Subrahmanian, Carnegie Mellon University

© Copyright by
Angeline Aguinaldo
2025

Dedication

To my loving husband, Alex, and the future we hold together.

Acknowledgments

With great pride, I say that this dissertation was advised by William (Bill) Regli. When I chose this path, I did so with the hope of becoming a skilled researcher and communicator—someone who could take (just one) pail to the ocean and boil it so that others could admire its contents. I had hoped to do all this in an environment that was challenging but encouraging: a place where I could develop my independence, be brave, and befriend scholars across the globe. It just so happens that I got that experience thanks to the guidance, mentorship, and example of Bill. This has been a lovely pairing that has only been enriched by all the roles and experiences had over the years. Thank you for your unwavering presence throughout this journey and for the genius that sparked this work.

To my committee—Dana Nau, John Yiannis Aloimonos, Eswaran (Sub) Subrahmanian, Dinesh Manocha, Huan (Mumu) Xu, and Mark Fuge. I have watched you work as professors and researchers and have endowed great respect. I have witnessed you share remarkable ideas with confidence, tact, and experience. It should come to no surprise that I sought out each of you as coveted audience members for this work. Thank you for supporting me in sharing my own ideas and for living out careers worthy of aspiration.

Before this endeavor, I believed that a small, elite group was granted the gifts and talents to make scientific discoveries and shape the future: making my journey a test that would decide if I was among them. However, one of the most important personal discoveries I've made is that a good scientist is differentiated by their willingness to engage in dialogue with others. It is now clear to me that what drives scientific progress are people, their choices, and with whom they share those choices with. I learned that every choice I made—whether it was a picture I drew on a whiteboard, the word I used in an email, or the decision to cross the ocean to be amidst a group of strangers—shaped the way I and others thought in meaningful ways. I learned that a grand contribution can begin with a simple 'hello.' So, I wanted to take a moment to thank everyone who went out of their way to say 'hello' to me and, in turn, shape this work.

To Spencer Breiner, Sub, Brendan Fong, and David Spivak—had it not been for the early discussions, tutorials, and invitation to join the 2019 Special Session on Applied Category Theory AMS meeting in Riverside, CA (courtesy of Spencer and Sub), I am not really sure how my research would have progressed. Because of this invitation, I found myself traveling alone to a part of the country I had never seen, sitting among strangers to learn in a language to which I was a novice. David and Brendan, your talks at that meeting disarmed me. As with anyone who encounters a new mathematical language, I was intimidated by having to decrypt the foreign sounds and rationale. Yet, your talks did not lead the audience through this territory with arrogance, but with humility. Brendan, since then, we have developed a wonderful friendship, initiated and strengthened by your continued invitation to the Adjoint School and other activities within the research community. I am grateful for these opportunities, as well as for the incredible strides the Topos Institute made in its own journey under your leadership while I was making strides in mine. Together, these experiences allowed me to find a research home in Berkeley, CA.

To Evan Patterson and James Fairbanks—you may not have realized that when we sat together in that Starbucks in Riverside in 2019, in my mind, you were giants. While giants you are, I'm grateful to now be able to call you both my friends and mentors. You welcomed me into

the AlgebraicJulia team on many occasions and that has meant so much to me. I recall during the early days of AlgebraicJulia (then Catlab), James and I were both visiting NIST. James was preparing for a meeting and asked if I wanted to sit in. *"It's a Catlab meeting, but usually it's just me and Evan,"* James remarked. Hindsight has allowed me to relish the fortune of having witnessed the beginning of a flourishing project.

I owe a large part of this success to my many other applied category theory collaborators and friends—Kristopher Brown, Kevin Arlin, Owen Lynch, Priyaa Varshinee Srinivasan, Joseph Moeller, Sophie Libkind, David Jaz Myers, Elena Di Lavore, Nathan Haydon, Ana Luiza da Conceição Tenorio, Daniel Cicala, Sean Wu, Micah Halter, Jacob Zelko, Paul Dancstep, Nelson Niu, Ian Levitt, Michael Lambert, Blake Pollard, Zinovy Diskin, Christian Williams, Valeria de Paiva, Vadim Shapiro, John Baez, and many more. Thank you for your warm greetings and willingness to trudge through the dialogue that often comes with sharing deep ideas with others. I also want to express gratitude for the opportunities I had to collaborate with individuals from Siemens, University of Florida, and around the University of Maryland—Arquimedes Martinez Canedo, Gustavo Quiros Araya, Jaime Ruiz, John Kanu, Jacob Bunker, Ankit Shukla, and Lazlo Nemeth.

I will never forget to thank the people at the Johns Hopkins University Applied Physics Laboratory (APL) who supported me relentlessly by cultivating a professional environment grounded in excellence and care. APL is where I developed and refined the skills I take great pride in: applied research, oratory, leadership, and connecting with others. In particular, I wanted to acknowledge the many people that had a hand in helping me accomplish this goal with ease—Pedro Rodriguez, Michael Castle, John Piorkowski, Gina Marshall-Johnson, Elisha Peterson, Emily Brown, Matthew Dinmore, Rob Nicholas, Bill Blackert, Chris Gifford, Jonathan Cohen, Samuel Matos, Alison Albin, Beatrice Garcia, Patrick Lee, Vikram Saraph, Gene Whipps, Charles Leeper, Ryan Amundsen, Cathy Chen-Kieffer, Lisbeth Campbell, Cynthia Collins, and many more friends and colleagues.

To Andrew Cohen and alumni of the Bioimage lab—my foray into academic research began at the Computational Image Sequence Analysis lab, or as affectionately called the Bioimage lab, at Drexel University. It was here that my scholarly voice sprouted. *'Can I show you something?'* was the simple, but powerful question that echoed through the lab daily. These words were triggered by successes, big and small. Since then, I have made a habit of using these words to acknowledge the greatness in my own work while courageously inviting the perspective of others. This lesson, among the many others, has unlocked gates in my professional and academic journey. Because of this, I would like to thank you, Mark Winter, Eric Wait, Edgar Cardenas, Dhantha Gunarathna, Bingyao Huang, Gjergji Konica, and Walter Mankowski for setting this example for me.

To my family—Mommy, Daddy, and Kristene—in whose honor I have chosen to immortalize our family name through this piece. My love for you exceeds the bounds of time and space.

To Alex and our special others, to whom this whole work is dedicated to. Thank you for your love that serves as the bookends of this journey. We have accomplished so much together and I am delighted to say this can be added to our list.

To my dearest friend of all, who was there in the beginning and will be there in the end: the Alpha and the Omega. It is to You that I owe this life.

Table of Contents

Dedication ii

Acknowledgements iii

Table of Contents v

List of Tables ix

List of Figures x

Chapter 1: Introduction 1
- 1.1 Motivation . . . 1
- 1.2 Research Questions . . . 3
- 1.3 Task Plan Transfer . . . 4
  - 1.3.1 The Conceptual Framework . . . 4
  - 1.3.2 The Mathematical Framework . . . 6
- 1.4 Summary of Research Contributions . . . 7
- 1.5 Structure of the Dissertation . . . 8

Chapter 2: Background 10
- 2.1 Introduction . . . 10
- 2.2 Sequential Transfer in Early Robot Programming . . . 11
  - 2.2.1 Motion Primitives and Skills . . . 11
  - 2.2.2 In AI Planning . . . 13
  - 2.2.3 Validating Sequential Transfer . . . 15
- 2.3 Hierarchical Transfer in Robotic Software Engineering . . . 16
  - 2.3.1 Modular Architectures . . . 16
  - 2.3.2 Hierarchical Task Networks (HTNs) . . . 17
  - 2.3.3 Validating Hierarchical Transfer . . . 19
- 2.4 Analogical Transfer in AI Planning and Transfer Learning . . . 19
  - 2.4.1 Formal Theory of Analogy . . . 20
  - 2.4.2 Using Program Translation . . . 21
  - 2.4.3 Using Transfer Learning . . . 22
  - 2.4.4 Using Case-based Planning . . . 23
  - 2.4.5 Using Ontology Alignment . . . 25
  - 2.4.6 Validating Analogical Transfer . . . 26
- 2.5 Summary . . . 27

Chapter 3: Category Theory Foundations 28

3.1 Categories, Functors, Natural Transformations . . . 30
3.1.1 Some Notable Functors and Natural Transformations . . . 34
3.2 Universal Properties . . . 37
3.3 Constructions . . . 43
3.4 Other Categories . . . 44
3.5 Applications of Category Theory . . . 47
3.5.1 String Diagrams . . . 47
3.5.2 Categorical Databases . . . 48
3.6 Summary . . . 51

Chapter 4: Sequential Transfer using String Diagrams 52
4.1 Overview . . . 53
4.2 The Planning Domain Definition Language (PDDL) . . . 54
4.3 String Diagrams . . . 55
4.4 Symmetric Monoidal Category, PDDLPlan . . . 58
4.4.1 Building a String Diagram in PDDLPlan . . . 60
4.5 Example: Blocksworld . . . 63
4.5.1 Domain . . . 63
4.5.2 Problem . . . 64
4.6 Discussion . . . 68
4.6.1 Advantages of the Categorical Structure for Sequential Transfer . . . 68
4.6.2 Limitations of the Categorical Structure for Sequential Transfer . . . 70
4.7 Conclusion . . . 71

Chapter 5: Hierarchical Transfer using Functors 72
5.1 Overview . . . 73
5.2 The RoboCat Framework . . . 74
5.3 Case Study: Machine-Tending Problem . . . 76
5.4 Modules using String Diagrams . . . 77
5.4.1 Functors Between String Diagrams . . . 77
5.5 Categorical Framework . . . 78
5.5.1 The Category of the Symbolic Work Cell, Task . . . 79
5.5.2 The Category of the Virtual Work Cell, Virtual . . . 80
5.5.3 The Category of the Motion Primitives, Motion . . . 82
5.5.4 $F : \mathsf{Task} \to \mathsf{Virtual}$ . . . 84
5.5.5 $G : \mathsf{Virtual} \to \mathsf{Motion}$ . . . 86
5.6 Application of the RoboCat Framework to the Machine-Tending Problem . . . 88
5.7 Discussion . . . 93
5.7.1 Advantages of the Categorical Structure for Hierarchical Transfer . . . 93
5.7.2 Limitations of the Categorical Structure for Hierarchical Transfer . . . 94
5.7.3 Exploring Future Applications of String Diagrams . . . 95
5.8 Conclusion . . . 96

Chapter 6: Structured Knowledge in Planning 97
6.1 Motivation . . . 97
6.2 Related Work . . . 99
6.2.1 Typing and Derived Predicates in PDDL . . . 100
6.2.2 Ontologies and Semantic Graphs . . . 102

6.3 Background . . . 104
6.3.1 State Transition System Model for Planning . . . 104
6.3.2 Classical Representation . . . 105
6.4 Categorical Representation . . . 107
6.4.1 World States as C-sets . . . 108
6.4.2 Actions as Spans in C-Set . . . 112
6.4.3 Applicability using Monomorphisms . . . 113
6.4.4 Transition using Double-Pushout (DPO) Rewriting . . . 115
6.5 Comparison for a Manufacturing Assembly Task . . . 121
6.5.1 Planning Domain . . . 121
6.5.2 Initial State . . . 124
6.5.3 Action . . . 127
6.5.4 Next State . . . 131
6.6 Discussion . . . 133
6.6.1 Advantages of the Categorical Structure for Planning . . . 133
6.6.2 Limitations of the Categorical Structure for Planning . . . 134
6.7 Conclusion . . . 135

Chapter 7: Analogical Transfer using Data Migrations 137
7.1 Motivation . . . 138
7.2 Background . . . 139
7.2.1 Planning Domain Ontology . . . 139
7.2.2 Ontology Alignment . . . 140
7.2.3 Functorial Data Migration . . . 141
7.3 Types of Analogies and Case Studies . . . 143
7.3.1 Direct Analogy . . . 144
7.3.2 Structural Analogy . . . 145
7.4 Categorical Approach to Plan Transfer . . . 146
7.4.1 Ontology Alignment Map, $F : \mathsf{D}_T \to \mathsf{D}_S$ . . . 147
7.4.2 Plan Transfer Migration Functor, $\Delta_F : \mathsf{D}_S\text{-Set} \to \mathsf{D}_T\text{-Set}$ . . . 149
7.5 Software Constructors found within AlgebraicJulia . . . 152
7.6 A Case Study of Direct Analogy: From Assembly to Emergency Communications . . . 155
7.6.1 Domains . . . 155
7.6.2 Ontology Alignment . . . 158
7.6.3 Plan Transfer . . . 160
7.6.4 Validation . . . 165
7.6.5 Summary . . . 166
7.7 A Case Study of Structural Analogy: From Blocksworld to Kitchenworld . . . 167
7.7.1 Domains . . . 167
7.7.2 Ontology Alignment . . . 171
7.7.3 Plan Transfer . . . 173
7.7.4 Validation . . . 177
7.7.5 Summary . . . 180
7.8 Practical Considerations . . . 181
7.8.1 Patterns of Use for Analogical Plan Transfer . . . 181
7.8.2 Integration into a Planning Architecture . . . 184
7.8.3 Designing an Ontology Map . . . 184
7.9 Discussion . . . 187

7.9.1 Advantages of the Categorical Structure for Analogical Transfer 187
7.9.2 Limitations of the Categorical Structure for Analogical Transfer 189
7.10 Conclusion 191

Chapter 8: Conclusion 192
8.1 Summary of Research 192
8.2 Interdisciplinary Impact: Expanding Theoretical Tools for AI and Robotics 193
8.3 Areas of Future Work 194
8.3.1 Sequential Plan Transfer 194
8.3.2 Hierarchical Plan Transfer 195
8.3.3 Structured Knowledge in Planning 196
8.3.4 Analogical Plan Transfer 197
8.3.5 Integrated Sequential-Hierarchical-Analogical (SHA) Transfer Framework . 198
8.4 Concluding Remarks 202

Appendix A: Supplementary Material 203
A.1 Forward Planning using C-sets and DPO Rewriting 203
A.2 Schema Categories 203
A.2.1 Assembly 203
A.2.2 EmergencyComms 205
A.2.3 ColorBlocksworld 205
A.2.4 Kitchenworld 206
A.3 Translation Functors 208
A.3.1 Translation Functor from EmergencyComms to Assembly 208
A.3.2 Translation Functor from Kitchenworld to ColorBlocksworld 208

Appendix B: Scholarly Contributions 211
B.1 Papers 211
B.2 Talks 211
B.3 Community Service 212

Bibliography 213

List of Tables

4.1 World states in terms of explicitly (**bold**) and implicitly propagated literals. 65

5.1 Pseudocode for functor $G : \mathsf{Virtual} \to \mathsf{Motion}$. 92
5.2 A temporal view of Equation 5.6. 94

6.1 Terminology map between the categorical representation and classical representation for planning 121
6.2 The state X in Assembly-Set. 125
6.3 Pre (and Keep) for action AttachPart 128
6.4 NACs (in addition to Pre) for action AttachPart. 128
6.5 Match m from Pre to the initial state X. 131
6.6 Modified IsAttachedTo table within the new state Y. 132

7.1 Table comparing Assembly-Set and EmergencyComms-Set 161
7.2 Table comparing ColorBlocksworld-Set and Kitchenworld-Set. 174
7.3 Table comparing the goal and end states to validate $\pi_{\mathsf{Kitchenworld}}$ 178

List of Figures

1.1 Examples of robots in complex environments. . . . 2
1.2 Types of robot plan, π, transfer. . . . 5

2.1 The Unimate robot pouring a cup of coffee [1]. . . . 11
2.2 Programming using skills in the *Assembly* tool [2]. . . . 12
2.3 The ROCI architecture contains modules, pins, tasks, and nodes [3]. . . . 17
2.4 A plan within a hierarchical task network (HTN) [4]. . . . 18
2.5 Analogies are constructed using maps between interconnected network of concepts. (a) Initial map between concepts. (b) New inference derived from the initial map. (c) A shared pattern between two networks of concepts. Image from [5]. . . . 21
2.6 The case-based planning (CBP) cycle [6]. . . . 24
2.7 An ontology alignment for kitting applications [7]. . . . 25

3.1 Conventional versus string diagram syntax for composite morphisms. . . . 48
3.2 An olog representing the amino acid, arginine [8] 49

4.1 The braiding and identity isomorphisms of string diagrams. . . . 56
4.2 Planar deformation in string diagrams. . . . 57
4.3 Illustration of algorithm for building a string diagram. . . . 61
4.4 String diagram for a plan in the Blocksworld domain. . . . 65
4.5 String diagrams corresponding to actions in the Blocksworld plan. . . . 66

5.1 The RoboCat Framework. . . . 75
5.2 Two robots opening a slide door using the same high-level program. . . . 76
5.3 The full string diagram for the example robot program. . . . 91

6.1 Types of knowledge sources for a knowledge-based planning system [9]. . . . 100
6.2 A semantic graph based on the KnowRob ontology [10]. . . . 103
6.3 A BoxOnSurface-set, namely a C-set for the schema category BoxOnSurface. . . . 108
6.4 An illustration of a single pushout in BoxOnSurface-set. . . . 116
6.5 An illustration of the Assembly domain. . . . 122

7.1 A schematic of analogical plan transfer. . . . 139
7.2 Examples of different types of analogies–direct and structural. . . . 144
7.3 Schematic of analogical plan transfer applied to plan π_S. . . . 149
7.4 Overview of the direct analogy case study. . . . 156
7.5 Ontology alignment using a direct analogy. . . . 159
7.6 The initial and goal states in the Assembly and EmergencyComms domains. . . . 162
7.7 Overview of the structural analogy case study. . . . 168
7.8 Illustration of ontology alignment using a structural analogy. . . . 172
7.9 The initial and goal states in Blocksworld and Kitchenworld. . . . 175

7.10 Schematic of the four patterns of analogical plan transfer: (i) abstract-to-specific, (ii) specific-to-specific, (iii) specific-to-abstract-to-specific, and (iv) abstract-to-abstract. 182

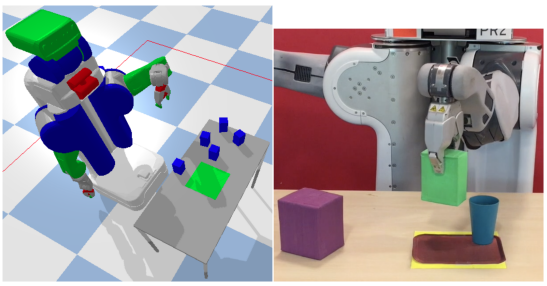
8.1 Example potential application areas for robot plan transfer 200

Chapter 1: Introduction

1.1 Motivation

As the world continues its relentless march towards automating laborious, mundane, and dangerous tasks, the complexity and diversity of those tasks have multiplied. No longer confined to industrial workcells, robots now tackle challenges in disaster response, medical service automation, and space exploration [11]. The tasks robots need to accomplish within each of these environments are abundant—making it difficult to anticipate, and therefore, program all the behaviors a robot might execute.

In response to these challenges, there has been a concerted effort to integrate artificial intelligence (AI) planning into robotics to support automated, task-oriented programming. Frameworks for this integration have been explored in the subdiscipline of task and motion planning (TAMP) [15–19]. However, even with these advanced techniques, it is time-consuming to compute plans as situations arise, making them costly in scenarios that require real-time decision-making. An alternative and potentially more efficient approach is to enable robots to draw on past experience to inform target behaviors. Transfer learning in robotics [20, 21] offers promising avenues for this type of experiential decision-making, providing methods to transfer task knowledge to new contexts and circumventing the need to learn from scratch. That said, much of the discipline focuses on the transfer of motion plans from one robot to another. The wide variation in robot



(a) The PR2 robot serving a meal [12].



(b) The Tiago robot lighting a match and fixing a fuse [13].



(c) The FIDO rover used for sampling Martian soil [14].

Figure 1.1: Examples of robots in complex environments.

embodiments, shown in Figure 1.1, make it challenging to design transfer learning techniques that are generalizable. Moreover, integral to the field is the use of deep learning which has known issues regarding its interpretability [22]—a feature critical for validating that transferred plans behave as expected. The rise of task-level abstractions in robotics due to TAMP presents an opportunity to explore transfer at the task level. To avoid associating the techniques in this dissertation with machine-learning methods, the term **task plan transfer**, or simply **plan transfer**, will be used, rather than 'transfer learning'.

TAMP involves coordinating high-level decision-making (task planning) with actions in physical space (motion planning). Task planning handles symbolic states, which represent the world at an abstract, conceptual level, while motion planning manages geometric states, which capture the

robot's configuration and joint positions. Plan transfer, in this context, refers to reusing a task or motion plan as a solution for a problem it was not originally designed to address. By focusing on task-level plan transfer, methods can be made agnostic to the robot's specific embodiment, leading to better generalization across different robotic platforms. Furthermore, task planning often relies on symbolic knowledge representation, which enhances the interpretability of both planning and plan transfer methods.

The concept of task-level plan transfer is still nascent. Currently, there is no framework that systematically supports the transfer of task plans, and existing formal languages lack the flexibility to capture the full range of potential use cases. Nonetheless, task plan transfer offers several unrealized advantages. Beyond generality and interpretability, it enables robots to execute behaviors that incorporate the constraints and structures of source plans—a feature that is particularly valuable in safety-critical or tightly constrained environments. Transferring plans at the task level also allows robots to tackle complex problems by using existing plans as building blocks for more intricate solutions. Finally, plan transfer enables robots to leverage prior experience to generate innovative behaviors by drawing on past solutions as inspiration for unconventional applications in new domains.

1.2 Research Questions

This research seeks to fill the gap related to task-level plan transfer in robotics. The overall question guiding this research is:

What is an appropriate conceptual and mathematical framework that supports a structured and formal approach to transferring task plans in robotics?

This guiding question is broken down into three questions:

RQ1. What is an appropriate conceptual framework for articulating the full range of task plan transfer use cases?

RQ2. What mathematical representation can formally support this conceptual framework?

RQ3. What unique advantages are afforded by the chosen mathematical representation for task plan transfer?

1.3 Task Plan Transfer

To address this gap and fully explore the advantages of task plan transfer, this dissertation proposes a conceptual framework for discussing different types of transfer methods, along with a mathematical framework to validate the success of each type of transfer.

Throughout this dissertation, a task-level **plan** will be denoted by the symbol π. A formal definition consistent with the concept of a sequential plan, as established within the AI planning community [23], is provided in Definition 1.

Definition 1 (Plan)**.** A **plan**, π, is an ordered sequence of discrete actions, a_i, defined as $\pi = \langle a_1, a_2, a_3, \ldots, a_n \rangle$, which transforms an initial state s_0 into a goal state s_g.

1.3.1 The Conceptual Framework

Existing literature in robotics and AI has extensively explored methods for transferring and adapting robot plans and programs, albeit often in isolation. A closer examination of these approaches reveals that these methods can be unified under a comprehensive framework—one that classifies each transfer use case as a sequential, hierarchical, or analogical transfer. A schematic of each type of transfer is presented in Figure 1.2.

Sequential transfer is grounded in the simple yet powerful idea of composing plans in a linear fashion, which mirrors the way tasks are often executed sequentially in real-world scenarios.

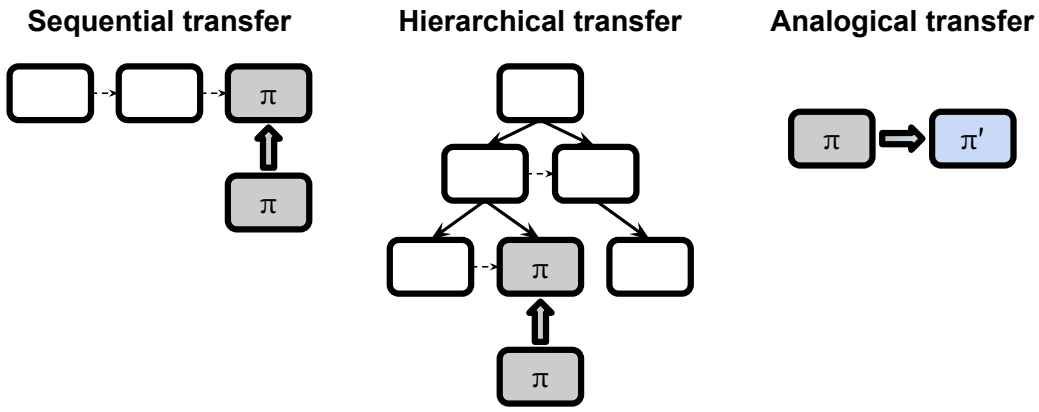


Figure 1.2: Types of robot plan, π, transfer.

The evolution of robot programming, from modules of manual pose recordings to the concepts of motion primitives [24] and skills [25] as means of reusing programs, reflects the core principle of sequential transfer.

Hierarchical transfer naturally arises from the widespread practice of decomposing complex tasks into modular, layered structures—a method that has been central to robot programming and task planning since the conceptualization of modular architectures [26]. Frameworks like the Remote Objects Control Interface (ROCI) [3] and Hierarchical Task Nets (HTNs) [27] emphasized the need for strong conventions and encapsulation, which are key aspects of hierarchical transfer.

Analogical transfer, on the other hand, reflects the cognitive process of leveraging structural similarities between situations to make novel inferences. Analogical reasoning has long been used in AI and robotics, particularly in robot program translation [28] and case-based planning (CBP) [29], which provide methods for mapping aspects of plans and programs so they can be adapted to fit new contexts.

Each type of transfer can dictate components of a more complex transfer scenario. For instance, task plans may be organized hierarchically or sequentially, and it may be desirable to transfer each plan analogically to a new setting. To support this, it is ideal to have a framework

that can coherently reason about all types of transfer within a unified formalism.

1.3.2 The Mathematical Framework

The rigidity of formal languages that are currently used to represent task plans and accompanying knowledge limit their ability to enforce desired properties of sequential, hierarchical, and analogical transfer simultaneously. As a result, the formalisms that might underpin plan transfer approaches become a heterogeneous mixture of formal languages. Efforts to integrate various transfer use cases within a more intricate plan transfer system may inadvertently overlook or violate key features of the underlying languages, leading to discrepancies in their representations. This challenge underscores the need for a formal language that unifies these features within a single framework. The mathematics of **category theory**, with its emphasis on abstraction and compositionality, provides a powerful framework to address these limitations.

Category theory is a branch of mathematics that provides a framework for formalizing and unifying various mathematical concepts through the use of categories, functors, and natural transformations [30]. Categories are structures that capture objects (representing concepts) and morphisms (representing relationships between concepts). Functors allow us to map one category to another, preserving these relationships, and natural transformations provide a structured way to relate different functors to one another. By representing concepts and their relationships in this way, category theory offers a flexible language for modeling interconnected ideas and their interactions in a coherent and modular way. Category theory has been used to understand information structures [31], automate information fusion systems [32], represent quantum processes [33, 34], and synchronize models in systems engineering [35–37]. A consequence of this work is an extension of the repertoire of applications of category theory to the field of robotics and AI, specifically addressing problems regarding systems of interoperable and reusable task plans.

1.4 Summary of Research Contributions

The overall contributions of this work include (i) a conceptual framework for discussing robot plan transfer methods, (ii) a formal language, based on category theory, that is aligned with the proposed framework, (iii) and case studies demonstrating the unique advantages of the formal language in transferring plans. Specific contributions are:

i. **A Conceptual Framework for Task Plan Transfer**

   This research surveys the literature around robot program and plan reuse and considers a unified view of these approaches framed in terms of task plan transfer. The proposed framework establishes a taxonomy of transfer strategies, classifying plan transfer methods as sequential, hierarchical, or analogical (see Chapter 2).

ii. **A New Method for Task Plan Composition**

   This dissertation introduces a method using symmetric monoidal categories (SMCs) and string diagrams to represent and validate action composition within robotic task plans, ensuring coherence across entire plans. This method supports sequential transfer by providing a robust and flexible method for sequentially composing plans (see Chapter 4.2).

iii. **Development of the RoboCat Framework**

   RoboCat, a programming framework grounded in category theory, is introduced to support hierarchical transfer. This framework provides a functorial approach to compile high-level task specifications into low-level robot commands, allowing for modularity and interchangeable programming components (see Chapter 5.5).

iv. **A New Representation Language for Task Planning**

   To enable analogical transfer, a planning representation language based on C-sets and double-

pushout (DPO) rewriting is defined. This language manages complex, structured knowledge and preserves semantic structure in world states and action effects throughout the planning process (see Chapter 6.4).

v. **An Algorithm for Analogical Plan Transfer**

A novel algorithm that leverages ontology alignment maps and automates task plan transfer between different planning domains using functorial data migrations is introduced. This algorithm translates each state and action in the task plan across planning domains, circumventing the need for re-planning in the target domain (see Chapter 7.4.1).

vi. **Case Studies Demonstrating Types of Plan Transfer**

The effectiveness of the proposed framework is demonstrated through case studies within various planning domains, such as in Blocksworld (see Chapter 4), machine-tending (see Chapter 5), manufacturing assembly (see Chapter 6), and a kitchen domain derived from the AI2-THOR simulation environment [38] (see Chapter 7).

1.5 Structure of the Dissertation

The goal of this dissertation is to motivate the cause for research related to task-level plan transfer in robotics and systematically argue the use of category theory as a mathematical framework for plan transfer techniques. Chapter 1 introduces the problem, providing an overview of the motivation and significance of the research. In Chapter 2, the historical context is presented to justify the conceptual framework introduced. Chapter 3 offers a detailed, technical exposition of the core concepts of category theory that serve as the foundation for this work.

The subsequent chapters are organized according to the conceptual framework presented in Section 1.3, highlighting the innovative integration of category theoretic concepts within each task

plan transfer use case.

Chapter 4 explores the use of SMCs [39] and string diagrams [40] as a formalization for sequential transfer. Here, we demonstrate that this approach guarantees global action and plan composition, with a case study focused on composing actions in the Blocksworld domain.

Building on this work, Chapter 5 introduces the RoboCat programming framework, which leverages SMCs and monoidal functors to support the modular specification of low-level robot programs within the context of high-level actions. This framework guarantees valid plan composition while preserving high-level behavior imposed by the task hierarchy—providing a formalization for hierarchical transfer. In this chapter, we present a case study related to the interchange of work-cell doors and robot gripper parts in a machine-tending task.

Prior to discussing analogical transfer, we find it necessary to introduce a formalism that captures knowledge about the world in a precise way. Chapter 6 presents a novel planning representation language that manages structured and connected knowledge in robot task planning through the use of C-sets [41] and double-pushout (DPO) rewriting [42]. This C-set-based representation language enables the analogical transfer of plans between domains using functorial data migrations [43], as detailed in Chapter 7. The case studies in both chapters are centered around manufacturing assembly and kitchen automation tasks.

Finally, Chapter 8 discusses future research opportunities and practical applications of the categorical framework for robot plan transfer, highlighting the impact of the methods introduced in this dissertation.

Chapter 2: Background

2.1 Introduction

At the outset, robots were commended for their versatility [24]. Robots are programmable. This means that a single robot could assist in, theoretically, an endless variety of industrial tasks. Becoming a robot programmer in the 1950s, however, made you more akin to an animator than a computer programmer. This was because the most common form of robot programming was done by manually moving all joints of the robot to their desired positions and recording that pose, as with the Unimate robot in 1954 [44] (see Figure 2.1). When the program was run, it would playback all the poses in the sequence they were recorded [24]. As one could imagine, this way of programming was inefficient, brittle, and not conducive to collaboration between humans. A program to accomplish the composite motion of gripping a beam, moving to a position, and releasing the beam may look different depending on the programmer. Beyond that, where the gripping motion ended and the movement motion began was indistinguishable, requiring a new program for every change to the task or environment.

The inefficiencies of manual programming revealed a desire to reliably preserve and reuse the programming work already done across different tasks and environments. This desire fueled work in robot program translation, robotic software engineering, AI planning, and transfer learning for robotics. The methods emerging from these fields, we argue, can be classified into three core types



Figure 2.1: The Unimate robot pouring a cup of coffee [1].

of program transfer: sequential, hierarchical, and analogical.

2.2 Sequential Transfer in Early Robot Programming

Sequential transfer has been a cornerstone of robot programming and AI planning since the inception of these fields. This approach—composing tasks in a linear, sequential fashion—allowed early roboticists to manage the complexity of tasks by breaking them down into smaller, reusable components. In sequential transfer, the output of one task typically serves as the input for the next, forming the basis for more complex behaviors by chaining simpler ones together. This enabled efficient reuse of plans and programs in both early and modern systems.

2.2.1 Motion Primitives and Skills

One of the earliest applications of sequential transfer in robotics was the introduction of *motion primitives*, as described by Murray et al. [45]. Motion primitives are basic units of movement that serve as building blocks of robot motion for more complex behaviors. For example,

a task as simple as picking up an object could be broken down into distinct, reusable motions: grip the object, lift it, move to a new position, and release. As a result, a sequence of motion primitives could be chained together in novel contexts without needing complete reprogramming. The concept of motion primitives formed the basis for more abstract representations of robot tasks that could be reused across tasks and environments.

Building on this concept, Bogh et. al. [25] introduced task- and object-centric *skills*, which further refined the idea of reusable motion sequences by focusing on task-level abstractions. Skills are composed of motion primitives but are tailored for specific types of tasks, such as industrial assembly, logistics, or assistive robotics. For example, the skill 'Move To' may be part of a logistics task, whereas the skill 'Open' may be part of an assistive task. Bogh et. al. [25] demonstrated that many tasks in logistics and assistive domains could be represented as sequences drawn from a small, predefined library of skills. This decomposition of tasks into reusable skills continues to be a powerful strategy in modern robotic systems evidence by programming graphical user interfaces such as the *Assembly* tool [2] (see Figure 2.2), and others [46–57].

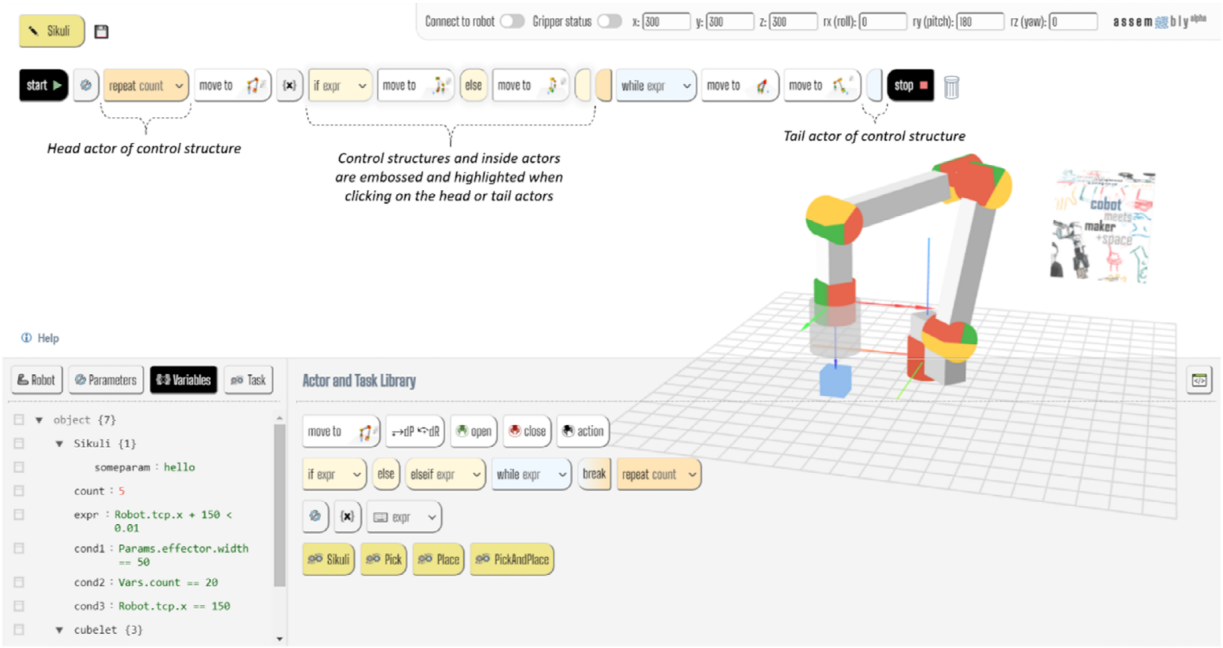


Figure 2.2: Programming using skills in the *Assembly* tool [2].

Another significant development that enabled sequential transfer across diverse robotic systems was the standardization of command languages. One of the earliest examples is the G-code standard [58], which was initially developed for computer numerical control (CNC) machines. G-code enabled sequential execution of predefined commands to control the movement of machine tools. Another significant development was the Canonical Robot Command Language (CRCL) introduced by Proctor et al. at the National Institute of Standards and Technology (NIST) [59]. CRCL provides a high-level task-oriented language for controlling robots across different platforms. By standardizing the command set, G-Code and CRCL enabled seamless sequential execution and reuse of command sequences.

The paradigm of sequential transfer was not limited to robotic programming but also became foundational in AI task planning.

2.2.2 In AI Planning

While motion primitives laid the groundwork for sequential transfer in robotic programming, the concept was also realized in the field of AI planning through representation languages like STRIPS and PDDL. STRIPS (Stanford Research Institute Problem Solver) [60] is a planning language and system that was developed over the course of the late 1960s and early 1970s by Richard Fikes and Nils Nilsson at the Stanford Research Institute. STRIPS was the first automated planning system and is often used as the basis for other more complex planning systems. In 1998, the Planning Domain Definition Language (PDDL) [61] was introduced as a language that extended STRIPS to support a richer set of features, the notion of a planning domain and a planning problem. PDDL is a widely adopted representation for robotic task planning for many reasons, but a beneficial feature is the fact that domain actions can be specified discretely and generically. Their transfer can be realized by sequentially composing grounded forms of these

actions for a variety of planning problems.

To understand how actions are composed in this language, it is essential to consider the theoretical foundations of action composition. In AI planning, problems can be modeled using state machines or finite automata, which provide a formal way of representing the transitions between different states as actions are executed [23]. These models define what it means for a sequence of actions to be valid. For example, the execution of sequential plans is sometimes represented using situation calculus [23]. This formalism checks that executing a sequence of plans leads from an initial situation to a desired final situation. This is represented as $\text{Exec}(\pi_1; \pi_2, s, s')$, which checks that $\pi_1; \pi_2$ leads from situation s to situation s'. This requires an intermediate situation s'' that is an exact match, such that $\exists s''(\text{Exec}(\pi_1, s, s'') \wedge \text{Exec}(\pi_2, s'', s'))$ [23]. In practice, this condition of exactness may be overly conservative. Even minor discrepancies that do not affect the overall outcome could cause a plan to be deemed invalid. Sequential plan execution problems commonly suffer this tension—strict correctness negates the ability to accommodate minor variations that do not compromise the ultimate goal.

In PDDL, plans are composed by validating that the preconditions of each action in the sequence are satisfied by the effects of the preceding action in the plan [61]. This is because most classical PDDL planners do not track the current state throughout the planning process (see Chapter 2.4.5 in [23]). While this approach allows for flexibility, it can make it difficult to precisely trace the provenance of preconditions and effects, especially when they are established several steps earlier. As a result, execution failures may occur when preconditions that were originally satisfied at the start of one plan become invalid after the plan is combined with another. Because of this, manual validation is necessary to ensure that preconditions are met across the entire sequence of actions, particularly when merging plans that were originally solutions to different problems.

2.2.3 Validating Sequential Transfer

As we have shown, sequential transfer has been a fundamental principle in both robotic programming and AI planning. However, as tasks and environments become more complex, the limitations of current methods for composing plans in sequence have become more apparent.

A primary challenge in sequential transfer is the inadequate consideration of preconditions and effects during plan composition. Current methods often assume that the preconditions of each action are automatically satisfied by the effects of the preceding action, which works well in simple use cases but breaks down when handling situations with multistep dependencies. For instance, in warehouse operations, a robot may have separate plans—one to stack an object and pickup a different object, and another to perform another stacking action.

Specifically, the robot may first stack object B on A, then move object C, and finally try to stack C on B, as shown here:

```
plan1:
  stack(B A)
    pre: (clear A) (holding B)
    eff: (not (clear A)) (not (holding B)) (on B A) (clear B)

  pickup(C)
    pre: (clear C)
    eff: (holding C)

plan2:
  stack(C B)
    pre: (clear B) (holding C)
    eff: (not (clear B)) (not (holding C)) (on C B) (clear C)
```

If only direct preconditions and effects are validated, the system might prevent stacking `C` on `B` because the effect of picking up `C` does not satisfy the precondition `(clear b)` necessary for stacking `C` on `B`. This illustrates the need to validate global conditions across the sequence to guarantee successful plan composition.

To ensure the validity of sequential transfer, future research must focus on managing preconditions and effects across the entire plan sequence. One promising direction is the development of more sophisticated plan validation techniques that can track the provenance of preconditions and effects across multiple steps. This would allow for better error handling and flexibility in intricate planning environments which has the potential to improve the reliability of sequential transfer in robotic and AI planning systems.

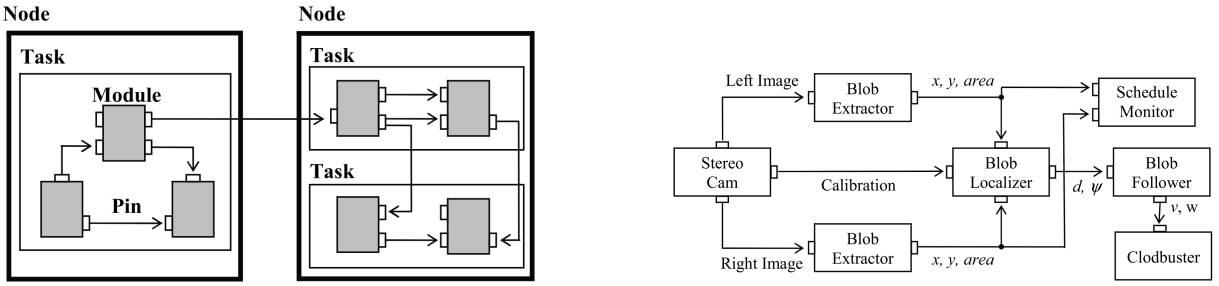
2.3 Hierarchical Transfer in Robotic Software Engineering

As robotic tasks grew in complexity, it became clear that a purely sequential approach to reusing robot program and plans was insufficient. The need for a more structured and scalable method of organizing tasks led to the emergence of patterns that can be best described as “hierarchical". Hierarchical transfer involves reusing plans that have been decomposed into components within modular, layered structures, where each module can be independently executed and reused across different contexts. This approach mirrors the problem-solving strategies employed by human engineers, who often break down complex problems into manageable sub-tasks. By the 1980s, robot programming languages began to incorporate the concept of hierarchical separation-of-concerns, transitioning from simple point-to-point control to more sophisticated task-oriented interfaces [24, 62]. This shift allowed for more efficient task reuse, enabling robots to perform increasingly complex operations by organizing plans in a modular, hierarchical fashion.

2.3.1 Modular Architectures

The adoption of modular architectures in the 2000s further solidified hierarchical transfer as a foundational principle in robotic software engineering [63]. These architectures facilitated the creation of independent units of behavior, which could be organized hierarchically and reused

across various platforms. This approach is exemplified by several modular programming frameworks and middlewares that emerged during this time [3, 64–66]. Frameworks such as the Remote Objects Control Interface (ROCI) [3] emphasized the importance of encapsulation and modularity, introducing strongly-typed interfaces that allowed for reliable communication between modules.



(a) Components of the ROCI architecture.

(b) An example of a ROCI task.

Figure 2.3: The ROCI architecture contains modules, pins, tasks, and nodes [3].

While this approach was promising, one limitation was its reliance on strict guidelines for interface design, which had to be communicated between human engineers. Like many other frameworks [65, 67–71], ROCI lacked a formalism that could automatically validate the correct arrangement of modules. This absence of formal validation introduced potential vulnerabilities in system reliability, as errors in module design or integration could remain undetected until runtime, rather than being caught during development.

2.3.2 Hierarchical Task Networks (HTNs)

The hierarchical paradigm gained significant attention in the AI planning community, as evidenced by the successful introduction of hierarchical task networks (HTNs) [27] and their many applications [4, 17, 72–75]. HTNs allow for the modular construction of plans, where sub-tasks can be independently developed and reused across various planning problems.

The core idea behind HTNs is to decompose tasks hierarchically, allowing a system to tackle large, complex problems by breaking them down into simpler, interconnected steps (see

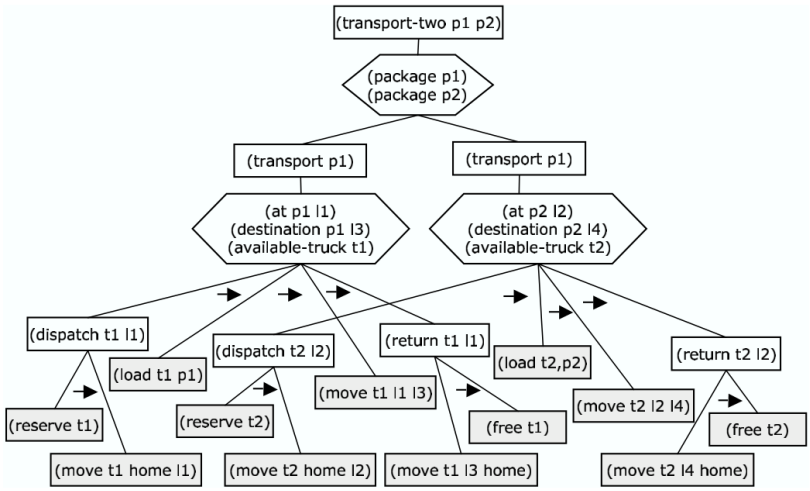


Figure 2.4: A plan within a hierarchical task network (HTN) [4].

Chapter 11 of [23]). In an HTN, tasks are categorized into two types: *primitive tasks*, which represent actions that can be directly executed, and *compound tasks*, which are decomposed into a series of compound or primitive tasks. *Methods* specify the decomposition of compound tasks into subtasks. Methods, like actions, are context-dependent and apply only when certain preconditions are satisfied. Whether a method is applicable depends on the current state and is determined at the time of planning. This process continues recursively until all tasks are reduced to primitive tasks, which are then executed sequentially to achieve the overall goal. An example of a plan within an HTN is illustrated in Figure 2.4.

While this hierarchical decomposition enables efficient planning within a specific context, it also introduces significant limitations when it comes to reusing plans or methods across different different problems. As stated, HTN plans are highly context-dependent and methods are typically defined with limited regard for their impact on other parts of the plan [76]. This makes integrating a new method into an existing plan particularly difficult. The effects of primitive tasks are only validated locally, which can result in tasks appearing valid in isolation but failing when executed as part of a larger plan. Consequently, any attempt to incorporate new methods or adapt existing

plans often requires replanning from scratch, as cross-method interactions and cumulative effects across the hierarchy are not automatically accounted for. While the resulting plan may appear correct in theory, it can be flawed in practice due to unhandled interactions and dependencies between methods.

2.3.3 Validating Hierarchical Transfer

While reusing plans and programs within a hierarchical structure has proven to be a powerful approach in both robotic software engineering and AI planning, several challenges remain that limit its full potential. One of the most significant issues is the lack of formal mechanisms to enforce the consistent interpretation and integration of modules across different layers of a hierarchy. This absence of formal enforcement can lead to confusion, particularly when modules developed in isolation are reused in different contexts or combined in novel ways. Without a formal approach guiding the integration, subtle differences in how modules interpret inputs or generate outputs can lead to unexpected behavior, potentially undermining the reliability of the entire system.

To advance hierarchical transfer methods, future research must focus on developing formal frameworks that ensure consistent interpretation and integration of modules across all levels of the hierarchy. Such frameworks could involve formal methods for specifying and verifying the interfaces between modules, ensuring they not only adhere to syntactic standards but also meet semantic expectations.

2.4 Analogical Transfer in AI Planning and Transfer Learning

While sequential and hierarchical transfer methods effectively address the challenges of reusing and adapting tasks within a single context, they often fall short when it comes to transferring knowledge across different context—a desirable feature for general-purpose robotics. Ap-

proaches that construct mappings between domains, such as program translation, transfer learning in robotics, and case-based reasoning in AI planning aim to address this limitation.

Wilkins and DesJardins [9], in their call for knowledge-based planning, argued that an ideal system should mimic human behavior by using *analogies* to transfer solutions from one problem to another. This perspective supports the idea of identifying structural similarities between contexts and using them to adapt plans from one context to another—which we refer to as analogical transfer.

2.4.1 Formal Theory of Analogy

Analogy has long been recognized as a fundamental process in human reasoning. Analogies have been studied extensively in the field of cognitive psychology since the foundational work "Structure-Mapping: A Theory of Analogies" was published by Gentner in 1983 [77]. Analogical reasoning in this context is defined as the cognitive activity related to identifying common features of two situations and deriving further inferences based on those relations [5]. According to the formal theory of analogies, there are three phases in analogical reasoning: retrieval, mapping, and evaluation. The mapping stage involves identifying and aligning commonalities between two memories, otherwise known as structure mapping. An example of structure mapping can be seen in Figure 2.5. According to this theory, analogical maps are *structurally consistent*, meaning they must adhere to two key principles—(i) *one-to-one correspondence*, where each concept in the source memory maps to at most one concept in the target memory, and (ii) the *systematicity principle*, where analogical maps connect large, densely connected systems of concepts, instead of just a handful of isolated concepts. This formalization is the most widely accepted framework for understanding analogical processing and provides a strong basis for transferring plans and programs between contexts in robotics.

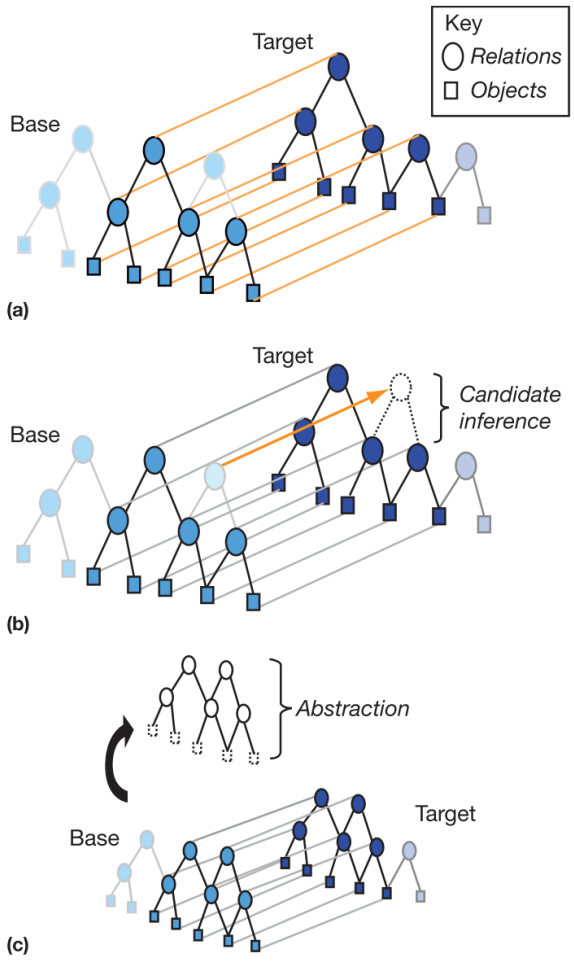


Figure 2.5: Analogies are constructed using maps between interconnected network of concepts. (a) Initial map between concepts. (b) New inference derived from the initial map. (c) A shared pattern between two networks of concepts. Image from [5].

Approaches across programming languages, transfer learning, and AI planning contribute promising methods toward this goal. Here, we evaluate these methods based on how well they address the criteria for transferring plans and programs using analogies.

2.4.2 Using Program Translation

At the intersection of programming languages and robotics, analogy-like transfer methods manifest as program translation methods. Frameworks for translating robot programs to preserve behavior across different robot platforms were introduced in the early 2000s. Freund et al. [28,78]

presented a novel framework for robot program translation, which generates robot program variants in manufacturer-specific languages. A key component of their framework is a syntax map between the two controller languages, which facilitates translation from the source program to the target language.

The translation framework converts high-level robot programming languages into controller-specific code using abstract syntax trees (ASTs) and predefined templates. While it ensures that commands are correctly mapped to the target language, it does not adhere to the systematicity principle because it does not consider the connected structure that results from the grammar of the language. Other systems such as AdaptPack Studio Translator [79,80] and MontiArcAutomaton [81, 82] follow similar architectures to Freund's framework, inheriting the same limitations in supporting analogical transfer for task plans. While these frameworks successfully clone the operational behaviors of one robot onto another, they fall short when it comes to transferring higher-level task plans that govern abstract decision-making and problem-solving. Nevertheless, the energy behind these approaches highlights the demand for methods that can imbue robots with the behavior of another.

2.4.3 Using Transfer Learning

Recent methods that share similar goals to analogical transfer can be found in the emerging discipline of transfer learning in robotics. Transfer learning in robotics [20] often depends on a universal representation constructed using meta-neural networks, frequently incorporating reinforcement learning (RL) architectures that attempt to learn multiple tasks simultaneously [21,83–87].

One particular approach claims to train an RL model to learn analogies between source and target subtasks by enforcing an analogy-making constraint in its objective function [85]. This objective imposes constraints that ensure the differences between task embeddings are consistent

across analogous tasks. For example, it enforces constraints such as $||\Delta(g_A, g_B) - \Delta(g_C, g_D)|| \approx 0$ where $g_A : g_B :: g_C : g_D$. In other words, if tasks g_A and g_B are analogous to tasks g_C and g_D, the difference vectors between their embeddings should be similar.

This approach aligns with the rigorous definition of analogies between planning domains by attempting to establish one-to-one correspondences and maintain systematic relationships in the task embedding space. However, as with many deep-learning methods, it lacks the interpretability and transparency needed to ensure that these analogical mappings are accurately established and properly utilized. This opaqueness makes it difficult to verify whether the inferences generated by these models are valid, relevant, or capable of providing new knowledge that is beneficial to the task. Therefore, while transfer learning approaches show promise, they often fall short at creating interpretable analogies.

2.4.4 Using Case-based Planning

The methods that best align with the formal theory of analogies come from the fields of AI planning and knowledge representation. Case-based planning (CBP), pioneered by Janet Kolodner [29], aims to solve new problems by retrieving and adapting previous plans based on past experiences, as shown in Figure 2.6. Early systems like CYRUS [29] exemplified this approach by retrieving relevant plans, mapping them to new problems, and modifying them as needed to fit the new context. Since then, many algorithms [88–94] have been developed to improve the efficiency and effectiveness of CBP in different domains.

The mapping process within CBP identifies correspondences between the elements of the source and target problems. A key aspect of CBP is its focus on preserving the structural relationships between the steps in the retrieved plan. CBP methods also support the generation of new inferences by adapting the retrieved plan to fit the new problem. This adaptation process

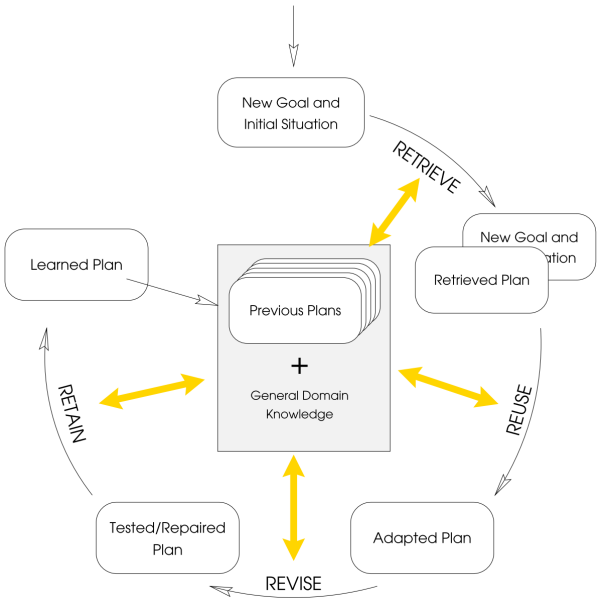


Figure 2.6: The case-based planning (CBP) cycle [6].

involves identifying differences between the old and new problems, modifying the plan accordingly, and generating new steps or actions that were not explicitly present in the original plan. Despite these strengths, CBP methods have not been widely adopted in robotics due to several practical challenges [95]. One of the key issues is the high variability and context sensitivity of real-world robotic tasks. Variations in planning domains make it difficult to retrieve and adapt relevant past cases, particularly when they originate from different planning domains with different constraints and requirements.

2.4.4.1 Modern Approaches resembling Case-based Planning

Several modern approaches have been proposed that resemble the plan reuse step of CBP. For instance, Elimelech et al. [96, 97] introduced a method for cross-domain task plan transfer, which focuses on caching abstract skills in a dynamically-defined abstract domain. Their approach transfers successful task plans into an abstract domain where they are stored as *abstract skills*. This technique utilizes *abstraction keys* to generalize these skills and reconstruct them in new

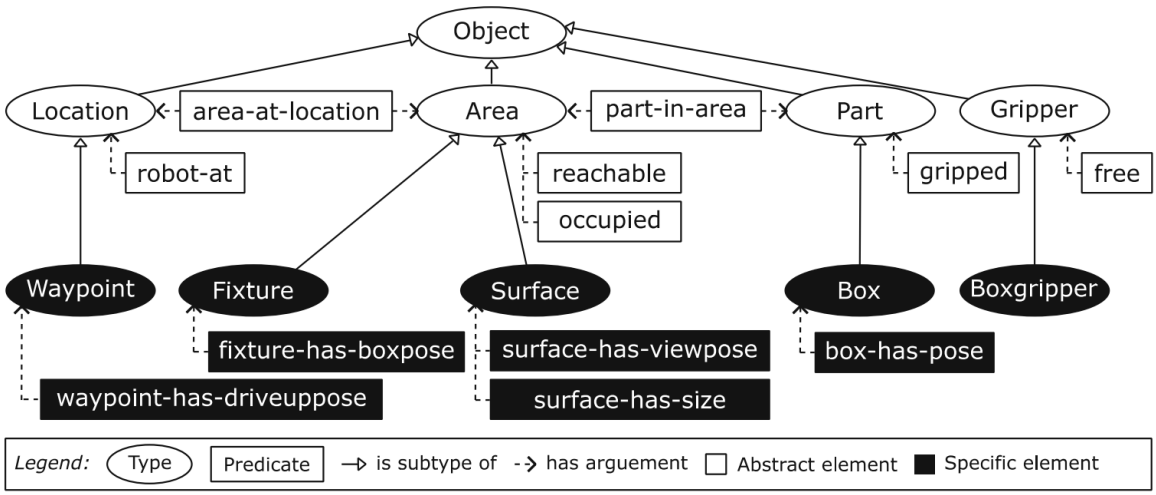


Figure 2.7: An ontology alignment for kitting applications [7].

domains on demand. The method adheres to the strict definition of analogies by establishing one-to-one correspondences through abstraction keys. This correspondence is said to maintain the plan structure formed by dependencies between states in the original plan. In another approach, Segovia-Aguas et al. [98] propose transferring plans by guiding planning in the target scenario using landmark states. This ensures that plans achieve key intermediate states, which are necessary across different problems and contexts.

In both approaches, the correspondences are made between plans, rather than between the underlying domain structure. While these methods effectively generalize and transfer plans across different domains, they do not explicitly maintain or map the intricate relationships and dependencies within domain knowledge. As a result, these techniques excel at high-level plan transfer but may not fully capture deeper analogical connections between domain elements.

2.4.5 Using Ontology Alignment

Ontology alignment is an area of research in knowledge representation that provides promising candidates for establishing analogies between knowledge of different planning domains. During ontology alignment, correspondences between entities (such as types, predicates, or instances) in

different ontologies are established with consideration for the overall structure [99]. Given that ontologies are often implemented according to logical frameworks, logical inferences can be made based on the aligned entities and their relationships. Some examples of ontology alignment within robotics include work done by Kainzner et. al. [100] and Heuss et. al. [7]. Kainzner et. al. used ontology alignment to integrate and standardize knowledge across different ontologies in manufacturing maintenance. Heuss et. al. presented a novel method that aligns the ontologies from abstractly modeled planning domains to application-specific planning domains for industrial tasks (see Figure 2.7). These approaches, however, do not suggest how to use aligned ontologies to reuse plans from one domain in another, which provides an avenue of future research.

2.4.6 Validating Analogical Transfer

Analogical transfer is particularly relevant in robotics as it enables the reuse of task plans across different domains—something that sequential and hierarchical transfer methods struggle to achieve. By leveraging analogy, robots can transfer solutions from one domain to another, allowing them to adapt to new and unforeseen challenges more effectively. Despite promising advancements, some open problems related to validating analogical transfer remain.

One key challenge is the need for formal frameworks to establish analogical maps, as well as the limited interpretability of analogies generated by deep learning models. A critical issue in this work is preserving the highly-connected relational structures within world knowledge during plan transfer. While current methods focus on overall plan correspondence, they often fail to maintain the intricate dependencies and relationships embedded in domain knowledge, which are essential for true analogical reasoning.

Ontology alignment shows promise in establishing analogies between knowledge domains; however, further research is needed to leverage these aligned ontologies for reusing plans across

different domains. These gaps underscore the necessity for more sophisticated frameworks that integrate interpretability, preserve detailed relational structures, and support systematic plan reuse. Advancing these capabilities is essential for the practical application of analogical plan transfer in robotics.

2.5 Summary

In summary, program and plan reuse in robotics and AI has evolved through three paradigms: sequential, hierarchical, and analogical transfer. Transfer methods rarely manifest in isolation, and modern task planning may require combining these approaches. The future of plan transfer lies in developing a formal representation that accommodates all three within a single, interoperable framework. This integration would create a more powerful and flexible approach to task planning while enabling a broader range of transfer techniques. In the next chapter, we introduce category theory's formal concepts, which serve as the basis for a unified language that supports the interoperability of sequential, hierarchical, and analogical transfer.

Chapter 3: Category Theory Foundations

Plan transfer would allow a robot to solve past and present problems simultaneously. One way to be in two places at once is through *homomorphisms*.

In mathematics, a homomorphism is a map between two algebraic structures of the same type that preserves their structure. In other words, homomorphisms consider the relationships between elements in the domain and ensures that the image of these relationships are respected in the codomain. In this way, homomorphisms provide a bridge between two worlds—one that ensures that events happening in the codomain are consistent reflections of those in the domain.

If the structures *composition* and *identity* govern the domain, then the homomorphism becomes what category theorists call a *functor*. In far more precise terms, the foundational paper by Samuel Eilenberg and Saunders Mac Lane, "General Theory of Natural Equivalences," published in 1945 [30], introduced the basic concepts of category theory—namely, categories, functors, and natural transformations. Eilenberg and Mac Lane developed category theory as a tool to formalize and unify various mathematical concepts in algebra and topology. Over time, category theory has proven to be a powerful and versatile mathematical tool that can provide a coherent framework naturally aligned with the principles of composition, modularity, knowledge representation, and analogies. In this chapter, we will introduce definitions of concepts from category theory that are foundational to the work of this dissertation.

Before providing formal definitions, let us ease into the discussion for unfamiliar readers of

mathematics. It is helpful to think of mathematical structures as languages, each providing its own types and rules to express certain ideas. For instance, in the language of graphs, there are two basic types: (i) nodes and (ii) edges. The primary rule governing this language is simple—an edge connects two nodes (in an undirected graph). This minimal set of types and rules is sufficient for describing certain relationships, more complex relationships, however, would require additional structure.

The language of a category, on the other hand, introduces two different types: (i) **objects** and (ii) **morphisms**. At first glance, one might assume that graphs and categories have similar expressive power because they both rely on two basic types. However, categories are equipped with a richer set of rules that allow for more sophisticated constructions. Here are the fundamental rules of categories:

- **Morphisms** are directed arrows between two objects, going from a *domain* object and to a *codomain* object.
- **Morphisms** can be *composed* if the codomain of one morphism matches the domain of another.
- This composition operation is *associative* which means the order in which a pair of morphisms are evaluated does not affect the result.
- Each object has a special morphism, called the **identity morphism**, which maps the object to itself.
- **Categories** consist of a collection of objects and a collection of morphisms that obey these rules.

These basic rules lay the foundation for category theory. The simplicity and flexibility of these definitions allows categories to model more complex structures while still maintaining closure within the categorical language. For instance, a category itself can be an object in a category.

In this case, a morphism between two categories would be called a **functor**. Because a functor maps between two categories and preserves composition and identity, it aligns with the concept of a homomorphism but operates within the richer context of category theory. Many structures can be built out of these rules, as we will see in the remainder of this chapter.

3.1 Categories, Functors, Natural Transformations

Categories, functors, and natural transformations are fundamental building blocks for many category-theoretic constructs [101]. To begin, we formally define a category in Definition 2.

Definition 2 (Category)**.** A **category** C consists of the following components:

- A collection of **objects**, denoted $\mathrm{Ob}(\mathsf{C})$;
- For each pair of objects $X, Y \in \mathrm{Ob}(\mathsf{C})$, a collection $\mathrm{Hom}_{\mathsf{C}}(X, Y)$ of **morphisms from** X **to** Y, with each morphism $f \in \mathrm{Hom}_{\mathsf{C}}(X, Y)$ denoted as $f : X \to Y$;
- A **composition** operation which, for every pair of morphisms $f : X \to Y$ and $g : Y \to Z$, defines a **composite** morphism $g \circ f : X \to Z$;
- For every object X, an **identity** morphism $1_X : X \to X$.

These components must satisfy the following laws:

- (Associativity law) For any composable morphisms f, g, and $h : Z \to W$, the equation $h \circ (g \circ f) = (h \circ g) \circ f$ must hold.
- (Unitality law) For every morphism $f : X \to Y$, composition with the identity morphisms must satisfy $f \circ 1_X = f$ and $1_Y \circ f = f$.

Some canonical examples of categories include the categories Set, Pos, and Vec_k [101, 102].

Example 1 (Set)**.** The category Set is a category whose objects are sets and whose morphisms are functions.

Example 2 (Pos)**.** The category Pos is the category whose objects are partially ordered sets (posets) and whose morphisms are monotone functions between them.

Example 3 (Vec_k)**.** The category Vec_k is the category whose objects are vector spaces over a fixed field k, typically $\mathbb{R}$ or $\mathbb{C}$, and whose morphisms are linear maps between those vector spaces.

Some additional categories with concrete examples may aid readers, such as the category formed by the powerset of $\{1, 2\}$, or $\mathcal{P}(\{1, 2\})$, and the category formed by the preorder of natural numbers, $\mathbb{N}_{\leq}$.

Example 4 ($\mathcal{P}(\{1, 2\})$)**.** The category of the powerset of $\{1, 2\}$, $\mathcal{P}(\{1, 2\})$, is the category whose objects are elements of the powerset of $\{1, 2\}$, $\mathcal{P}(\{1, 2\}) = \{\emptyset, \{1\}, \{2\}, \{1, 2\}\}$. For objects $X, Y \in \mathcal{P}(\{1, 2\})$, there exists a unique morphism $f : X \to Y$ if and only if $X \subseteq Y$, namely *subset inclusion*.

Example 5 ($\mathbb{N}_{\leq}$)**.** The category of the preorder of natural numbers, $\mathbb{N}_{\leq}$, is the category whose objects are natural numbers, $\mathbb{N} = \{0, 1, 2, \dots\}$. For objects $m, n \in \mathbb{N}$, there exists a unique morphism $f : m \to n$ if and only if $m \leq n$ according to the standard ordering of natural numbers.

A category can be qualified as being **finitely generated** which means that all objects and morphisms within each category are derived from a finite set of objects and morphisms. For example, the category $\mathcal{P}(\{1, 2\})$ is finitely generated.

Categories can themselves be treated as objects in a larger category called Cat, whose morphisms are known as **functors**. A functor describes a mapping between two categories, preserving both the composition of morphisms and identity morphisms. Formally:

Definition 3 (Functors)**.** A **functor** F from a category C to another category D, denoted $F : \mathsf{C} \to \mathsf{D}$, consists of:

- A function $F : \mathrm{Ob}(\mathsf{C}) \to \mathrm{Ob}(\mathsf{D})$ that maps each object in C to an object in D;
- For every pair of objects $X, Y \in \mathsf{C}$, a function $F : \mathrm{Hom}_{\mathsf{C}}(X, Y) \to \mathrm{Hom}_{\mathsf{D}}(F(X), F(Y))$ that maps each morphism in C to a morphism in D.

These functions must satisfy the following conditions:

- $F(g \circ f) = F(g) \circ F(f)$ for every composable pair of morphisms $X \xrightarrow{f} Y \xrightarrow{g} Z$ in C;
- $F(1_X) = 1_{F(X)}$ for every object $X \in \mathsf{C}$.

Functors allow us to capture information about a category by relating it to other categories. For example, consider functors from the category $\mathcal{P}(\{1, 2\})$ to $\mathbb{N}_{\leq}$. Such functors can convey information about objects in $\mathcal{P}(\{1, 2\})$, such as their cardinality (size) or the sum of their elements.

Example 6 (Cardinality functor for $\mathcal{P}(\{1, 2\})$)**.** Let $|.| : \mathcal{P}(\{1, 2\}) \to \mathbb{N}_{\leq}$ be a functor that maps each subset in $\mathcal{P}(\{1, 2\})$ to its cardinality as a natural number, and each subset inclusion relation to the corresponding inequality between their cardinalities.

Object map. The functor $|.|$ maps each object (set) in $\mathcal{P}(\{1, 2\})$ to a natural number based on its cardinality:

$$\emptyset \mapsto 0, \quad \{1\} \mapsto 1, \quad \{2\} \mapsto 1, \quad \{1, 2\} \mapsto 2$$

Morphism map. The functor $|.|$ also maps each morphism in $\mathcal{P}(\{1, 2\})$ to a morphism in $\mathbb{N}_{\leq}$:

$$f : X \to Y \quad (\text{where } X \subseteq Y) \in \mathcal{P}(\{1, 2\}) \quad \mapsto \quad |X| \leq |Y| \in \mathbb{N}_{\leq}$$

Specifically, the morphism mappings are:

$$\emptyset \subseteq \{1\} \mapsto 0 \leq 1, \qquad \{1\} \subseteq \{1, 2\} \mapsto 1 \leq 2, \qquad \emptyset \subseteq \{2\} \mapsto 0 \leq 1,$$

$$\{2\} \subseteq \{1, 2\} \mapsto 1 \leq 2, \qquad \emptyset \subseteq \{1, 2\} \mapsto 0 \leq 2$$

Example 7 (Sum functor for $\mathcal{P}(\{1,2\})$)**.** Let $\Sigma : \mathcal{P}(\{1,2\}) \to \mathbb{N}_{\leq}$ be a functor that maps each subset in $\mathcal{P}(\{1,2\})$ to the sum of its elements as a natural number, and each subset inclusion relation to the corresponding inequality between their sums.

Object map. The functor Σ maps each object (set) in $\mathcal{P}(\{1,2\})$ to a natural number based on the sum of its elements:

$$\emptyset \mapsto 0, \quad \{1\} \mapsto 1, \quad \{2\} \mapsto 2, \quad \{1,2\} \mapsto 3$$

Morphism map. The functor Σ also maps each morphism in $\mathcal{P}(\{1,2\})$ to a morphism in $\mathbb{N}_{\leq}$:

$$\Sigma : f : X \to Y \quad (\text{where } X \subseteq Y) \in \mathcal{P}(\{1,2\}) \quad \mapsto \quad \Sigma(A) \leq \Sigma(B) \in \mathbb{N}_{\leq}$$

Specifically, the morphism mappings are:

$$\emptyset \subseteq \{1\} \mapsto 0 \leq 1, \qquad \{1\} \subseteq \{1,2\} \mapsto 1 \leq 3, \qquad \emptyset \subseteq \{2\} \mapsto 0 \leq 2,$$

$$\{2\} \subseteq \{1,2\} \mapsto 2 \leq 3, \qquad \emptyset \subseteq \{1,2\} \mapsto 0 \leq 3$$

Functors can also serve as objects within a category. In such a category, the objects are functors between fixed domain and codomain categories, such as C and D. The morphisms between these functors are called **natural transformations**. Formally:

Definition 4 (Natural Transformation)**.** Given two categories C and D, a **natural transformation** α between two functors $F, G : \mathsf{C} \to \mathsf{D}$ is a collection of morphisms in D:

$$\alpha_X : F(X) \to G(X) \text{ for each object } X \in \mathsf{C}$$

$\mathbb{N}_{\leq}$. Namely, for every morphism, $f : X \to Y$, in C the following diagram commutes:

$$\begin{array}{ccc} F(X) & \xrightarrow{F(f)} & F(Y) \\ {\scriptstyle \alpha_X}\downarrow & & \downarrow{\scriptstyle \alpha_Y} \\ G(X) & \xrightarrow[G(f)]{} & G(Y) \end{array}$$

This commutativity condition means that: $\alpha_Y \circ F(f) = G(f) \circ \alpha_X$.

Notice that, like $F(f)$ and $G(f)$, the natural transformation components, η_X and η_Y, are morphisms in D. For example, the components of the natural transformation between the cardi-

nality functor, $|.| : \mathcal{P}(\{1,2\}) \to \mathbb{N}_{\leq}$, and sum functor, $\Sigma : \mathcal{P}(\{1,2\}) \to \mathbb{N}_{\leq}$ are inequality relations between natural numbers.

Example 8 (Natural transformation $\alpha : |.| \Rightarrow \Sigma$)**.** The natural transformation $\alpha : |.| \Rightarrow \Sigma$ is defined by the mappings $\alpha_X : |X| \to \Sigma(X)$ for each object X in $\mathcal{P}(\{1,2\})$. The object map for α is defined as follows:

$$\alpha_{\emptyset} : |\emptyset| = 0 \to \Sigma(\emptyset) = 0,$$

$$\alpha_{\{1\}} : |\{1\}| = 1 \to \Sigma(\{1\}) = 1,$$

$$\alpha_{\{2\}} : |\{2\}| = 1 \to \Sigma(\{2\}) = 2,$$

$$\alpha_{\{1,2\}} : |\{1,2\}| = 2 \to \Sigma(\{1,2\}) = 3.$$

The transformation shown in Example 8 demonstrates how cardinality can be associated with the sum of elements. Though simple, this example shows how natural transformations provide a structure for examining how values change under different functors. This concept will be particularly useful for managing states and actions in planning in this dissertation.

3.1.1 Some Notable Functors and Natural Transformations

Of the many constructions that can be achieved using categories and functors, we highlight a few essential ones that will be used throughout this dissertation.

3.1.1.1 Diagrams and Spans

A diagram in category theory is a functor that maps an **index category** to a category, C. The purpose of a diagram is to identify a specific pattern of objects and morphisms, given by the index category, within another category. Formally, a diagram is defined as:

Definition 5 (Diagram, Index Category)**.** Let C be a category and J be an **index category**. A **diagram** D of shape J in the category C is a functor $D : \mathsf{J} \to \mathsf{C}$.

One common example of a diagram that will appear frequently in this dissertation is a **span**.

Definition 6 (Span)**.** A **span** is a diagram with an index category containing three objects and two morphisms arranged like:

$$X \xleftarrow{f} Z \xrightarrow{g} Y$$

Here, f is called the **left leg** of the span and g is called the **right leg** of the span. The object Z is called the **apex** of the span.

Example 9 (Span in Set)**.** Let the category J be:

$$X \xleftarrow{f} Z \xrightarrow{g} Y$$

Object map. The span, $D : \mathsf{J} \to \mathsf{Set}$, maps the objects in J:

$$Z \mapsto \{1, 2\}, \qquad X \mapsto \{c, d, e\}, \qquad Y \mapsto \{a, b\}$$

Morphism map. The span, D, maps the morphisms in J:

$$D(f) : Z \to X, \text{ where } f = \{1 \mapsto d, 2 \mapsto e\}$$

$$D(g) : Z \to Y, \text{ where } g = \{1 \mapsto a, 2 \mapsto b\}$$

A Set-valued span captures how the apex set, Z, simultaneously embeds into the target sets X and Y. Later, we will see that a universal property in category theory can be used to construct a new object in Set that represents the disjoint union of sets X and Y, quotiented by the equivalence induced by their common elements in Z.

3.1.1.2 Representable Functor

Another noteworthy functor is a **representable functor**. A representable functor [103] is a functor that maps an object, $X \in \mathsf{C}$, to the set of morphisms originating from X. By doing so, the representable functor captures how that object relates to all others within the category. We

formally define a representable functor as:

Definition 7 ((Covariant) Representable Functor)**.** Given a category C and an object X in C, the **hom-functor**, $\mathrm{Hom}_{\mathsf{C}}(X, -) : \mathsf{C} \to \mathsf{Set}$, is defined as follows:

- Each object $Y \in \mathsf{C}$ is mapped to the set of morphisms $\mathrm{Hom}_{\mathsf{C}}(X, Y)$ in C:

$$\mathrm{Hom}_{\mathsf{C}}(X, -)(Y) = \mathrm{Hom}_{\mathsf{C}}(X, Y)$$

- Each morphism $g : Y \to Z$ in C is mapped to the function:

$$\mathrm{Hom}_{\mathsf{C}}(X, -)(g) : \mathrm{Hom}_{\mathsf{C}}(X, Y) \to \mathrm{Hom}_{\mathsf{C}}(X, Z)$$

which maps each morphism $f \in \mathrm{Hom}_{\mathsf{C}}(X, Y)$ to the composite morphism $g \circ f : X \to Z$.

A functor $F : \mathsf{C} \to \mathsf{Set}$ is called a **representable functor** if there exists an object X in C and a natural isomorphism [103] $F \cong \mathrm{Hom}_{\mathsf{C}}(X, -)$.

Let us take an example of a hom-functor $\mathrm{Hom}_{\mathsf{C}}(X, -)$ for a specific set X, and demonstrate how it maps objects and morphisms in the category Set.

Example 10 (Hom-Functor in Set)**.** Let $X = \{1, 2\}$ be a fixed set in Set. We define the functor $\mathrm{Hom}_{\mathsf{Set}}(X, -) : \mathsf{Set} \to \mathsf{Set}$, which maps each object Y in Set to the set of functions from X to Y, denoted by $\mathrm{Hom}_{\mathsf{Set}}(X, Y)$.

For any set $Y \in \mathsf{Set}$, $\mathrm{Hom}_{\mathsf{Set}}(X, Y)$ is the set of all functions from X to Y. For example, if $Y = \{a, b\}$, then $\mathrm{Hom}_{\mathsf{Set}}(X, Y)$ consists of all functions $f : X \to Y$. In this case, there are $2^2 = 4$ possible functions, since each element in X can be mapped to either a or b, namely:

$$\mathrm{Hom}_{\mathsf{Set}}(X, Y) = \{f_1, f_2, f_3, f_4\}$$

where:

$$\begin{aligned} f_1(1) &= a, \quad f_2(1) = a, \quad f_3(1) = b, \quad f_4(1) = b \\ f_1(2) &= a, \quad f_2(2) = b, \quad f_3(2) = a, \quad f_4(2) = b \end{aligned}$$

Thus, $\mathrm{Hom}_{\mathsf{Set}}(X, Y)$ is the set of all functions from $\{1, 2\}$ to $\{a, b\}$.

Now, let $Z = \{c\}$. Let $g : Y \to Z$ be a function between sets Y and Z such that

$g : \{a, b\} \to \{c\}$ is the unique function that maps both a and b to c. The morphism $\mathrm{Hom}_{\mathsf{Set}}(X, -)(g)$ contains the following maps:

$$f_1 \mapsto g \circ f_1, \quad f_2 \mapsto g \circ f_2, \quad f_3 \mapsto g \circ f_3, \quad f_4 \mapsto g \circ f_4$$

With this example in mind, it is easier to see how hom-functors can generalize to categories beyond Set. A functor $F : \mathsf{C} \to \mathsf{Set}$ is called *representable* if there exists an object X in C such that F is naturally isomorphic to the hom-functor $\mathrm{Hom}_{\mathsf{C}}(X, -)$. In other words, F is representable if it "behaves like" the hom-functor $\mathrm{Hom}_{\mathsf{C}}(X, -)$, mapping each object Y to the set of morphisms from X to Y.

Representable functors are particularly useful for isolating the relationships of an object with all other objects in a category. For example, if the category represents a type hierarchy within an ontology, a representable functor for a given type could collect all connected types in that ontology. With this understanding, universal properties—particularly colimits—over diagrams of representable functors can induce a more expansive network of types, which is leveraged in this dissertation.

3.2 Universal Properties

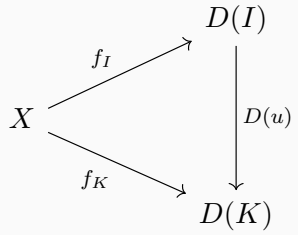
Objects in categories can satisfy universal properties such as being a **limit** or **colimit** [101]. A universal property characterizes an object uniquely (up to isomorphism) by specifying its relationships to other objects in a way that satisfies a particular universality condition. In the case of limits and colimits, the universality conditions satisfied are those that make the object either the most specific (or minimal) in the case of limits, or the most general (or maximal) in the case of colimits, such that it commutes with a given diagram in the category. The definitions of limits and colimits are provided in Definition 8 and 9, respectively.

Because colimits, which are dual to limits, play a central role in the framework developed in

this dissertation, we will define both concepts but focus primarily on colimits in this exposition. Although limits and colimits are first introduced in their general form, we will also give particular attention to a specific type of colimit—the *pushout*—in this section.

Definition 8 (Limit)**.** Let $D : \mathsf{J} \to \mathsf{C}$ be a diagram.

A **cone** over D is an object $X \in \mathsf{C}$ and a family of morphisms in C, $(X \xrightarrow{f_i} D(i))_{i \in \mathsf{J}}$, such that the triangle:



commutes for every morphism $u : I \to K$ in J.

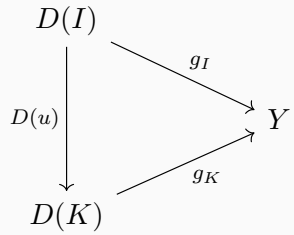
A **limit** of D is a cone $(L \xrightarrow{\pi_i} D(i))_{i \in \mathsf{J}}$ over D with the property that for any cone over D, as above, there exists a unique morphism $f_! : X \to L$ such that $\pi_i \circ f_! = f_i$ for all $i \in \mathsf{J}$.

A limit is a concept that describes how to combine objects, $D(i)$ where $i \in J$, in a way that preserves their relationship. It identifies a single object, L, that minimally encodes all the relationships among the objects in a coherent, consistent manner.

As mentioned, colimits are defined dually to limits, which means the morphisms forming the cone are reversed to form a *cocone*.

Definition 9 (Colimit)**.** Let $D : \mathsf{J} \to \mathsf{C}$ be a diagram.

A **cocone** under D is an object $Y \in \mathsf{C}$ and a family of arrows in C, $(D(i) \xrightarrow{g_i} Y)_{i \in \mathsf{J}}$ such that the triangle:



commutes for every morphism $u : I \to K$ in J. A **colimit** of D is a cocone $(D(i) \xrightarrow{\iota_i} C)_{i \in \mathsf{J}}$ under D with the property that for any cocone under D, as above, there exists a unique morphism $g_! : C \to Y$ such that $g_! \circ \iota_i = g_i$ for all $i \in \mathsf{J}$.

A **colimit** is a concept that describes how to merge objects, $D(i)$ where $i \in J$, in a way that captures how they come together. It identifies a single object, C, that combines all the relationships among the objects in the most inclusive, consistent way.

Colimits (and limits) can be computed over a variety of diagrams. Of particular relevance to this dissertation, is the colimit over a span. A colimit of a span is called a **pushout** which forms a pushout square. Formally:

Definition 10 (Pushout)**.** Let $\mathsf{D} : \mathsf{J} \to \mathsf{C}$ be a diagram, where J is an index category with the shape (a span):

$$\begin{array}{ccc} Z & \xrightarrow{g} & Y \\ {\scriptstyle f}\downarrow & & \\ X & & \end{array}$$

A **pushout** of this diagram consists of an object $P \in \mathsf{C}$ along with two morphisms $p_X : X \to P$ and $p_Y : Y \to P$ such that the following diagram commutes:

$$\begin{array}{ccc} Z & \xrightarrow{g} & Y \\ {\scriptstyle f}\downarrow & \ulcorner & \downarrow{\scriptstyle p_Y} \\ X & \xrightarrow[p_X]{} & P \end{array}$$

The pushout is a colimit of a span which means it satisfies the following universal property—for any other object $Q \in \mathsf{C}$ with morphisms $q_X : X \to Q$ and $q_Y : Y \to Q$ such that the diagram:

$$\begin{array}{ccc} Z & \xrightarrow{g} & Y \\ {\scriptstyle f}\downarrow & & \downarrow{\scriptstyle q_Y} \\ X & \xrightarrow[q_X]{} & Q \end{array}$$

commutes, there exists a unique morphism $u_! : P \to Q$ such that $u_! \circ p_X = q_X$ and $u_! \circ p_Y = q_Y$.

To state this more concretely, in the category Set, a pushout can be thought of as the disjoint union of two sets that are joined along a common subset represented by the apex of the span, as was foreshadowed in Section 3.1.1.1. This means that given two sets X and Y with a common subset Z and functions $f : Z \to X$ and $g : Z \to Y$, the pushout forms the union of X and Y while identifying elements in X and Y that correspond to the same element in Z. This construction, in essence, glues the sets together along Z. An example demonstrating this for specific objects in Set is presented in Example 11.

Example 11 (Pushout in Set)**.** The span is the same as in Example 9. The pushout of this span in Set is:

$$P = \{c, d = a, e = b\}.$$

In the pushout set P, the element $d \in X$ is identified with $a \in Y$, and $e \in X$ is identified with $b \in Y$. The element $c \in X$ remains distinct because it is not related to any element in Y.

It is also possible to take the colimit over diagrams involving more sophisticated objects, like representable functors.

Example 12 (Colimit of Representable Functors in a Small Category, C)**.** Let C be a small category with two objects, A and B, and a single morphism:

$$f : A \to B$$

along with identity morphisms 1_A and 1_B.

The representable functor, $\mathrm{Hom}_{\mathsf{C}}(A, -)$, is defined according to the following object and morphism maps.

Object map. The functor, $\mathrm{Hom}_{\mathsf{C}}(A, -)$, maps each object $X \in \mathsf{C}$ to the set representing morphisms from A to X:

$$\mathrm{Hom}_{\mathsf{C}}(A, A) = \{1_A\}, \quad \mathrm{Hom}_{\mathsf{C}}(A, B) = \{f\}$$

Morphism map. The functor, $\mathrm{Hom}_{\mathsf{C}}(A, -)$, maps the morphism $f \in C$:

$$\mathrm{Hom}_{\mathsf{C}}(A, -)(f) : \mathrm{Hom}_{\mathsf{C}}(A, A) \to \mathrm{Hom}_{\mathsf{C}}(A, B)$$

The representable functor, $\mathrm{Hom}_{\mathsf{C}}(B, -)$, is defined according to the following object and morphism maps.

Object map. The functor, $\mathrm{Hom}_{\mathsf{C}}(B, -)$, maps each object $X \in \mathsf{C}$ to the set representing morphisms from B to X:

$$\mathrm{Hom}_{\mathsf{C}}(B, B) = \{1_B\}, \quad \mathrm{Hom}_{\mathsf{C}}(B, A) = \emptyset$$

Morphism map. The functor, $\mathrm{Hom}_{\mathsf{C}}(B, -)$, maps the morphism $f \in C$:

$$\mathrm{Hom}_{\mathsf{C}}(B, -)(f) : \mathrm{Hom}_{\mathsf{C}}(B, A) \to \mathrm{Hom}_{\mathsf{C}}(B, B)$$

Natural transformation. The morphism $f : A \to B$ induces a natural transformation, α, between these representable functors, namely:

$$\begin{array}{ccc} \mathrm{Hom}_{\mathsf{C}}(A, -)(A) & \xrightarrow{\mathrm{Hom}_{\mathsf{C}}(A,-)(f)} & \mathrm{Hom}_{\mathsf{C}}(A, -)(B) \\ \downarrow{\scriptstyle \alpha_A} & & \downarrow{\scriptstyle \alpha_B} \\ \mathrm{Hom}_{\mathsf{C}}(B, -)(A) & \xrightarrow[\mathrm{Hom}_{\mathsf{C}}(B,-)(f)]{} & \mathrm{Hom}_{\mathsf{C}}(B, -)(B) \end{array}$$

The components, α_A and α_B, are the following maps:

$$\alpha_A : \{1_A\} \mapsto \emptyset$$

$$\alpha_B : \{f\} \mapsto \{1_B\}$$

The colimit of this diagram of representable functors is the hom-functor Y defined by the following object and morphism maps.

Object map. The functor, Y, maps each object $X \in \mathsf{C}$ accordingly:

$$Y(A) = \emptyset, \quad Y(B) = \{\star\}$$

Because α_A identifies $\{1_A\}$ with the empty set $\emptyset$, this causes $\{1_A\}$ to be viewed as equivalent to $\emptyset$, or nothing. Because α_B identifies $\{f\}$ with and $\{1_B\}$, $Y(B)$ consists of a single element, which we denote as $\star$, representing their equivalence class.

Morphism map. The functor, Y, maps the morphism $f \in C$ to:

$$Y(id_A) : \emptyset \to \emptyset$$

$$Y(id_B) : \{\star\} \to \{\star\}$$

$$Y(f) : \emptyset \to \{\star\}$$

Notice that Y is representable because it is naturally isomorphic to $\mathrm{Hom}_{\mathsf{C}}(B, -)$.

In the colimit of the representable functors shown in Example 12, Y acts as a compressed or quotient view of C in which the distinction between the morphisms f and 1_B is removed giving a simplified view of the relationships in C. This has computational advantages as it allows for a compressed view of the structure formed by a category. When the category encodes types in an ontology, for example, this approach allows relationships between types to be represented in a minimal form.

It is important to note that not all diagrams have limits and colimits in every category [101]. A category that has limits for all diagrams is called *complete*, while a category that has colimits

for all diagrams is called *cocomplete*. If a category only has limits or colimits for some diagrams but not all, then it is referred to as *incomplete* and *non-cocomplete*, respectively. For example, the category, Set, is complete and cocomplete, whereas the category of finite sets, FinSet, is complete and non-cocomplete [101].

3.3 Constructions

In category theory, constructions provide systematic methods for creating new categories, functors, or other categorical objects that reveal or exploit underlying structure from existing ones.

The **category of elements construction**, [101], is an example of a categorical construction that transforms a functor $F : \mathsf{C} \to \mathsf{Set}$ into a single category. Formally:

Definition 11 (Category of Elements)**.** Let $F : \mathsf{C} \to \mathsf{Set}$ be a functor. The **category of elements** of F, denoted $\int F$, is the category defined as follows:

- The objects of $\int F$ are pairs (C, X), where $C \in \mathrm{Ob}(\mathsf{C})$ and $X \in F(C)$.
- The morphisms of $\int F$ are $(f, X) : (C, X) \to (D, Y)$, where $f : C \to D$ is a morphism in C and $F(f)(X) = Y$.

Given two composable morphisms:

$$(f, X) : (C, X) \to (D, Y), \quad (g, Y) : (D, Y) \to (E, Z) \quad \in \int F$$

their composition $(g, Y) \circ (f, X)$ is defined as $(g \circ f, X) : (C, X) \to (E, Z)$[a].

The identity morphism for an object (C, X) in $\int F$ is $(1_C, X)$, where 1_C is the identity morphism on C in C.

[a] This composition works because $F(g \circ f)(X) = F(g)(F(f)(X)) = F(g)(Y) = Z$ which satisfies the conditions for morphisms in $\int F$.

To help illustrate this idea, let us consider an example of a category of elements construction for a simple C-Set functor.

Example 13 (Category of Elements for $F : \mathsf{C} \to \mathsf{Set}$)**.** Consider the small category, C.

$$A \xrightarrow{f} B$$

The identify morphisms 1_A and 1_B are also present but not drawn.

Let $F : \mathsf{C} \to \mathsf{Set}$ be a functor defined by the following object and morphism assignments.

Object map. The functor, F, maps the objects $X \in \mathsf{C}$ accordingly:

$$F(A) = \{a, b\}, \quad F(B) = \{c\}$$

Morphism map. The functor, F, maps the non-identity morphism, $f \in \mathsf{C}$ accordingly:

$$F(f)(a) = c, \quad F(f)(b) = c$$

The identity morphisms, 1_A and 1_B, are identity functions.

The category of elements construction constructs a new category, $\int F$, based on F.

Objects. The objects of $\int F$ are:

$$\int F(a) = (A, a), \quad \int F(b) = (A, b), \quad \int F(c) = (B, c)$$

Morphisms. The morphisms of $\int F$ are:

$F(f)(a) : (X, a) \to (Y, c)$ $\quad$ $F(id_X)(a) : (X, a) \to (X, a)$ $\quad$ $F(id_Y)(c) : (X, c) \to (X, c)$

$F(f)(b) : (X, b) \to (Y, c)$ $\quad$ $F(id_X)(b) : (X, b) \to (X, b)$

This construction provides a way to reveal the structure of a set-valued functor by organizing its elements and the point-wise maps between them as objects and morphisms in a new category.

3.4 Other Categories

The algebraic theory of categories can be extended to include additional structure, such as in monoidal categories and their extensions. A **monoidal category** [39] is a category equipped with a monoidal product that allows objects to be combined in a way that mimics familiar operations like multiplication or parallel composition.

For instance, monoidal categories have been applied to model the composition of quan-

tum states and processes [33], and as a framework for understanding parallelism and concurrency in programming [104, 105]. Extensions of monoidal categories, such as braided and symmetric monoidal categories, add further structure that models commutativity or more intricate interactions. Formally, a monoidal category is defined:

Definition 12 (Monoidal Category)**.** A **monoidal category** is a category, M, consists of:

- A bifunctor $\otimes : \mathsf{M} \times \mathsf{M} \to \mathsf{M}$, called the **monoidal product**,
- An object $1 \in \mathsf{M}$, called the **unit object**,
- A natural isomorphism $a_{X,Y,Z} : (X \otimes Y) \otimes Z \to X \otimes (Y \otimes Z)$, called the **associator**,
- A natural isomorphism $\lambda_X : 1 \otimes X \to X$, called the **left unitor** and a natural isomorphism $\rho_X : X \otimes 1 \to X$, called the **right unitor**

such that the *pentagon identity*—that enforces that all ways of associating a fourfold monoidal product are equivalent—and the *triangle identity*—that enforces compatibility between the associator and the unitors—are satisfied.

A braided monoidal category extends the definition to include an isomorphism called a *braid* [39]. Here, we provide an abbreviated definition of a braided monoidal category:

Definition 13 (Braided Monoidal Category)**.** A **braided monoidal category** is a monoidal category M equipped with a natural isomorphism:

- $B_{X,Y} : X \otimes Y \to Y \otimes X$ called the **braiding**.

such that *hexagon identities*—that enforce the compatibility of braiding with the associator—and the *inverse braiding condition*—that guarantees that applying the braiding in both directions $B_{X,Y}$ and $B_{Y,X}$ results in the identity on $X \otimes Y$—are satisfied.

A **symmetric monoidal category (SMC)** further extends the definition of a braided monoidal category by requiring the braiding to be symmetric, meaning that swapping two objects twice

results in the identity transformation. We formally define an SMC as:

Definition 14 (Symmetric Monoidal Category (SMC))**.** A **symmetric monoidal category** is a braided monoidal category M where the braiding is symmetric. This means that the braiding natural isomorphism $B_{X,Y} : X \otimes Y \to Y \otimes X$ satisfies the condition:

$$B_{Y,X} \circ B_{X,Y} = 1_{X \otimes Y}$$

for all objects X and Y in M.

Given that all monoidal categories are categories, there can exist functors between these categories, namely monoidal functors. As with regular functors, monoidal functors preserve composition, however, they also preserve the other structures present within the algebraic theory of a monoidal category. For monoidal functors between SMCs, they must preserve the monoidal product and its symmetry [103]. An abbreviated definition is provided below:

Definition 15 (Monoidal functor between SMCs)**.** Let C and D be SMCs. A **monoidal functor** $F : \mathsf{C} \to \mathsf{D}$ is a functor equipped with:

- A natural isomorphism $\phi_{X,Y} : F(X) \otimes_{\mathsf{D}} F(Y) \cong F(X \otimes_{\mathsf{C}} Y)$ for all objects $X, Y \in \mathsf{C}$,
- A morphism $\phi_I : J \to F(I) \in \mathsf{D}$ that preserves the unit object $I \in \mathsf{C}$,

such that the following laws are satisfied:

(i) the *associativity compatibility law* that enforces that the functor respects the associativity of the monoidal product in both categories,

(ii) the *unit compatibility law* that enforces that the functor maps the unit object of the source category to the unit object of the target category in a way that maintains the identity properties of these objects, and

(iii) the *symmetry compatibility law* that enforces that the symmetry of the monoidal product is preserved under the functor

For a more detailed definition of each of the laws, we direct the reader to Chapter 11 in [103].

As we will see in this dissertation, SMC are particularly useful for modeling systems where objects interact through combination, such as in the semantics of concurrent actions in planning.

3.5 Applications of Category Theory

Category theory has seen many applications since its inception. For example, category theory has been used to understand information structures [31], automate information fusion systems [32], represent quantum processes [33, 34], synchronize models in systems engineering [35–37], connect mathematics and physics [106], model data analysis and sensor networks [107,108], analyze visual representations [109], verify concurrent systems [110], and formalize network-style diagrammatic languages [111–113] to name a few.

Several mathematical inventions that have emerged as a result of these diverse applications serve as the foundation for this work. Here, we introduce each of these foundational concepts briefly, with detailed treatments provided in the following chapters. In particular, we will provide brief overviews of string diagrams, categorical databases, and functorial data migration.

3.5.1 String Diagrams

String diagrams and functors between them will be used as the mathematical tooling to enable sequential and hierarchical plan transfer. A diagrammatic representation of an SMC is called a **string diagram** [114], an example of which can be seen in Figure 3.1. This figure presents two graphical representations for the composition of morphisms $f : A \to B$ and $g : B \to C$. In Figure 3.1b, the lines are called **strings**, which symbolize identity morphisms (e.g., id_A). The rectangles, or boxes, symbolize non-identity morphisms. The junction where strings and boxes meet are realizations of composition between identity morphisms and non-identity morphisms. A



(a) Conventional syntax (b) String diagram syntax

Figure 3.1: Conventional versus string diagram syntax for composite morphisms.

common shorthand in string diagrams is to label each string according to its associated object rather than explicitly indicating the identity morphism. Because an SMC is a category, it also allows for morphisms that capture both sequential compositions and hierarchical structures, as we will explore in Chapters 4 and 5.

3.5.2 Categorical Databases

In this dissertation, categorical databases provide a structured, formal means of representing and managing knowledge needed for planning. By organizing data in a way that reflects types and relationships within a planning domain, categorical databases allow for flexible, precise data manipulation and integration across different contexts.

3.5.2.1 Ologs

At the core of categorical databases lies a foundation built on categorical structures that facilitate organized and meaningful knowledge representation. This foundation traces back to the concept of *ologs* (ontology logs), which were introduced in 2011 by David I. Spivak and Robert E. Kent [8]. Ologs represent knowledge in a categorical manner, where objects and morphisms capture types of things and their predicate relationships. A definition of ologs is provided in Definition 16.

Definition 16 (Ologs)**.** An olog is a category where:

- **Objects** represent types or classes of things.
- **Morphisms** represent predicate relationships or attributes between objects.

A schematic of the olog for the amino acid, arginine is shown in Figure 3.2. Here, the fact "arginine is an amino acid" is captured through the morphism: (arginine $\xrightarrow{\text{is}}$ an amino acid).

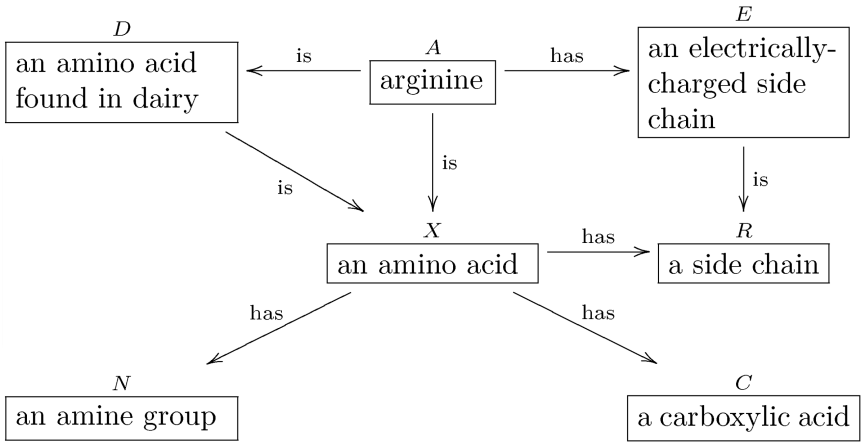


Figure 3.2: An olog representing the amino acid, arginine [8]

The ontology web language (OWL) [115] offers a similar ability to express schemas with constraints on classes and properties. However, ologs provide certain advantages towards expressiveness and interoperability with other categorical frameworks. The resource description framework (RDF) [116] is another language for representing knowledge using graphs. However, RDF lacks the ability to specify commutativity, a feature that ologs provide, allowing for a more concise and expressive representation of knowledge.

Ologs can serve as both descriptive and prescriptive database schemas, allowing them to represent existing data structures and to prescribe new ones. The categorical structure of ologs also enables functorial data migration, which we will introduce shortly, facilitating seamless transfer of data between different schemas. Within the context of categorical databases, we will refer to ologs as (database) **schemas** for the remainder of this dissertation.

3.5.2.2 C-set

Instance data can be assigned to schemas to form **categorical databases**. An assignment of instance data to a schema C is a functor $X : \mathsf{C} \to \mathsf{Set}$, which assigns a set to each type and a function to each relationship between types while preserving their composition. The functor X is referred to as a C-set or copresheaf on the category C [8]. For practical purposes, particularly in computing, it is often the case that the sets involved are finite [41]. In this dissertation, we will rely heavily on this formalism as part of our categorical planning representation language.

3.5.2.3 Functorial Data Migrations

Functorial data migrations are transformations between categorical databases that preserve the integrity of the data imposed by the schema [43] and will be the mathematical tool we will use for analogical plan transfer. Spivak identified three types of *migration functors* that can be used for transforming categorical databases: the pullback functor (Δ), the right pushforward functor (Π), and the left pushforward functor (Σ). Each migration functor is informed by a *translation functor* that maps from one schema to another. The Δ migration functor takes instance data from the target schema back to the source schema. The Π migration functor transforms instance data from the source schema to the target schema while preserving relationships (such as joins). The Σ migration functor also transforms instance data from the source schema to the target schema, but focuses on combining or aggregating data (such as unions). The functoriality of these transformations ensure that data relationships remain consistent according to the schema category throughout the process. In this dissertation, we will focus our attention on variants of pullback migration functors (Δ) as part of our analogical plan transfer approach. Further work is needed to assess the semantic relevance of joins and unions of instance data within the context of (analogical) plan transfer.

3.6 Summary

In this chapter, we have explored several key concepts and frameworks—SMCs, categorical databases, and functorial data migrations—that leverage the core ideas of categories, functors, and natural transformations to represent and manipulate complex structures. These constructions form the backbone of the theoretical tools essential for the development of this dissertation as we will see in the remaining chapters.

Chapter 4: Sequential Transfer using String Diagrams

Sequential transfer represents the simplest form of plan transfer, providing a straightforward way to reuse and combine existing plans in sequence without requiring extensive modification or adaptation. Its simplicity makes it valuable in scenarios where a plan needs to be extended or prepended to achieve new goals.

Sequential transfer composes a plan, π_1, with another plan, π_0, to form a new plan, π', within the same planning domain. A key aspect of sequential transfer is ensuring that the composite plan is a valid composition. This means that the resulting plan, π', consists of a sequence of actions from both π_0 and π_1, such that all action preconditions are satisfied. We provide a formal definition of sequential transfer in Definition 17.

Definition 17 (Sequential Transfer)**. Sequential transfer** is the composition of two or more task plans, each represented as a sequence of actions, with the same planning domain.

Let π_0 and π_1 be task plans, where each plan is an ordered sequence of actions $\pi_i = \langle a_{i1}, a_{i2}, \ldots, a_{in} \rangle$ for $i = 0, 1$. Sequential transfer creates a new task plan π' by composing π_0 and π_1 such that:

$$\pi' = \pi_1 \circ \pi_0$$

where $\circ$ denotes the composition operator.

The resulting plan π' is an ordered sequence of actions that first executes all actions in π_0 followed by all actions in π_1, namely: $\pi' = \langle a_{01}, a_{02}, \ldots, a_{0n}, a_{11}, a_{12}, \ldots, a_{1m} \rangle$.

Valid Composition. It is necessary to check that all action preconditions in π' are met by the cumulative effects of previous actions or the initial state. This condition is defined as:

$$\text{is\_valid\_sequence}(\pi_1 \circ \pi_0) = \text{True}$$

where the validation function verifies that the preconditions of each action in π' are satisfied.

This type of transfer can also be applied to situations where a new plan is inserted between two existing plans to bridge them into a coherent sequence. This approach is advantageous in scenarios where a robot has an established plan that it executes reliably, and that plan needs to be integrated with others as part of a solution to a subproblem within the same domain.

4.1 Overview

One of the primary shortcomings of sequential transfer, as discussed in Chapter 2.2.3, is the insufficient consideration of preconditions and effects across the entire plan during composition. To address this, we propose using monoidal categories and their graphical syntax—string diagrams—to model plans and validate their sequential composition. Monoidal categories provide a robust framework for describing the compositional structure of task plans, while string diagrams offer a clear, visual representation that simplifies tracking the relationships between actions, preconditions, and effects.

In the remainder of this chapter, we delve into the fundamentals of string diagrams and an interpretation of them within AI planning, specifically for the Planning Domain Definition Language (PDDL) [61]. An overview of PDDL and string diagrams is provided in Section 4.2. In Section 4.4, we introduce how string diagrams can be used to represent domains and plans within PDDL. Section 4.5 presents a visual example for the Blocksworld domain and describe how we encode PDDL domains, problems, and plans into the string diagram representation. Finally, in Section 4.6, we discuss key observations and explore how this representation can further enhance

the explainability of plans and their sequential transfer.

4.2 The Planning Domain Definition Language (PDDL)

Before diving into the specifics of string diagrams, it is essential to ground our discussion in the planning language of PDDL, as it is commonly used to express tasks in robotics [18]. By understanding the syntax of PDDL, we can better appreciate how string diagrams provide an expressive and transparent mechanism for modeling the provenance of preconditions and effects in a plan. This section outlines the key components of PDDL that will later be captured in our categorical framework.

PDDL was introduced as an extension of STRIPS [60], making an explicit distinction between a **planning domain** and a **planning problem**, among other language features. A PDDL domain description consists of:

- A finite set of **predicates**, which define templates for facts used to represent the world state. For example, the predicate `(on ?x ?y)` provides a template for representing the fact of one object, `?x`, being on another object, `?y`.
- A finite set of **actions**, each specified with parameters, preconditions, and effects. **Preconditions** describe the conditions that must hold for the action to be applied, and **effects** describe how the world changes after the action is applied. **Parameters** are free variables within the preconditions and effects that generalize actions into action schemas, or **ungrounded actions**, which can later be instantiated to form **grounded actions**.

Preconditions, effects, and states are typically represented using a subset of first-order logic (see Section 2.4 in [23]). In first-order logic, a key feature for representing PDDL states is the use of conjunctions to specify facts that must hold simultaneously. PDDL also supports disjunctions, negations, and quantified expressions (introduced in later versions, such as PDDL 2.1 [117]),

though conjunctions remain the most commonly used feature of the language.

A PDDL **plan** consists of a list of grounded actions that are specified using grounded predicates, or **literals**, as part of their preconditions and effects. In addition to grounding actions, literals are used to instantiate the planning problem. A PDDL problem description consists of:

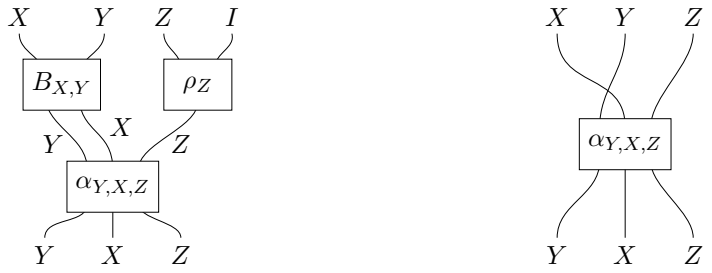
- The **initial state** which is a set of literals describing the facts about the world that are true at the start of the planning process.
- The **goal state** which is a set of literals describing the facts that must be true after the plan has been executed.

We now turn our attention to a category-theoretic framework that can model these concepts. While PDDL provides a structured syntax for specifying plans, category theory offers a mathematically rigorous semantics useful for sreasoning about the sequential composition of plans.

4.3 String Diagrams

Recall from Section 3.5.1 that symmetric monoidal categories (SMCs) can be expressed using a graphical syntax, called **string diagrams**, for illustrating morphisms and their compositions [118]. In this graphical syntax, morphisms are expressed as boxes, while identity morphisms on objects are expressed as strings. Multiple boxes and strings can occupy the same horizontal plane of a string diagram, which can be interpreted as a monoidal product found in SMCs. For instance, horizontally adjacent strings represent the monoidal product of the identity morphisms of the corresponding objects, such as $X \otimes Y$, which itself is an object in the SMC. Additionally, as with any monoidal category (see Definition 12), there exists unitor isomorphisms, namely $\lambda_X : 1 \otimes X \to X$ (left unitor) and $\rho_X : X \otimes 1 \to X$ (right unitor). SMCs also include a braiding isomorphism, $B_{X,Y} : X \otimes Y \to Y \otimes X$.

There are a few conventions for reading string diagrams [118]. In this dissertation, diagrams



(a) Explicit (b) Shorthand

Figure 4.1: The braiding and identity isomorphisms of string diagrams.

are read from top to bottom, with the identity morphisms of domain objects represented by the strings entering the boxes from above and the identity morphisms of codomain objects represented by the strings exiting the boxes from below. Since braiding isomorphisms and unitor isomorphisms are both morphisms, they can also be represented as boxes, as seen in Figure 4.1a. However, a common convention is to replace boxes representing braids and morphisms with crossing strings for braids or simply a string for unitor isomorphisms, as seen in Figure 4.1b.

The graphical syntax can also be captured in a linear notation. For example, the linear notation corresponding to the diagram in Figure 4.1 is:

$$(id_X \otimes id_Y \otimes id_Z \otimes id_I); (B_{X,Y} \otimes \rho_Z); (id_Y \otimes id_X \otimes id_Z); (\alpha_{Y,X,Z}); (id_Y \otimes id_X \otimes id_Z)$$

This notation clearly represents the morphisms and the order in which they are composed. For example, if a horizontal line is drawn such that it cuts the diagram at the top, the strings intersected are X, Y, Z, and I, corresponding to the linear expression $(id_X \otimes id_Y \otimes id_Z \otimes id_I)$. Similarly, if a horizontal line is drawn to intersect the boxes below those strings, it would cross $B_{X,Y}$ and ρ_Z, corresponding to the linear expression $(B_{X,Y} \otimes \rho_Z)$.

Note that the composition order, denoted by ; in the linear notation, mirrors the way the string diagram is drawn from top to bottom. This notation is often referred to as the *diagrammatic convention* for morphism composition. Alternatively, the *functional convention* for composition, denoted by $\circ$, reverses the order relative to the diagrammatic convention, aligning with the typical

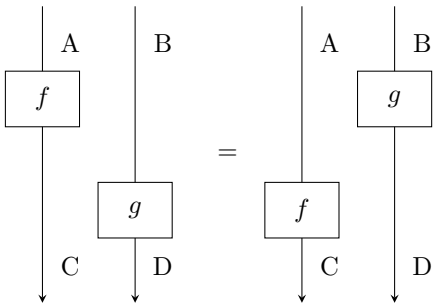


Figure 4.2: Planar deformation in string diagrams.

function composition order in mathematics. Throughout this dissertation, we will adhere to the diagrammatic convention when expressing the linear syntax of string diagrams.

4.3.0.1 Invariance under Planar Deformation

A noteworthy consequence of the monoidal product and its properties is the equivalence between string diagrams under planar deformation [118]. For example, consider the equivalence expressed in Equation 4.1 and its corresponding graphical representation shown in Figure 4.2.

$$(f \otimes id_B); (id_C \otimes g) = (id_A \otimes g); (f \otimes id_D) \tag{4.1}$$

This equivalence allows boxes to slide past each other along strings, effectively rearranging compositions while preserving the underlying structure. This property is particularly useful when representing alternative sequences of compositions. As a result, this representation has the potential to encode *partially ordered plans* [23] by allowing the rearrangement of actions without violating the overall structure or dependencies. This flexibility is essential for modeling scenarios where the precise order of some actions can vary without affecting the final outcome. The connection between this notation and partially ordered plans is left for future work.

4.4 Symmetric Monoidal Category, PDDLPlan

In this section, we describe how key elements of the PDDL domain, problem, and plan are translated into string diagram notation. We construct a symmetric monoidal category (SMC) called PDDLPlan. From a categorical perspective, each grounded PDDL action is represented as a morphism in PDDLPlan, mapping a conjunction of precondition literals to a conjunction of effect literals. Literals are objects, and the monoidal product corresponds to the logical conjunction of these literals. Formally:

Definition 18 (PDDLPlan)**.** The category PDDLPlan is an SMC whose:

- **Objects** are PDDL literals, representing properties of the world state.
- **Morphisms** are grounded PDDL actions, representing state transitions from a conjunction of precondition literals to a conjunction of effect literals.
- **Composition** is defined as the sequential application of actions, where two morphisms $f : X \to Y$ and $g : Y \to Z$ compose to yield a morphism $g \circ f : X \to Z$, provided that the effect literals of the first action match the precondition literals of the second action.
- **Monoidal product** is defined as the logical conjunction of literals at the object level, with the monoidal unit as the empty conjunction (true).

The remainder of this section discusses how each syntactical feature of PDDL domain and problem files can be interpreted from this categorical point of view.

A PDDL domain file consists of the planning domain name, predicates, and actions. This specification provides the domain and codomain signatures for morphisms in our category. We restrict the PDDL features to the STRIPS requirement[1], which includes the conjunction, addition, and deletion of literals. This requirement excludes the use of conditional effects, disjunctions, and

[1]The STRIPS requirement is often denoted as `(:requirements :strips)` in a PDDL domain file.

negation in the preconditions.

As discussed in Section 4.2, actions are operator data models that describe the parameters, preconditions, and effects of a given action, denoted by the `:action` token in a PDDL domain file. Syntactically, preconditions and effects and their conjunctions are denoted by parentheses separating out the individual literals and an `AND` term joining the predicates. The identity morphism for each literal in the precondition forms a domain (input) string, and the identity morphism for each literal in the effects forms a codomain (output) string.

In the STRIPS requirement, effect literals can have a negated form, which in PDDL is denoted with a `NOT` prefix. For example, `(ontable A)` and `(NOT (ontable A))` are positive and negative forms of the literal `(ontable A)`, respectively. In $catPDDLPlan$, negated literals are considered distinct from their positive counterparts. Because `(ontable A)` and `(NOT (ontable A))` are distinct objects, the relationship between them through logical negation is not formally captured.

The PDDL problem file specifies the initial and goal states of the planning problem. The initial state, denoted by the `:init` token in the problem file, is a conjunction of literals that are true at the start of the planning problem. Each literal in the initial state is an object in PDDLPlan. The identity morphisms of these literals form the top-most strings of the diagram. Other literals may be horizontally composed with the initial state if an action later in the plan depends on a literal that is not explicitly stated initial state.

The goal state, denoted by the `:goal` token, is a conjunction of literals that *must* be true at the end of the planning problem. Each literal in the goal state is also an object in PDDLPlan and their identity morphisms are represented as their own strings in the diagram. The monoidal product of these strings forms the bottom-most strings in the diagram. Other literals may be horizontally composed with the goal state if an earlier action in the plan generates a literal that

is not explicitly required by the goal state.

4.4.1 Building a String Diagram in PDDLPlan

Recall that a plan is an ordered sequence of grounded actions (Definition 1). In PDDLPlan, each grounded action of a plan can be converted into a morphism. These morphisms are composed to generate a fully connected string diagram. As part of this dissertation, a naive chaining algorithm is implemented to infer valid monoidal products and sequential compositions. This algorithm identifies scenarios where directly adjacent morphisms are incompatible based on their preconditions and effects. The primary objective of this algorithm is to ensure compatibility between the domain and codomain strings of adjacent morphisms in order to achieve valid composition.

As part of this algorithm, a conceptual abstraction called (horizontal) **slices** is introduced. Each slice is an ordered list of morphisms, including identities and braids, that are composed in parallel using $\otimes$, from left to right. For valid sequential composition, it is crucial that the domain and codomain of each slice align, such that the codomain objects of the current slice, i, and the domain objects of the next slice, $(i+1)$, are exactly equal.

The algorithm proceeds through four main steps:

1. *Backward pass*: This step weaves the domain (input) strings of each morphism from the goal state to the initial state (bottom-up), supplementing horizontal slices with strings where necessary.
2. *Forward pass*: This step weaves the codomain (output) strings of each morphism from the initial state to the goal state (top-down), again supplementing horizontal slices with strings as needed.
3. *Add braids*: This step injects braids, that swap strings, where the horizontal order of strings is incompatible with the composition using a bubble sort algorithm.

4. *Compose*: This step chains together the slices from top to bottom, constructing the complete string diagram.

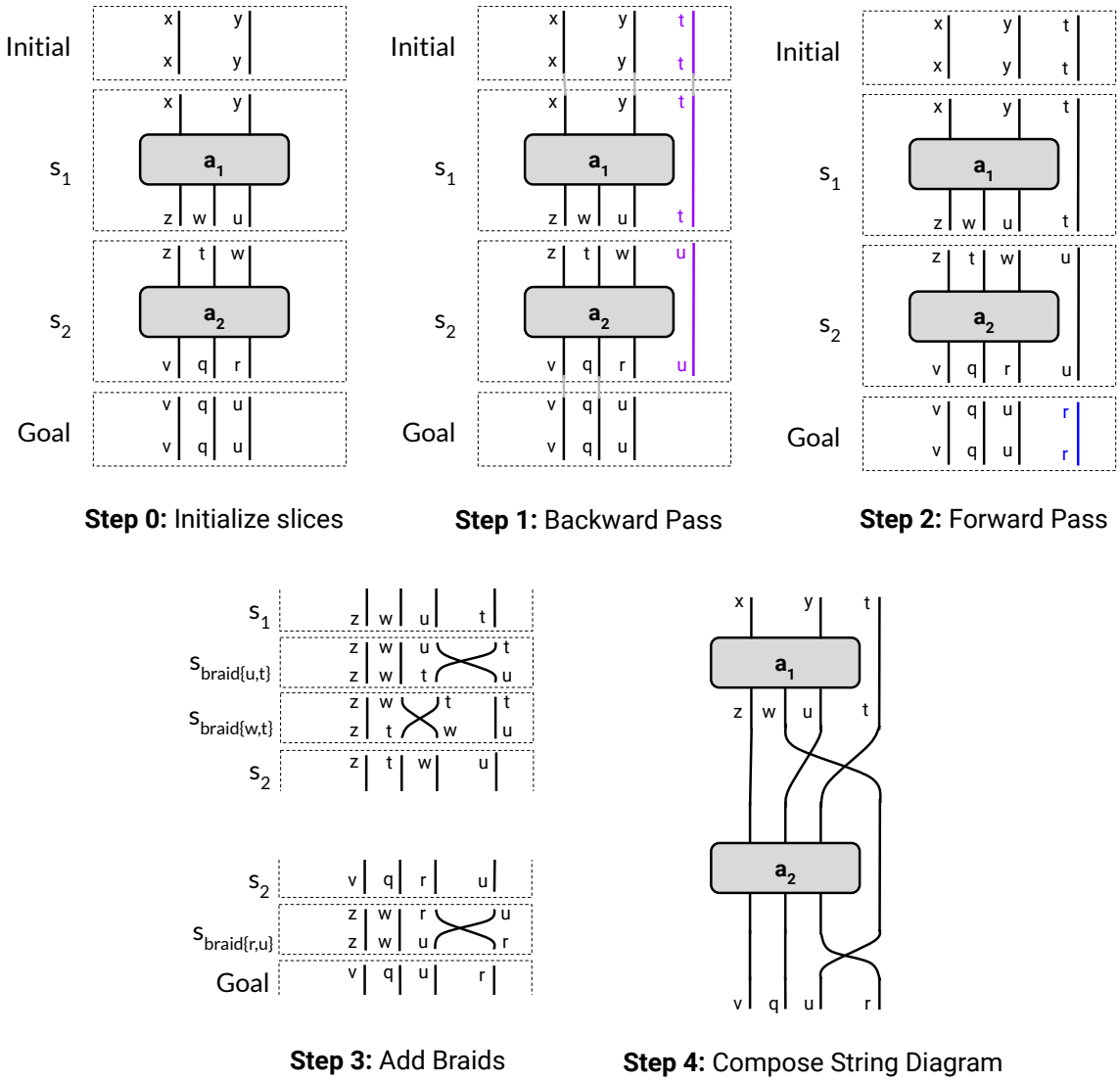


Figure 4.3: Illustration of algorithm for building a string diagram.

An illustration of each step of the algorithm is shown in Figure 4.3. The pseudocode for constructing the string diagram is outlined in Algorithm 1.

A byproduct of this chaining algorithm is the propagation of literals both upstream and downstream, revealing literals that are implicitly instantiated according to the plan. This propaga-

Algorithm 1: Build String Diagram

Input: Plan `plan`, Problem `problem`, Domain `domain`

Output: String Diagram `string_diagram`

`actions` ← Convert plan into action morphisms;
`n_actions` ← Get number of `actions`;
`init, goal` ← Convert the initial and goal states of the PDDL problem into slices;
Backward Pass*: Weave input strings from the goal state upwards through the plan*;
Initialize the goal state as the current slice as `curr_slice` ← `goal`;
Initialize the list of slices `slices` ← [`goal`];
for $i \leftarrow [n\_actions : -1 : -1]$ **do**
    if $i < 0$ **then**
        `above_slice` ← `init`;
    else
        `above_slice` ← `actions[i]`;
    `above_slice` ← Append new input strings from `curr_slice` into `above_slice`;
    `curr_slice` ← `above_slice`;
    Push `curr_slice` to the beginning of `slices`;
Forward Pass*: Weave output strings from the initial state downwards*;
Initialize the initial state as the current slice as `curr_slice` ← `init`;
for $i \leftarrow [1 : 1 : length(slices)]$ **do**
    `below_slice` ← `slices[i]`;
    `below_slice` ← Append new output strings from `curr_slice` into `below_slice`;
    `curr_slice` ← `below_slice`;
Add Braids*: Introduce braid morphisms to resolve monoidal product ordering mismatches between slices*;
`slices` ← Apply `add_braids`() to ensure consistent ordering of strings between slices;
Compose String Diagram*: Combine all slices into a single string diagram representation*;
Initialize the composite diagram `string_diagram` ← `nothing`;
for *each* `slice` *in* `slices` **do**
    if `string_diagram` == `nothing` **then**
        `string_diagram` ← `slice`;
    else
        try
            `string_diagram` ← `compose(string_diagram, slice)`;
        catch
            return `Fail`;
return `string_diagram`;

tion ensures that both explicit and implicit preconditions and effects are accounted for throughout the entire execution sequence. Since a single action can be considered the most trivial form of a plan, it is easy to see how this method can be applied to sequentially compose entire plans.

Complexity. The complexity of the algorithm is $O(n \cdot k^2)$, where n is the number of morphisms (actions) in the plan and k is the maximum number of strings (or literals) per morphism. The backward and forward passes both contribute $O(n \cdot k)$ to the complexity, as they process the morphisms and literals sequentially. However, the most computationally expensive operation comes from adding braids. Adding braids makes use of a bubble sort algorithm to reorder strings within each slice, resulting in a complexity of $O(k^2)$ for each slice. When applied to all slices, this leads to an overall complexity of $O(n \cdot k^2)$, indicating that the algorithm's performance is quadratic in the number of literals and linear in the number of morphisms. This algorithm and implementation[2] is an original contribution of this dissertation.

4.5 Example: Blocksworld

In this section, we will walk through an example of translating actions and plans within the PDDL Blocksworld[3] domain into string diagrams. Here, we demonstrate how the proposed algorithm verifies that the actions within the plan are composed correctly and show how the algorithm reveals the implicit preconditions and effects necessary for composing with other plans.

4.5.1 Domain

The Blocksworld domain provides a scenario where `blocks` are placed on a table, denoted by the predicate `(ontable ?x)`. The objective is to stack the blocks according to the configuration described by the goal state. Only one block can be placed on top of another block at a time. The

[2] https://github.com/aaguinal/string-diagram-viewer
[3] https://github.com/pellierd/pddl4j/tree/master/pddl/blocksworld

presence of a block on another is denoted as `(on ?x ?y)`. If a block is not underneath another block, it is considered clear, denoted by the predicate `(clear ?x)`. A hand is used to manipulate the blocks, and it can either be empty, represented by `(handempty)`, or holding a block, represented by `(holding ?x)`.

The domain consists of four actions: `pick-up`, `put-down`, `stack`, and `unstack`. For this example, the plan uses grounded versions of `pick-up` and `stack`.

The `pick-up` action requires that the block is clear `(clear ?x)`, on the table `(ontable ?x)`, and the hand is empty `(handempty)`. Executing this action changes the world state so that the block is no longer on the table `(NOT (ontable ?x))`, is no longer clear `(NOT (clear ?x))`, the hand is no longer empty `(NOT (handempty))`, and the hand is holding the block `(holding ?x)`.

The `stack` action requires that the hand is holding a block `(holding ?x)` and that another block `(clear ?y)` is clear. Executing this action changes the world state so that the hand is no longer holding the block `(NOT (holding ?x))`, the other block is no longer clear `(clear ?y)`, the hand is empty `(handempty)`, and the block is stacked on top of the other block `(on ?x ?y)`.

4.5.2 Problem

In this example, the problem initializes three blocks as objects (`a`, `b`, and `c`) and states that all the blocks are clear, on the table, and that the hand is empty. The goal is to have block `c` on top of block `b`, and block `b` on top of block `a`. In PDDL, this is specified as:

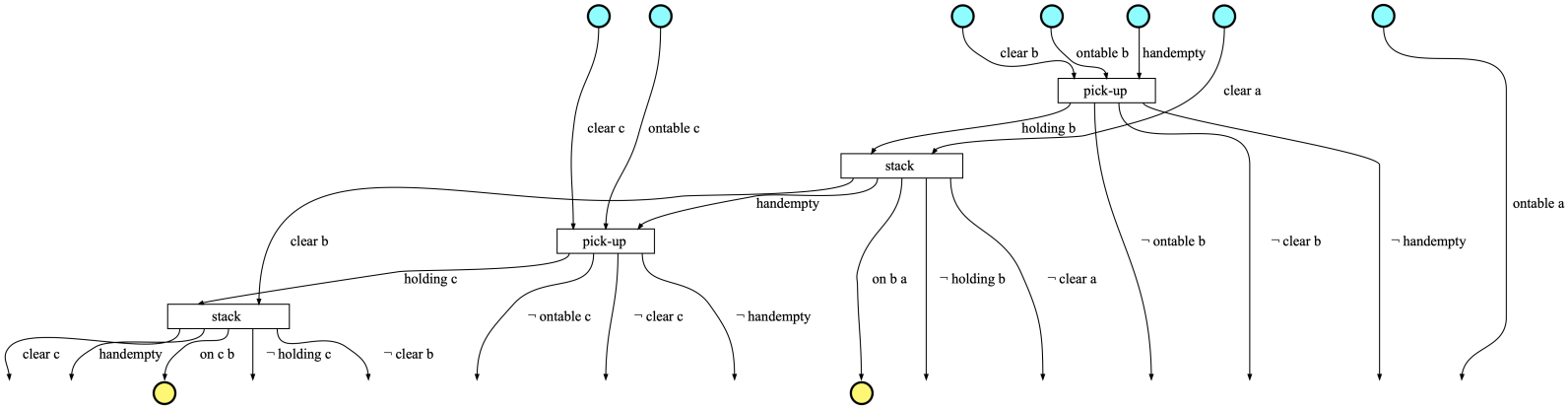
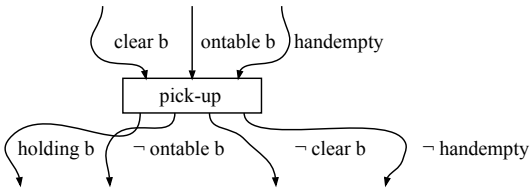


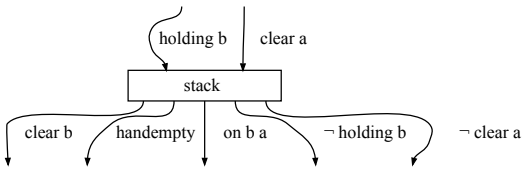
Figure 4.4: String diagram for a plan in the Blocksworld domain.

State	*Part of World State*	*Action*
Initial	**clear c**, **ontable c**, **clear b**, **ontable b**, **handempty**, **clear a**, **ontable a**	
Step 1	clear c, ontable c, **clear b**, **ontable b**, **handempty**, clear a, ontable a	pick-up
Step 2	clear c, ontable c, **holding b**, **clear a**, ¬ontable b,¬clear b, ¬handempty, ontable a	stack
Step 3	clear b, **clear c**, **ontable c**, **handempty**, on b a, ¬holding b, ¬clear a, ¬ontable b, ¬clear b, ¬handempty, ontable a	pick-up
Step 4	**holding c**, **clear b**, ¬ontable c, ¬clear c, ¬handempty, on b a, ¬holding b, ¬clear a, ¬ontable b, ¬clear b, ¬handempty, ontable a	stack
Goal	clear c, handempty, **on c b**, ¬holding c, ¬clear b, ¬ontable c, ¬clear c, ¬handempty, **on b a**, ¬holding b, ¬clear a, ¬ontable b, ¬clear b, ¬handempty, ontable a	

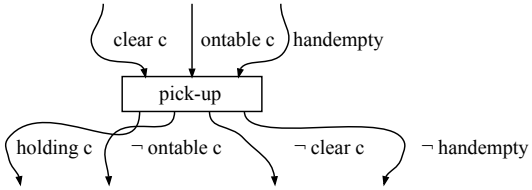
Table 4.1: World states in terms of explicitly (**bold**) and implicitly propagated literals.



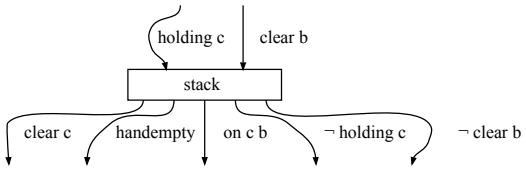
(a) First step.



(b) Second step.



(c) Third step.



(d) Fourth step.

Figure 4.5: String diagrams corresponding to actions in the Blocksworld plan.

```
(define (problem BLOCKS-3-0)
(:domain BLOCKS)
(:objects a b c)
(:init (clear c) (clear a) (clear b) (ontable c) (ontable a) (ontable b)
       (handempty))
(:goal (AND (on c b) (on b a))))
```

The heuristic search planner (HSP) [119] and the FF heuristic [120] from the PDDL4J toolkit [121] was used to solve for the plan. The planner's output was parsed to extract the plan while ignoring other outputs, such as cost and runtime. The parsed plan for this example is:

```
pick-up b
stack b a
pick-up c
stack c b
```

The individual steps in the plan can be understood as morphisms, and likewise can be represented as boxes in a string diagrams. Figure 4.5 illustrates the specific objects and morphisms from PDDLPlan.

A program[4], written in the Julia programming language, was developed as part of this dissertation to automatically generate string diagrams from PDDL plans. This program parses

[4] https://github.com/aaguinal/String-Diagrams-For-PDDL

PDDL domain, problem, and solution files, encoding their elements into categorical constructs that are JSON-serializable.

The fully composed string diagram containing all the steps of the Blocksworld plan is shown in Figure 4.4. The top of the diagram represents the strings corresponding to the literals of the initial state, along with assumptions introduced by the plan. The bottom of the diagram shows the strings representing the literals of the goal state, as well as the effects generated by the plan. The required literals at every step of the plan can be concluded from the linear notation that encodes the string diagram representation shown in Figure 4.4. One can verify this by reading the diagram from left to right at various horizontal positions and noting the strings encountered. The literals that satisfy preconditions, the initial state, or the goal state are highlighted in **bold text** in Table 4.1.

There are some noteworthy observations in this example that can be made from the string diagram representation of this plan. For example, by examining Figure 4.4 from top to bottom, we can observe that the first `stack` relies on `(holding b)`, which is an effect produced by the preceding `pick-up` action. This dependency highlights the sequential nature of the plan, where the success of each action is contingent on the effects of its predecessors. Additionally, the diagram shows how certain literals from the initial state, such as `(clear c)` and `(ontable c)`, are only referenced later in the plan during the second `pick-up` action, demonstrating how sequential transfer operates on both explicit and delayed preconditions.

By referencing Table 4.1, we can see that no new implicit assumptions are introduced into the plan, as indicated by the absence of unbolded literals in the **Initial** row. Furthermore, both Table 4.1 and Figure 4.4 illustrate the state of the world at each step of the plan, including literals like `(clear b)` and `(NOT (clear b))` in **Step 4**, which suggests the solver had to resolve contradictions during the plan's execution.

This analysis reveals how sequential transfer not only connects actions through explicit preconditions and effects but also exposes the implicit changes in the world that may not be evident from a simple step-by-step plan description. Moreover, it helps identify unanticipated literals that could cause errors, particularly under the closed world assumption [122].

4.6 Discussion

In this chapter, we introduced an approach using string diagrams to represent and validate the sequential composition of a PDDL plan. The string diagram representation not only shows which literals are initialized and satisfied throughout the plan but also reveals how actions are composed sequentially, revealing both explicit and implicit dependencies between them. This visualization helps determine whether the sequential composition is valid and, perhaps more importantly, why the composition may succeed or fail.

The same approach can be extended to sequential plan transfer by using the algorithm to compute the composition of two plans. In doing so, it evaluates whether the preconditions of the current plan align with the effects of the preceding plan, and vice versa. If the algorithm fails to construct a string diagram, it indicates incompatibilities. Sequential transfer thus allows for seamless integration of plans, highlighting the flexibility of string diagrams in supporting sequential plan transfer across different problems.

4.6.1 Advantages of the Categorical Structure for Sequential Transfer

Checking valid plan composition. A significant advantage of the categorical structure is its ability to validate plan composition. By using parallel ($\otimes$) and sequential (; or $\circ$) composition, various parts of a plan can be effortlessly combined. This feature supports sequential transfer by validating the compatibility of plans that may have been used to address distinct problems.

Error detection through failed composition. Along the same lines, the ability of the categorical structure to detect incompatibilities during plan composition is another advantage. If the preconditions and effects of adjacent actions are not compatible, the algorithm will fail to construct a valid string diagram. This mechanism offers automatic validation, reducing the need for manual checks and enabling more robust error detection in complex, multi-step plans.

Deformation invariance. A key benefit of this representation is its invariance under planar deformation (see Section 4.3.0.1) which means the diagram can be manipulated mathematically without changing the underlying semantics. This property enables users to explore alternative sequences of actions by sliding boxes along strings, analogous to algebraically rearranging terms in an equation. This allows for the exploration of alternative but valid plans by reordering actions in the diagram.

Capturing implicit conditions. The categorical structure ensures that all preconditions and effects, including implicit conditions, are captured explicitly as objects in the string diagram. This makes it easy to track how literals propagate through the plan, both upstream and downstream. Users can directly observe how actions influence the plan and how intermediate states satisfy preconditions for subsequent actions. This explicit tracking of conditions aids in identifying points of failure in the plan, making debugging and refinement more intuitive.

Automatic layout encoding. The linear notation associated with the string diagram representation directly corresponds to the layout of boxes and strings, making it particularly useful for conveying the order and temporal progression of a plan. This also provides a native encoding for potentially large string diagram visualizations.

Support for modularity. The categorical approach inherently supports modularity and scalability. By allowing string diagrams to be composed, plans can be broken down into manageable

subcomponents, which can be independently validated and later combined into larger, more complex plans. This modular nature is particularly advantageous in robotics, where complex tasks are often composed of smaller, reusable sub-tasks that can be dynamically integrated to meet evolving requirements, as we will explore in the next chapter.

4.6.2 Limitations of the Categorical Structure for Sequential Transfer

Limited support for PDDL extensions. The current encoding scheme is restricted in terms of the PDDL features it supports. Notably, constructs such as quantifiers, equalities, and other advanced extensions are not yet captured in this string diagram representation. These limitations restrict the applicability of the current approach to only a subset of PDDL's capabilities, which could hinder the analysis of more complex plans that make use of these features.

Inability to capture complementary relationships between literals. Another limitation is the inability to capture the inherently complementary relationships between the positive and negated versions of the same literal. Under the current implementation, positive and negated literals are treated as independent entities and can be components of monoidal objects simultaneously. This approach can lead to inconsistencies when negated literals are involved in a plan, as the relationships between such literals are not properly encoded or visualized.

Visually intractable for long plans. A significant limitation of the current system is the scalability of the visualization for long plans with many actions. As the number of actions grows, the resulting diagrams become increasingly unwieldy and difficult to interpret. This challenge can obscure insights that could be drawn from the visualization, making it less effective for analyzing complex plans. To address this, future work could focus on implementing algorithms that detect repeated patterns in actions and group them into subtasks, providing more efficient visualizations.

4.7 Conclusion

In this chapter, we provided a formal definition of sequential transfer, outlining how plans can be composed within the same domain while ensuring compatibility between preconditions and effects. We introduced string diagrams as a powerful tool for interpreting PDDL plans, enabling a clear visualization of both explicit and implicit dependencies between actions. Additionally, we discussed the novel algorithm for building string diagrams, which automates the composition of plans and ensures their validity. The advantages of this approach, including seamless plan composition, deformation invariance, and improved error detection, were highlighted, alongside key limitations such as visual scalability for complex plans. This also included how sequential transfer can support explainability by showing how the formal structure provided by string diagrams enhances transparency and user understanding of action dependencies and plan execution. In the next chapter, we will build on these ideas to discuss hierarchical transfer and its validation, extending our framework to handle modular plans and programs.

Chapter 5: Hierarchical Transfer using Functors

Hierarchical transfer involves reusing a task plan π by integrating it into a hierarchically structured framework of plans, where the transferred plan is inserted at a specific level of abstraction. Unlike sequential transfer, hierarchical transfer requires that the inserted plan not only integrates sequentially with other modules but also preserves the overall behavior dictated by the plan hierarchy. Building on the formal structures introduced in earlier chapters, we explore how category theory can be used to ensure these properties in hierarchical transfer.

Definition 19 (Hierarchical Transfer)**. Hierarchical Transfer** refers to the process of replacing an existing plan, denoted $\pi_{(i+1)k}$, within a hierarchy H with a new task plan π', while ensuring valid composition and preserving the overall behavior of the hierarchical structure. The first index, $(i+1)$, indicates the level of abstraction within the hierarchy, and the second index, k, refers to the plan's position within the sequence of plans at the same level.

The hierarchy H is defined as a collection of mappings from plans at the i-th level to sequences of plans at the $(i+1)$-th level:

$$H = \{\dots, \pi_{in} \mapsto \pi_{(i+1)(k-1)} \circ \pi_{(i+1)k} \circ \pi_{(i+1)(k+1)}, \dots\}$$

where the mappings satisfy the following conditions:

$$\text{Pre}(\pi_{in}) \mapsto \text{Pre}(\pi_{(i+1)[0]}), \quad \text{Eff}(\pi_{in}) \mapsto \text{Eff}(\pi_{(i+1)[\text{end}]})$$

The mappings ensure that the precondition of π_{in} matches the precondition of the first plan

in the sequence, $\pi_{(i+1)[0]}$, and the effect of π_{in} matches the effect of the last plan, $\pi_{(i+1)[\text{end}]}$.

When the plan $\pi_{(i+1)k}$ is replaced by π', a new hierarchy H' is produced. The following conditions must be satisfied:

(i) **Valid Composition**. The new plan π' must compose sequentially with the adjacent plans in the hierarchy at level $(i+1)$. Specifically, the following conditions must hold:

$$\text{is\_valid\_sequence}(\pi' \circ \pi_{(i+1)(k+1)}) = \text{True}$$

$$\text{is\_valid\_sequence}(\pi_{(i+1)(k-1)} \circ \pi') = \text{True}$$

This ensures that π' can be correctly integrated into the sequence of plans at the $(i+1)$-th level without introducing conflicts in execution.

(ii) **Preservation of Behavior**. The new hierarchy H' must preserve the behavior specified by the highest-level plans, specifically π_{in}. For each high-level plan π_{in} that invokes plans at the $(i+1)$-th level, the overall composition of the plans should remain valid:

$$\text{is\_valid\_hierarchy}(\pi_{in} \mapsto \pi_{(i+1)(k-1)} \circ \pi' \circ \pi_{(i+1)(k+1)}) = \text{True}$$

This validation condition ensures that replacing $\pi_{(i+1)k}$ with π' maintains the alignment with the preconditions and effects of the higher-level plan π_{in}.

Hierarchical transfer is particularly valuable in robotic systems, where tasks are often decomposed into subtasks across various levels of abstraction. For instance, in a machine tending scenario can be decomposed into subtasks like opening a door, picking up an object, placing it in the machine, and closing the door. Hierarchical transfer enables the replacement of specific plans while ensuring that the overall system behavior remains consistent throughout the hierarchy.

5.1 Overview

Given the importance of hierarchical transfer in ensuring consistent system behavior across task levels, a structured approach to defining and managing these hierarchical relationships be-

comes essential. It should come as no surprise to practitioners and engineers that initial design discussions for robot programs typically begin with high-level, verbal descriptions of task objectives, followed by a breakdown of actions that depend hierarchically on previous steps. In this chapter, we argue that these task specifications and decompositions can be rigorously framed using category theory, which allows us to formalize hierarchical relationships between different levels of abstraction. This formalism is particularly beneficial for hierarchical transfer, as it provides a set of rigorous rules for validating plan transfers.

Our approach leverages the mathematical tools of category theory, specifically *monoidal categories* and *functors*, to formalize these rules. Monoidal categories enable us to capture the compositional relationships between task modules in a sequence, while functors and monoidal functors provide structured mappings between different levels of abstraction—from high-level tasks to low-level ones.

In the remainder of this chapter, we present a detailed exposition of this hierarchical framework. Section 5.2 introduces our approach within the context of a framework designed to translate high-level task specifications to low-level motion commands, called the RoboCat framework. Section 5.4 reviews the concept of monoidal categories and functors. In Section 5.5, we discuss the formalization provided by RoboCat and its application to machine-tending problems, which we demonstrate with an example in Section 5.3. Finally, Sections 5.7 and 5.8 will conclude the chapter and explore directions for future research.

5.2 The RoboCat Framework

The RoboCat framework defines a categorical architecture consisting of three distinct categories: the high-level symbolic task specification, denoted as Task; the virtual, or software-based, specification, called $\mathsf{Virtual}$; and the low-level motion command specification, Motion. The goal

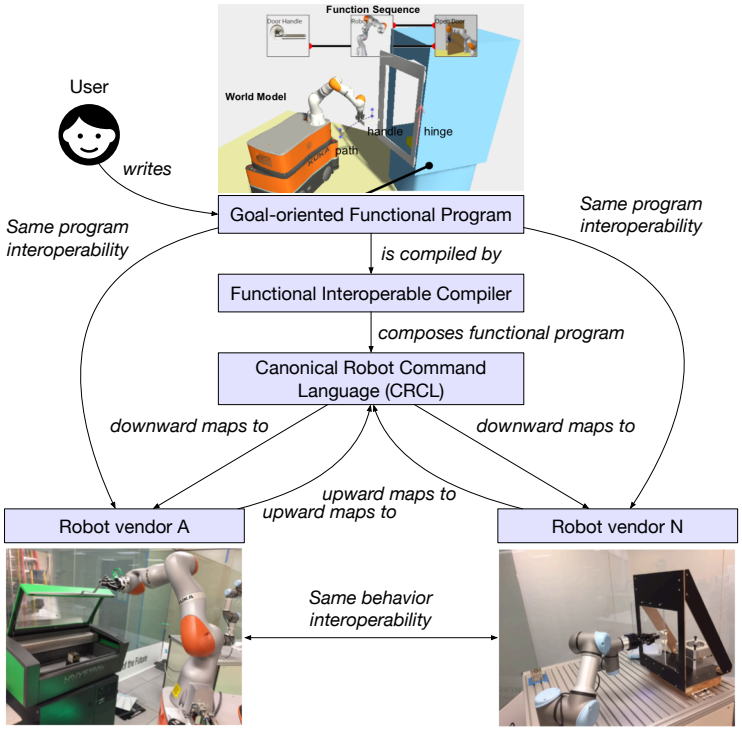


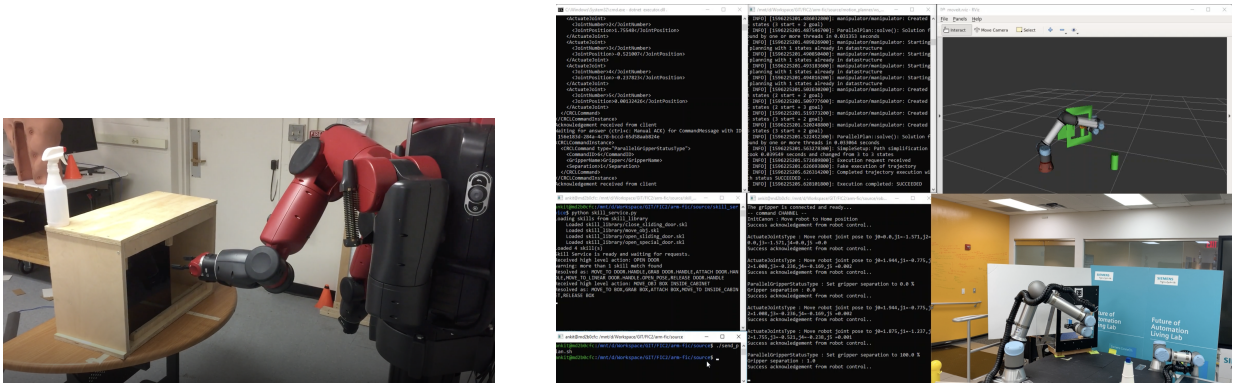
Figure 5.1: The RoboCat Framework.

of RoboCat is to guide robot programmers in implementing a sequence of high-level actions that form a structured, multi-layered task plan. This plan automatically translates the high-level task specification into corresponding low-level motion commands. The low-level motion commands are expressed using the Canonical Robot Command Language (CRCL) [123], [124], which is compatible with the Robot Operating System (ROS) [125], making it a widely interoperable option across modern robotic platforms. Programs developed in accordance with this framework leverage the formal properties of categories and functors, ensuring that hierarchical transfer is reliable, predictable, free of side effects, and capable of preserving intended behavior at each level of the task hierarchy.

Ultimately, this framework offers three key benefits: (i) a goal-oriented, functional approach to robot programming that ensures hierarchical consistency, (ii) automatic compilation of high-level, goal-oriented task plans into low-level motion specifications, and (iii) interoperability across

different robotic platforms, which preserves the integrity of the high-level task plans.

5.3 Case Study: Machine-Tending Problem



(a) ReThink Robotics Baxter robot. (b) Universal UR5 robot.

Figure 5.2: Two robots opening a slide door using the same high-level program.

To illustrate this framework, we walk through an example where two goal-oriented programs were compiled to instruct a Rethink Robotics Baxter robot and a Universal Robot UR5 to open a sliding door in two distinct operating environments, simulating a machine-tending scenario (see Figure 5.2). Machine-tending involves a robot loading and unloading workpieces for a machine to process. Examples of such machines include CNC machines, paint machines, and lithography machines. These machines often have a compartment protected by a hinge or sliding door where the unprocessed workpiece must be placed. This requires the robot to be programmed to open a compartment door, grip, and transfer workpieces.

This case study is particularly useful for examining the benefits of hierarchical transfer, as each of these actions—such as opening a door, gripping, and transferring—can be executed in a hierarchical manner. The hierarchical structure allows us to swap out different components of the machine-tending task, like the gripper type or the door mechanism, while preserving the overall system behavior. By isolating each subtask in this way, we can easily adapt to different setups or constraints without requiring a complete reprogramming of the task flow, demonstrating the

flexibility and robustness of hierarchical transfer.

5.4 Modules using String Diagrams

To formally capture the modular structure of robotic tasks, we represent modules as morphisms within string diagrams. As we have seen in Chapter 4, string diagrams offer a powerful diagrammatic language that naturally extends beyond task plan composition and is particularly effective for modeling functional modules of robot programs. In this categorical perspective, each box in a string diagram corresponds to a module, with incoming strings representing input arguments or preconditions, and outgoing strings representing output arguments or effects.

String diagrams offer several advantages for capturing hierarchical structures. They provide a composable representation of robot program modules, allowing complex behaviors to be broken down into smaller, reusable components. This abstraction hides the internal logic of individual modules while ensuring that the sequence of modules remains valid. Consequently, programmers can focus on the overall structure and flow of the task plan without needing to delve into specific implementation details. As a result, string diagrams become a powerful tool for formalizing abstract and transferable modules.

5.4.1 Functors Between String Diagrams

Effectively relating modules as string diagrams across different layers of a hierarchy—while preserving their structure and behavior—requires a formal mechanism, which we have chosen to represented as functors between string diagrams.

Functors provide the appropriate mathematical foundation for connecting modules of plans and programs across different abstraction levels because they ensure that both the structural relationships and the compositional integrity of the modules are preserved across hierarchical

layers. As discussed in Definition 3 in Section 3.1, functors map objects and morphisms from one category to another, preserving the compositional structure of the original category within the target category. When both the source and target categories are monoidal categories, a monoidal functor requires that not only are the objects and morphisms be mapped correctly, but that the monoidal products and unit objects of the source category are preserved in the target category. Additionally, when the categories involved are symmetric monoidal categories (SMCs), the functor must also respect the symmetry of the monoidal product. A formal definition of a symmetric monoidal functor can be found in Definition 15.

When a task plan from one level of abstraction is mapped to another—such as moving from a high-level plan to a more detailed implementation—monoidal functors ensure that the composition of modules remains coherent, thereby preserving the overall behavior of the plan.

If one of the categories involved is non-monoidal, the functor acts as a regular functor, mapping objects and morphisms between categories without needing to preserve the monoidal structure. This flexibility is useful when transferring between categories that do not require the preservation of monoidal products, such as when dealing with purely functional operations that do not involve resource management or parallel execution. By leveraging functors in this way, the RoboCat framework ensures that task plans can be transferred across different abstraction levels and scenarios while maintaining their compositional and hierarchical integrity.

5.5 Categorical Framework

In this framework, we model the hierarchical abstraction levels of a robotic task plan and program using two SMCs and one non-monoidal category. Each category corresponds to a different aspect of the robot's task planning and execution hierarchy, representing increasingly concrete levels of abstraction, from high-level task planning to low-level motion execution. These cate-

gories are Task, Virtual, and Motion. The SMC Task represents symbolic work cell operations and goal-oriented task planning. This level focuses on abstract task specifications and the symbolic manipulation of objects within the work cell. The SMC Virtual represents the virtual work cell, providing a detailed simulation of the physical environment, including the robot's components and the physical properties of objects. The category Motion, without a monoidal structure, represents the low-level motion primitives of the robot. This level handles the specific configuration spaces and low-level actuating commands required for physical movements.

In the following sections, we define each of these categories in greater detail, specifying their objects, morphisms, composition, and monoidal products (where applicable). We then introduce two functors: $F : \mathsf{Task} \to \mathsf{Virtual}$ and $G : \mathsf{Virtual} \to \mathsf{Motion}$, which map between these categories.

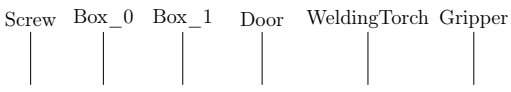
5.5.1 The Category of the Symbolic Work Cell, Task

The category for the symbolic work cell and high-level task, called Task, is an SMC defined in terms of the following objects, morphisms, monoidal product, and composition.

Objects. Objects consist of symbols representing real-world items physically available during program execution. At least one symbol must refer to the robot's gripper. Some examples include:

- Relevant physical resources within the work cell such as `Door`, `Box`, `Screw`,
- Robotic accessories and devices, including end-effectors such as `Gripper`, `WeldingTorch`, `ForceTorqueSensor`, `CollisionSensor`, etc.

These objects can be visually represented as strings, as shown in Eq. 5.1.



(5.1)

Multiple strings of the same type are indexed to identify unique instances of that type which

is captured by the "Box_0" and "Box_1" strings in Eq. 5.1.

Morphisms. Morphisms represent a finite set of task actions, a_i, describing robot behaviors from the perspective of the robot manipulator. Examples include `OpenDoorUsingGripper` and `PlaceBoxInCabinet`. Task actions are further characterized by their interactions with resources: if an object is passed as input and returned as output, the resource is required but not consumed; if an object is passed as input but not returned as output, the resource is considered consumed during the action.

Monoidal product, $\otimes$. The monoidal product represents the logical conjunction of objects and morphisms, allowing the combination of resources and actions within the same task.

Composition, $\circ$. Composition is defined such that the composite morphism $a_1; a_2$ implies that the output symbols of a_1 match the input symbols of a_2. Semantically, this means that the resources freed up after a_1 are required as inputs to execute a_2.

5.5.2 The Category of the Virtual Work Cell, Virtual

The category for the virtual work cell, called Virtual, is an SMC defined in terms of the following objects, morphisms, monoidal product, and composition.

Objects. Objects consist of elements from a world model file expressed using standard data formats, such as the Simulation Description Format (SDF) [126]. These elements represent physical objects, robot components, and their geometric and physics-based attributes. At least one object must refer to the robot gripper. Each element in the world model is represented as an XML-based hierarchical data structure, making the data for objects in Virtual richer in structure than those in Task. For example:

- `Gripper`: A model element with attributes such as the pose for each link and the gripper status.

```
<model name="Gripper">
  <!-- Initial position (x, y, z) and orientation (roll, pitch, yaw) -->
  <pose>0 0 0 0 0 0</pose>
  <link name="gripper_link">
    <!-- Relative pose of the link -->
    <pose>0 0 0 0 0 0</pose>
  </link>
  <link name="gripper">
    <!-- true for closed/gripped, false for open/released -->
    <status>true</status>
  </link>
</model>
```

- `Box_0`: A model element with attributes such as pose, dimensions, and material properties.

```
<model name="Box_0">
  <!-- Position (x, y, z) and orientation (roll, pitch, yaw) -->
  <pose>1 1 0.5 0 0 0</pose>
  <link name="box_link">
    <!-- Relative pose of the link -->
    <pose>0 0 0 0 0 0</pose>
    <collision name="box_collision">
      <geometry>
        <box>
          <!-- Box dimensions -->
          <size>0.5 0.5 0.5</size>
        </box>
      </geometry>
    </collision>
    <visual name="box_visual">
      <geometry>
        <box>
          <!-- Box dimensions -->
          <size>0.5 0.5 0.5</size>
        </box>
      </geometry>
      <material>
        <script>
          <!-- Material property of the box -->
          <uri>file://media/materials/scripts/gazebo.material</uri>
          <name>Gazebo/Red</name>
        </script>
      </material>
    </visual>
  </link>
</model>
```

Morphisms. Morphisms are software routines, s_i, that manipulate the contents of the data objects without introducing any side effects. For example, a morphism that instructs the gripper to move to its home position would update the `pose` attribute of the gripper element in the model file.

This requires each world element to have predefined attributes, such as the gripper's home pose, the door's open pose, and other key geometric markers. In SDF, these resources and markers are provided using custom tags. For instance, the `<home_pose>` can be specified as:

```
<home_pose> <pose>0 0 0 0 0 0</pose> </home_pose>
```

By adding a standardized set of markers for each robot manipulator, the software program remains agnostic to the specific manipulator being used. For example:

- `MoveToHome`: Updates the gripper's pose attribute to the stored home position.
- `PickUpBox`: Changes the gripper's pose and status attributes.

Monoidal product, $\otimes$. The monoidal product indicates that data objects exist in the same local scope, which means that they can be accessed and manipulated together with in the same software routine. This allows for the combination of multiple objects, such as a gripper and a box, within the same logical scope.

Composition, $\circ$. Composition implies sequential invocation of the software routines. The composite morphism $s_1; s_2$ implies that the output of s_1 feeds directly into the input of s_2. Semantically, this means that the results of s_1 are immediately passed to s_2.

5.5.3 The Category of the Motion Primitives, Motion

The category for motion primitives, called Motion, is a category without monoidal structure, defined in terms of the following objects, morphisms, and composition. This lack of monoidal structure reflects the sequential nature of the commands at this level of abstraction, where operations are performed one after the other rather than in parallel or as combinations of objects.

Objects. Objects consist of the set of robot configuration spaces for a single robot platform. The configuration space of a robot consists of all possible positions and orientations of its joints and

end-effectors. For instance, a 6-DOF robotic arm might have a configuration space defined by six joint angles.

Morphisms. Morphisms consist of the set of low-level, discrete commands, c_i, in the form of CRCL that update the robot's configuration space. Sample CRCL code for morphisms that represent movement and state change can be found below:

```
<CRCLCommand xsi:type="MoveToType">
  <MoveStraight>true</MoveStraight>
    <EndPosition>
      <Point>
        <X>8.25</X> <Y>1</Y> <Z>0.5</Z>
      </Point>
      <XAxis>
        <I>1</I> <J>0</J> <K>0</K>
      </XAxis>
      <ZAxis>
        <I>0</I> <J>0</J> <K>-1</K>
      </ZAxis>
    </EndPosition>
</CRCLCommand>

<CRCLCommand xsi:type="SetEndEffectorParametersType">
  <ParameterSetting>
    <ParameterName>mode</ParameterName>
    <ParameterValue>grip</ParameterValue>
  </ParameterSetting>
</CRCLCommand>
<CRCLCommand xsi:type="SetEndEffectorType">
  <Setting>1.0</Setting>
</CRCLCommand>
```

Each morphism updates the joint configuration space of the robot, producing a new configuration space. In Motion, each morphism has one input and one output, reflecting the sequential nature of low-level commands.

Composition, $\circ$. Composition implies sequential commands sent to the robot platform. The composite morphism $c_1; c_2$ implies that the output configuration space of c_1 is equal to the input configuration space of c_2. Semantically, this means that c_2 updates the configuration space produced by c_1.

To relate Task, Virtual, and Motion hierarchically, two functors must be defined, namely F : Task $\to$ Virtual and G : Virtual $\to$ Motion. These functors ensure that task plans are consistently mapped, from high-level symbolic tasks to virtual representations, and finally to low-level motion execution. The procedural steps for defining these functors, including the mappings for objects, morphisms, and compositions, are presented in the following sections.

5.5.4 F : Task $\to$ Virtual

The monoidal functor F is a functor between two SMCs, which means that it must preserve composition, monoidal product, and the symmetry of the braiding isomorphism in Task.

Object map. For each object in Task, create a corresponding element in the SDF world model file that includes the object's properties and object markers. For example:

- `Box_0` in Task maps to an SDF model element `<model name="Box_0">` in Virtual with attributes such as geometry, mass, and physical properties.
- `Gripper` in Task maps to an SDF model element `<model name="Gripper">` in Virtual with appropriate markers for predefined attachment points.

Morphism map. For each morphism in Task, implement an empty subroutine such that it accepts the corresponding input argument types and returns the corresponding argument types. For example, the morphism `OpenDoorUsingGripper` : `Gripper` $\otimes$ `Door` $\to$ `Gripper` $\otimes$ `Door` in Task maps to a subroutine in Virtual:

```
def OpenDoorUsingGripper(gripper: GripperModel, door: DoorModel)
                        -> (GripperModel, DoorModel):
    # Implementation to manipulate SDF elements
    return gripper, door
```

In some cases, a morphism in Task might map to a composite morphism in Virtual, where multiple subroutines are invoked sequentially to achieve the high-level task. This composite structure

reflects the breakdown of the higher-level task into more detailed operations in the virtual environment. For example, the morphism `OpenDoorUsingGripper` : `Gripper` $\otimes$ `Door` $\to$ `Gripper` $\otimes$ `Door` in Task could map to a composite morphism in Virtual that involves multiple steps such as moving to the door, grasping the door handle, and rotating the handle. This can be implemented as follows:

```
def OpenDoorUsingGripper(gripper: GripperModel, door: DoorModel)
                        -> (GripperModel, DoorModel):
    # Step 1: Move the gripper to the door handle
    gripper, door = MoveToDoor(gripper, door)
    # Step 2: Grasp the door handle
    gripper, door = GraspHandle(gripper, door)
    # Step 3: Rotate the handle to open the door
    gripper, door = RotateHandle(gripper, door)

    return gripper, door
```

In this example, the high-level task action of opening the door is broken down into smaller, sequential actions within the virtual environment. Each of these subroutines—`MoveToDoor`, `GraspHandle`, and `RotateHandle`—represents a step that collectively fulfills the task specified by the original morphism in Task.

Preserving composition. Composition is preserved if the program invokes each subroutine in the same order as the composition within Task. For example, given morphisms $a_1 : X \to Y$ and $a_2 : Y \to Z$ in Task, the composition $a_2 \circ a_1$ maps to:

```
def composed_function(x: X) -> Z:
  y = a1(x)
  z = a2(y)
return z
```

Preserving monoidal products. Monoidal products are preserved if the input argument types and output argument types are aligned with the expected data objects prescribed by the object map. For example:

- The monoidal product (`Gripper` $\otimes$ `Box_0`) in Task maps to the corresponding elements

(GripperModel $\otimes$ BoxModel) in Virtual.

Preserving the symmetry of the braiding isomorphism. The symmetry of the braiding isomorphism is preserved if swapping the order of a pair of arguments to a subroutine twice yields the same result as when the arguments are provided in their original order. For example:

```
def swap_and_swap_back(arg1: Any, arg2: Any) -> (Any, Any):
  swapped = (arg2, arg1)
  return swapped[1], swapped[0]

assert f(swap_and_swap_back(arg1, arg2)) == f(arg1, arg2)
# Test Passed
```

5.5.5 G : Virtual $\to$ Motion

The functor G is responsible for translating objects and morphisms from the virtual work cell to the robot's motion primitives, ensuring that the composition of actions in Virtual is preserved in Motion.

Object map. Each object in Virtual is mapped to a corresponding configuration space in Motion. This translation involves converting the geometric and physical attributes defined in the SDF world model into specific configuration states for the robot. For objects resulting from a monoidal product in Virtual, their interaction is translated into a configuration state. For example, the position of the gripper in Virtual might be mapped to the joint angles required to achieve that position in Motion.

Morphism map. Each morphism in Virtual is mapped to a corresponding low-level actuating command in Motion. These commands are specified using CRCL and are responsible for manipulating the robot's configuration space based on the software routines in Virtual. For instance:

- The morphism `MoveToHome` in Virtual maps to a CRCL command that moves the robot's

gripper to the home position:

```
<CRCLCommand xsi:type="MoveToType">
  <MoveStraight>true</MoveStraight>
  <EndPosition>
    <Point>
      <X>0</X> <Y>0</Y> <Z>0</Z>
    </Point>
    <XAxis>
      <I>1</I> <J>0</J> <K>0</K>
    </XAxis>
    <ZAxis>
      <I>0</I> <J>0</J> <K>-1</K>
    </ZAxis>
  </EndPosition>
</CRCLCommand>
```

- The morphism `PickUpBox` in Virtual maps to a composite morphism consisting of a sequence of CRCL commands that move the gripper to the box, grasp it, and move it to the desired location:

```
<CRCLCommand xsi:type="MoveToType">
  <MoveStraight>true</MoveStraight>
  <EndPosition>
    <Point>
      <X>1.5</X> <Y>1.0</Y> <Z>0.5</Z>
    </Point>
    <XAxis>
      <I>1</I> <J>0</J> <K>0</K>
    </XAxis>
    <ZAxis>
      <I>0</I> <J>0</J> <K>-1</K>
    </ZAxis>
  </EndPosition>
</CRCLCommand>

<CRCLCommand xsi:type="SetEndEffectorType">
  <Setting>0.5</Setting>
</CRCLCommand>
```

In practice, this is implemented by generating subroutines in Virtual that issue the corresponding CRCL commands to the robot platform.

Preserving composition. The composition of morphisms in Virtual is preserved in Motion by ensuring that the sequence of CRCL commands reflects the same order as the software routines in Virtual. For example, given the morphisms `PickUpBox` : `BoxModel` $\otimes$ `GripperModel` $\rightarrow$

`BoxModel` $\otimes$ `GripperModel` and `MoveToHome` : `GripperModel` $\to$ `GripperModel` in Virtual, their composition `PickUpBox`; `MoveToHome` maps to the concatenated sequence of the previous CRCL command blocks. This ensures that the ordered execution of tasks in the virtual space corresponds directly to the order of physical actions executed by the robot.

5.5.5.1 Considering Alternative Categories

An alternative approach could have been to model the objects and morphisms across all abstraction levels within one symmetric monoidal category, with equations like `OpenDoorUsingGripper` represented as a sequence of operations such as `MoveToDoor`; `AttachToDoor`; ...`MoveHome`. However, this would mathematically imply combining objects from the real-world, software abstraction, and motion primitive levels within the same context. In practice, this would lead to ambiguities regarding how the monoidal product behaves across such diverse representations. For instance, it would be difficult to interpret what it means to combine an object representing the symbolic door with an object representing to a particular robot configuration space. To avoid this inconsistency, we instead define distinct categories for each abstraction level, allowing us to rigorously define how $\circ$ and $\otimes$ operate within their respective contexts.

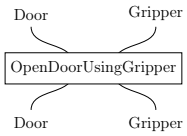
5.6 Application of the RoboCat Framework to the Machine-Tending Problem

Building on the categorical framework detailed in the previous section, we now apply these principles to a practical machine-tending task using the RoboCat framework. In this section, we present the open door instruction using string diagrams to illustrate the hierarchical structure and modular approach enabled by the framework. While the example focuses on the open door step, the full implementation also includes moving the object and closing the door.

The example was developed using a C++ program with methods corresponding to each step

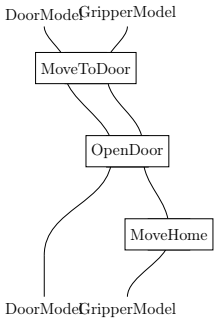
of the task: `MoveToDoor`, `AttachToDoor`, `OpenDoor`, `MoveToOpen`, `ReleaseDoor`, and `MoveHome`. Each method constructs the appropriate CRCL commands based on the object models provided and sends these commands to the robot controller. Additionally, an SDF model file was created to describe the geometries of both the Baxter robot and the sliding door. Specific markers were added to the SDF file to define key positions, including the gripper's natural pose, the attachment and release modes of the gripper, the gripper's home position, as well as the door handle's open and closed positions.

The category-theoretic model includes the categories Task, Virtual, and Motion as defined in Section 5.5, along with two functors: $F : \mathsf{Task} \to \mathsf{Virtual}$ and $G : \mathsf{Virtual} \to \mathsf{Motion}$. The symmetric monoidal category Task represents symbolic abstractions of the physical world, with objects like {`Door`, `Gripper`, ...}, and morphisms such as {`OpenDoorUsingGripper`, ...}, representing possible actions that use or consume resources. An example morphism in Task is shown in Equation 5.2.



(5.2)

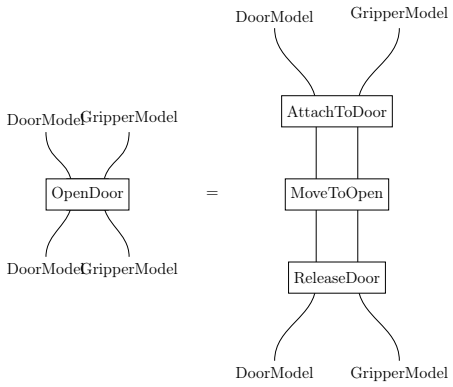
Morphisms in Virtual represent the skills needed to complete tasks, such as {`MoveToDoor`, `OpenDoor`, `MoveHome`, `AttachToDoor`, `MoveToOpen`, `ReleaseDoor`, ...}. An example morphism in Virtual is shown in Equation 5.3.



(5.3)

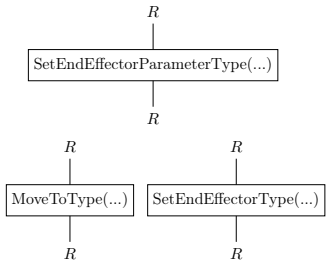
In Virtual, we specify semantic equivalences between functions using equations, and composi-

tions of functions are defined. For instance, we define `OpenDoor` as a composition of `AttachToDoor`, `MoveToOpen`, and `ReleaseDoor`, as shown in the following equation:



(5.4)

In practice, this equivalence is implemented as sequential invocation of `AttachDoor`, `MoveToOpen`, `ReleaseDoor` within the `OpenDoor` subroutine. Finally, the morphisms in Motion are fully parameterized CRCL commands [124], such as `MoveToType`, `SetEndEffector`, `SetEndEffectorParameterType`. An example of these morphisms depicted in diagrammatic form is shown in Equation 5.5.



(5.5)

Figure 5.3 provides an expanded view of this model, illustrating a full string diagram for the robot program described in Section 5.3. Notice that we have yet to discuss how these modules will be executed procedurally; the design focuses first on defining the functions and their interfaces, effectively isolating different parts of the code responsible for handling various aspects of the robot's state.

In this implementation, the functors between categories are achieved through offline programming with full attention to preserving functoriality at each abstraction level. Ideally, these

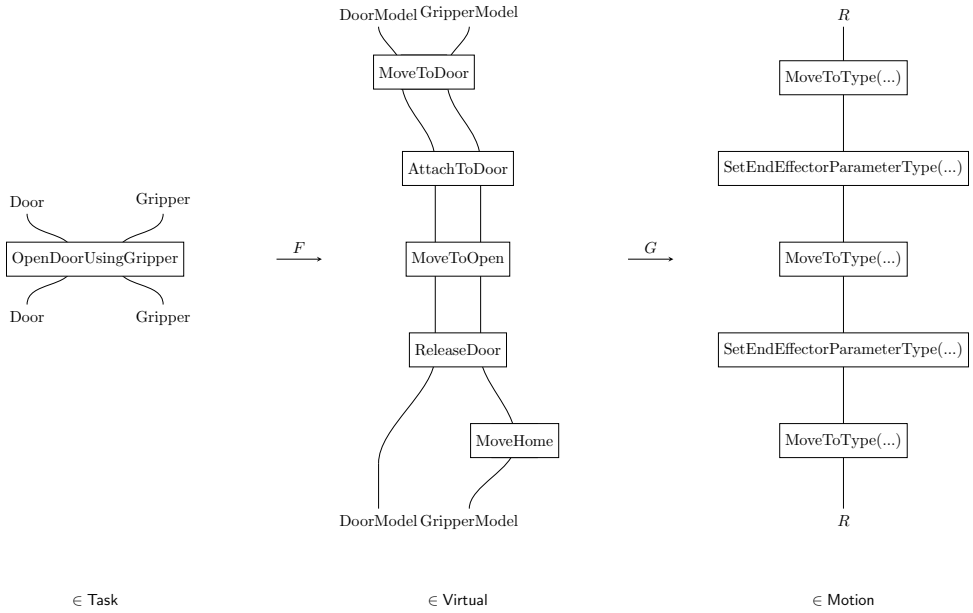


Figure 5.3: The full string diagram for the example robot program.

mappings would be handled automatically within a category-theoretic software framework, guaranteeing functoriality by design. However, in this case, the implementation is done manually, and formal verification is left for future work.

The functor F represents the human-performed task of developing a digital model of the physical environment and mapping abstract commands in the physical world to corresponding virtual skills. Since this process is relatively straightforward, we will not elaborate further on it.

On the other hand, the functor G is responsible for mapping the virtual world components to specific robot platforms. Table 5.1 lists the software functions that translate SDF model information (from the robot and door) into fully parameterized CRCL commands, thereby implementing G. Once the body of the function is implemented, it can be invoked without regard for how to construct the CRCL command.

Virtual Morphism	Motion Morphism	Functor G
MoveToDoor	MoveTo(...)	`MoveToDoor(GripperModel, DoorModel):` `pose = GetGripperNaturalPose(GripperModel)` `position = GetDoorHandlePosition(DoorModel)` `send(MoveToCRCL(pose, position))` `return GripperModel, DoorModel`
MoveToOpen	MoveTo(...)	`MoveToOpen(GripperModel, DoorModel):` `pose = GetGripperCurrentPose(GripperModel)` `position = GetDoorOpenPosition(DoorModel)` `send(MoveToCRCL(pose, position))` `return GripperModel, DoorModel`
MoveHome	MoveTo(...)	`MoveHome(GripperModel, DoorModel):` `pose = GetGripperNaturalPose(GripperModel)` `position = GetDoorHomePosition(DoorModel)` `send(MoveToCRCL(pose, position))` `return GripperModel, DoorModel`
AttachToDoor	SetEndEffector(...) $\circ$ SetEndEffectorParameter(...)	`AttachToDoor(GripperModel):` `endEffectorMode = GetGripperAttachMode(GripperModel)` `send(SetEndEffectorParameterCRCL(endEffectorMode))` `endEffectorSetting = GetGripperAttachMode(GripperModel)` `send(SetEndEffectorCRCL(endEffectorSetting))` `return GripperModel`

Table 5.1: Pseudocode for functor G : Virtual $\rightarrow$ Motion.

5.7 Discussion

While brute-force programming can achieve similar results, this framework provides a systematic approach to decomposing and organizing robot manipulator programs, ensuring a predictable hierarchical structure. In this chapter, we employ an algebra of functions to create mathematically sound programs for manipulators, enforcing semantic relationships through the use of functors, equations, and composition. This approach results in a hierarchical, human-interpretable compiler where users can follow the strings to compile the code, ensuring consistency across abstraction layers as task plans are transferred between hierarchies. In this section, we summarize the advantages and limitations of the propose categorical framework.

5.7.1 Advantages of the Categorical Structure for Hierarchical Transfer

Validation of Hierarchical Structure. The categorical framework provides a formal structure that supports validating the hierarchical structure of task plans. Through functors F and G, the symbolic objects in Task can be reinterpreted within the contexts of Virtual and Motion, ensuring that compositional integrity is preserved as task plans move from symbolic representations to executable commands. This validation guarantees that newly transferred plans can be adequately integrated with the interfaces of existing task plans at the same or higher-level.

Graphical Resource Tracking. The framework utilizes string diagrams as a powerful graphical tool for both transparent program compilation and resource tracking. In industrial settings, where monitoring resource consumption for robot manipulators is critical, string diagrams enable precise tracking of resources at different time resolutions. By examining slices of the diagram, it is possible to determine the resources consumed and produced at specific points in time, providing a temporal and resource-sensitive representation that enhances both safety and efficiency. For instance, Table

5.2 is temporal view of Equation 5.6 broken down into $n = 3$ time slices. This encoding tracks the entire process for `OpenDoorUsingGripper`, detailing the objects consumed and produced:

$$(id_{Door} \otimes id_{Gripper}); p; (id_{Door} \otimes id_{Gripper}) \tag{5.6}$$

t	symbolic	mathematical
1	`Door` $\otimes$ `Gripper`	$id_{Door} \otimes id_{Gripper}$
2	`OpenDoorUsingGripper`	p
3	`Door` $\otimes$ `Gripper`	$id_{Door} \otimes id_{Gripper}$

Table 5.2: A temporal view of Equation 5.6.

Explainable Hierarchical Transfer. String diagrams also support explainable hierarchical transfer by providing a clear visualization of task plans across multiple abstraction layers. This graphical approach allows users to intuitively track the flow and transformation of task plans which assures human validators that newly transferred modular components are properly integrated within the hierarchical structure. The combination of visual clarity and mathematical precision provides multiple forms of validation for integrating new plans in the hierarchy.

5.7.2 Limitations of the Categorical Structure for Hierarchical Transfer

Manual Implementation and Complexity. One limitation of the current framework is that the functors between categories, particularly those mapping virtual world components to robot platforms, are implemented manually. Ideally, these mappings would be automated within a category-theoretic software framework guaranteeing functoriality by definition. Currently, the manual process introduces complexity, requires expertise, and increases the potential for human error, reducing the scalability and accessibility of the framework.

Interoperability Across Diverse Robotic Platforms. Although the framework supports interoperability across different robotic platforms, this capability is currently constrained by the specific platforms and standards in use (e.g., CRCL and ROS). Extending the framework to handle a wider variety of robotic systems with different software architectures may require additional effort and refinement. Compatibility with non-standard platforms or emerging robotic ecosystems could prove challenging.

Resource Management in Dynamic Environments. The categorical framework's use of string diagrams offers a powerful tool for resource tracking, but its effectiveness relies on predefined resources and static environments. In highly dynamic environments where resources may change unpredictably (e.g., due to unexpected robot failures or environmental changes), ensuring that resource consumption and task execution remain valid can become difficult. Handling such variability while preserving the formal guarantees of the framework is a potential limitation.

5.7.3 Exploring Future Applications of String Diagrams

This chapter opens the door to further research exploring the advantages of using string diagrams as a representation for robot programs, particularly in the context of hierarchical transfer and validation. For example, string diagrams could be used to develop metrics that characterize the level of interoperability [127–129] or the similarity between robot program definitions by matching patterns on the hierarchical structures expressed through these diagrams. Additionally, the precise expression of temporal resource usage in string diagram formalism may serve as a valuable validation tool for detecting failures during execution [130]. The framework's use of functors to facilitate hierarchical transfer also presents new possibilities for Industry 4.0 initiatives [66,131], where the flexible and scalable transfer of task plans across different robotic platforms is essential.

Finally, the formal properties of string diagrams—such as their ability to represent sliding

and concatenating processes—can be leveraged to optimize hierarchical process plans for both single and multi-agent work cells [132]. Future work includes developing a software library that validates these structures in robot programs, enabling users to define hierarchical task plans while automatically benefiting from the formal guarantees provided by category-theoretic constructions.

5.8 Conclusion

This chapter introduced a definition of hierarchical transfer and presented a principled framework for defining hierarchical, goal-oriented programs and their programming interfaces, offering algebraic guarantees through category theory. By leveraging these mathematical structures, robot command sequences can be compiled from high-level descriptions in a modular and reliable manner, without requiring wholesale code replacement. The functorial requirements embedded within the categorical architecture ensure that programming interfaces support seamless composition and reuse of existing components. Additionally, the framework provides tools for validation which enable consistent verification of hierarchical structure and behavior as task plans are transferred and adapted. This was demonstrated in a machine-tending example, where hierarchical validation was achieved by construction.

Chapter 6: Structured Knowledge in Planning

Both sequential and hierarchical transfers share the fundamental quality of integrating plans within a broader framework of plans in the same planning domain. Analogical transfer, by contrast, aims to transfer entire plans from the original planning domain to another. For analogical transfer to work, it is essential to isolate the aspects of each domain where analogies can be established—specifically, the structured knowledge within them. Planning with such knowledge introduces the challenge of achieving a balance between abstraction and representation that is both semantically meaningful and computationally efficient. A well-designed planning domain abstraction simplifies world states and action models without compromising the essential semantics that accurately model the real world. The expressivity and structure of domain abstractions, however, depend on their representation (see Section 7.1 in [133]). A representation language should not only allow one express simultaneous facts about the world but also allow for the derivation implicit facts from explicitly stated ones [122]. It should efficiently update these facts about the world while preserving underlying semantics. In this chapter, we introduce a planning representation language that aims to achieve this balance and support analogical transfer.

6.1 Motivation

An inherent advantage of incorporating structured knowledge in planning, often termed *knowledge-based planning* [9], is the availability of facts that enable a comprehensive model of the

real world. Specifically, using domain-specific knowledge during planning supports more effective deduction about the impact of actions on real-world states. Knowledge-based planning employs structured representations, such as ontologies [75,134,135] and semantic networks [136–139], to accurately capture world states. However, faithfully integrating these knowledge-rich representations within a PDDL world state is challenging and, when achieved [137,139,140], often computationally expensive to propagate through the planning process.

One explanation for this computational inefficiency may be that the formal representations underlying PDDL—namely, propositional and first-order logic—do not natively support type hierarchies, transitive relationships, or densely connected predicates. Without mechanisms to reason about these structures, inferences based on this knowledge can become computationally intractable. Formalizing these structures is also essential for our purposes, as they provide the scaffolding necessary to establish mappings between different planning domains as part of analogical plan transfer.

To address both challenges, we propose a planning language that incorporates structured knowledge and supports analogical plan transfer within a single framework, eliminating the need for translation into a classical planning language like PDDL. Our approach is grounded in category-theoretic concepts, specifically **C-sets** [41] and **double-pushout (DPO) rewriting** [42]. This representation formalizes predicate semantics through functors that map a user-provided ontology to the data in the world state. By functorially adhering to the ontology, our language preserves the semantic integrity of the world state as it evolves.

In this chapter, we delve into related works about planning using semantic networks beginning in Section 6.2. We then provide a background on the model of planning as a state transition system in Section 6.3. In Sections 6.3.2 and 6.4, we thoroughly describe the features of both classical and categorical representations based on state transition systems. We then compare how

these two representations handle structured knowledge, satisfiability of preconditions, and implicit effects through an example in Section 6.5. The chapter concludes with Section 6.7 summarizing the advantages and limitations of the categorical structure.

6.2 Related Work

To place our work in context, we review existing literature on knowledge-based planning, with particular attention to the role of ontologies in structuring and enforcing connected world knowledge within planning systems.

In 2001, Wilkins & DesJardins published a seminal paper calling for the expanded use of knowledge in planning [9]. Knowledge during planning can come from a variety of sources as shown in Figure 6.1. Wilkins & DesJardins proposed that knowledge could enhance planning through the use of rich domain models, knowledge about good plans, and knowledge about the planning process to support plan repair. Based on this recommendation, our work focuses on integrating richer domain models to enable more descriptive and semantically rich representations of world states during planning.

Similar work has been done to enhance the representation of world states through various forms of structured knowledge. Approaches have included the use of typing and derived predicates in PDDL [61, 117, 141, 142]—axioms that capture implied conditions when certain predicates are satisfied—and planning over semantic graphs [137, 139, 140], which use graph structures to represent knowledge through nodes and links. In the remainder of this section, we will provide some background on these approaches and discuss their limitations.

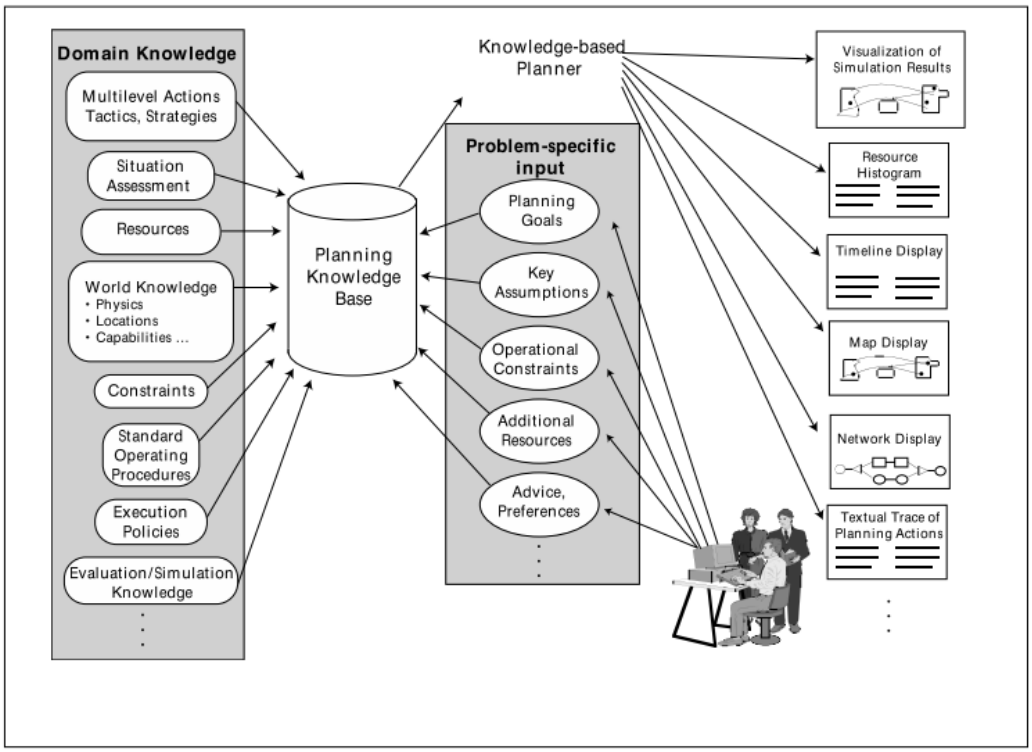


Figure 6.1: Types of knowledge sources for a knowledge-based planning system [9].

6.2.1 Typing and Derived Predicates in PDDL

6.2.1.1 Typing

Planning formalisms based on restricted first-order logic often struggle to represent the complexity of real-world domains accurately. To address this limitation, various syntactic and semantic extensions have been introduced to enhance the language's expressiveness and interpretability. Among these extensions, the introduction of typing for variables and predicates has proven particularly effective. For instance, PDDL 1.2 [61] introduced support for object typing, allowing domains to specify type hierarchies by defining abstract types. For example, by specifying `(:types building - place)`, `place` is defined as the parent type of `building`. When predicates found in preconditions and effects use the parent type (e.g., `place`), this allows all arguments of this type *or* its subtypes (like `building`) to be considered valid arguments. An example is

the predicate `in(?x - place)`, which accepts constants of type `place` as well as any subtypes of `place`, such as `building`.

However, while typing ensures that constants match their designated types, it does not consider relationships between those types as expressed by other predicates. For instance, consider the predicate `isLocatedIn(?y - office, ?x - building)`, which asserts that an office is located in a building. Since a building is a type of place, one would expect the system to infer that the office is also located in a place. However, PDDL does not reason over such relationships between types and predicates. This limitation requires domain designers to manually encode these typing relationships as predicates in addition to typing the arguments, adding complexity during planning.

6.2.1.2 Derived Predicates

A potential solution to the challenge of capturing implicit relationships between types, as well as other limitations, is the use of derived predicates. Derived predicates were introduced in PDDL [117] as a means to define the truth value of some predicates based on the values of other predicates. There has been work that demonstrate the value of derived predicates as a means of addressing the *frame problem* ([122]) by encoding rules grounded in expert knowledge [141] and helping derive indirect action effects [142]. This mechanism removes unnecessary choices from the planner's search space, freeing it to focus on more promising options. Employing derived predicates allows implicit relationships to be captured without requires explicitly enumerating those relationships within the preconditions of actions.

However, while derived predicates provide greater expressiveness than simple preconditions and effects, they are not without limitations. For example, modeling highly intricate logical dependencies—such as 'Condition B must be true only if Condition C and Condition D are true, where Condition D depends on Condition E and Condition F being true'—can become cumbersome

or infeasible due to the sheer complexity of these relationships. Such dependencies might require nested layers of derived predicates, which could increase the cognitive load for modelers and reduce the tractability of the planning domain.

6.2.2 Ontologies and Semantic Graphs

An alternate approach is the use of ontologies as a knowledge representation tool. An ontology is a formal, explicit specification of a shared conceptualization [143], allowing for the creation of structured, machine-interpretable representations of knowledge within a specific domain. Common syntactical frameworks for representing ontologies in practice include OWL (Web Ontology Language) [115] and RDF (Resource Description Framework) [116].

Ontologies have proven effective in robotics for capturing complex relationships between entities in dynamic environments. For example, in autonomous driving, ontologies enrich scene data by encoding spatial, temporal, and semantic relationships [144]. Similarly, Paulius & Sun [145] demonstrate that these benefits extend to industrial and service robotics, where knowledge about perception modules, sensor inputs, and motion planning outcomes can be captured in ontologies.

Several systems demonstrate the use of ontologies for robotics applications. KnowRob [10, 134, 135, 146] provides robots with reasoning capabilities over task knowledge (see Figure 6.2). ORO [147] serves as a commonsense knowledge base for human-robot interactions in domestic settings, while the Factory of the Future (FoF) ontology [148] supports knowledge-driven assembly processes. These systems illustrate how ontologies can enhance planning by connecting high-level conceptual knowledge to real-world robotic tasks. However, a key challenge remains—How can we express real-time perception data that respects this ontological structure in a language interpretable by planners?

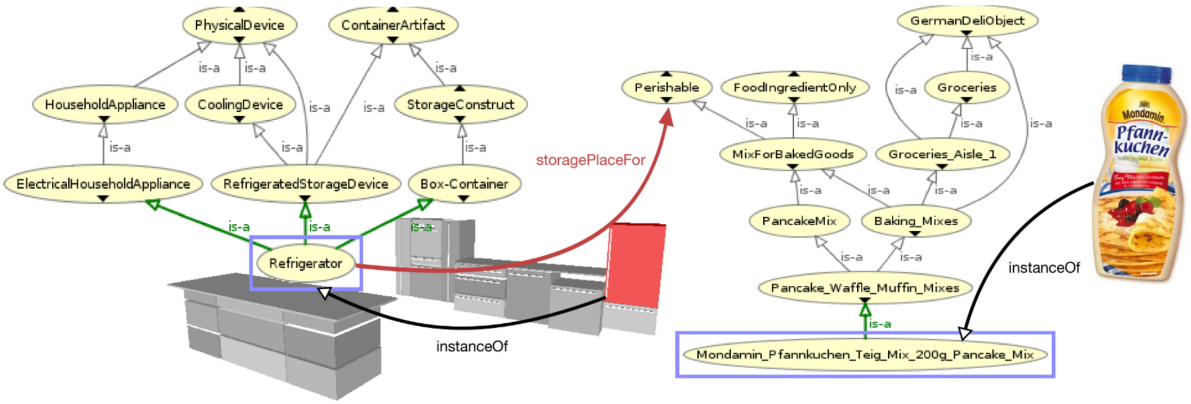


Figure 6.2: A semantic graph based on the KnowRob ontology [10].

6.2.2.1 Planning over Semantic Graphs

Task planning over semantic graphs, such as knowledge and scene graphs [149], has emerged as an approach to address the challenge of enriching world states with data during planning. One early method, proposed by Galindo et al. [137], used ontology alignment maps to integrate general and spatial knowledge for home navigation tasks. By linking nodes in a spatial graph to those in a general knowledge graph, their approach enriched the world state with implicit general knowledge, enabling more informed decision-making. Another example is Semantic Robot Programming (SRP) [140], where task and motion planning are performed over scene graphs. In SRP, a goal scene graph is constructed, and PDDL is used to generate a task plan. While effective, this approach treats each edge in the scene graph as a PDDL predicate, which can lead to significant performance bottlenecks due to the combinatorial explosion of the search space. To address this limitation, one method improves planning efficiency by reducing the size of the graph itself [139].

While promising, these methods still face challenges with real-time execution, largely due to their reliance on existing planning representations, which struggle with large state spaces. Such planners often fail to efficiently account for implicit predicates or lack native support for features of

the original data structures—such as their graph structure—that could enable more sophisticated inferences during planning. A representation language that can accommodate reasoning about both plans and knowledge graphs simultaneously may help address this issue.

6.3 Background

In this section, we introduce the state transition system model for planning and discuss key aspects of the classical planning representation, framed within this model.

6.3.1 State Transition System Model for Planning

To set the stage for the proposed language, we turn our focus to the state transition system model for representing planning problems [23]. This model captures the essential components of the planning process—states, actions, and transitions—which will form the basis for a systematic analysis comparing our approach to an existing representation language.

A state transition system model is typically defined as a tuple (S, A, γ), where:

- The *state space* $S = \{s_0, s_1, s_2, \dots\}$ is the set of all possible states. Each state $s \in S$ represents a snapshot of the world at a specific point in time, ideally capturing all relevant information, such as the location and properties of items in the world.
- The *action space* $A = \{a_0, a_1, a_2, \dots\}$ is the set of all possible actions.
- The *transition function* $\gamma : A \times S \to S$ is a partial function that maps an action and a state to the resulting state, where defined.

An action $a \in A$ is **applicable** at state $s \in S$ if $\gamma(a, s)$ is defined. A **plan**, $\pi = \langle a_1, a_2, \dots, a_n \rangle$, where $a_i \in A$, is any sequence of actions. It serves as a **solution** to the planning problem if it transitions the system from the initial state s_0 to the goal state s_g, i.e., $\gamma(a_i, s_{i-1}) = s_i$ for $i = 1, \dots, n$ and $s_n = s_g$.

6.3.2 Classical Representation

Classical and neoclassical planning have employed various representations to formalize planning domains and problems, such as the set-theoretic, classical, and state-variable representations [23]. Among these, the classical representation is most commonly associated with PDDL, and will therefore be the focus of our analysis. We will discuss how the classical representation aligns with the state transition system model introduced earlier, focusing on how states, actions, and transitions are formally captured within this framework.

6.3.2.1 World States, S

In classical planning, world states are typically represented as a conjunction of grounded literals and reside within the formalism of first-order logic, as described in Section 4.2. These literals capture knowledge about the world, such as the positions of objects and the states of sensors. World states can also be described more abstractly using predicates, which are n-ary relations between variables. For instance, a predicate like `on(x, y)` expresses that an object x is on top of another object y. These predicates provide a lifted representation, meaning they describe relationships in the world without binding the variables to specific objects. Once the variables are instantiated with constants, such as `x` $\mapsto$ `b_1` and `y` $\mapsto$ `b_2`, the predicate becomes a **grounded literal**. For example, `on(b_1, b_2)` asserts that block `b_1` is on block `b_2`. In another example, the state `AND(on(b_1, b_2), clear(b_1))` describes a world where block `b_1` is on block `b_2` and block `b_1` has no other blocks on top of it.

6.3.2.2 Actions, A

Actions are defined by schemas consisting of preconditions, effects, and parameters, as discussed in Section 4.2. Preconditions specify the conditions that must be true for the action to be

applicable, while effects describe how the action changes the world state. Both preconditions and effects are expressed using predicates, where the variables (also called parameters) can take on different values depending on the specific situation in which the action is applied.

Classical planning assumes that actions are deterministic, meaning their effects are fully specified and guaranteed to occur whenever the preconditions are satisfied. This eliminates uncertainty in action outcomes and ensures that each action has a predictable result. In practice, action schemas can include types that constrain the possible values parameters can take, as discussed in Section 6.2.1.1. Though types are not inherent to the first-order logic formalization of states, they are a supported feature in PDDL1.2 [61] and later versions.

6.3.2.3 Applicability

In classical planning, the applicability of an action in a given world state is determined by its preconditions. To check applicability, the planner checks whether the literals in the action's preconditions are a subset of the literals that are true in the current state (see Chapter 2 of [23]).

The runtime complexity of subset inclusion is $\mathcal{O}(|p| \cdot |s|)$ where $|p|$ is the number of preconditions and $|s|$ is the number of literals in the state. In extended representations that incorporate more features of first-order logic, such as disjunctions (OR) or negation, determining applicability may require more sophisticated reasoning. In these cases, the planner uses logical inference to verify whether the preconditions are logically entailed by the current state. This involves checking whether the logical conditions of the preconditions hold in the world state, which can increase the complexity of planning. In fact, logical inference in classical planning is PSPACE-complete [133], so planners often employ heuristics and optimizations to manage this complexity.

As discussed in Section 6.2.1.2, planners can use derived predicates to infer implicit conditions that affect action applicability. These derived predicates do not alter the world state

but allow some planners to infer additional facts that can influence whether an action can be applied [150].

6.3.2.4 Transition, γ

A state transition occurs when an action is applied to a world state, producing a new state. Like preconditions, the world state is represented as a set of grounded literals. When an action is applied, the transition function γ updates the current world state by modifying the set of literals based on the action's effects. Specifically, add (positive) effects introduce grounded literals into the set, while delete (negative) effects remove grounded literals from the set.

This process models a deterministic transition between states, ensuring that the next state is fully determined by the current state and the effects of the applied action.

6.4 Categorical Representation

While classical planning methods effectively represent state transitions and actions, they often struggle to manage complex semantic relationships with its syntax, especially when dealing with more structured and interconnected knowledge, as discussed in Section 6.2. To address these challenges, we propose adopting categorical logic as a more expressive and flexible formalism. An essential feature of categorical logic is its ability to precisely capture the relationship between logical syntax and semantics using functors, through a paradigm known as *functorial semantics* [151]. This approach moves beyond the limitations of first-order logic by treating the syntax of the language as an algebraic structure of its own that is bound, yet distinct, from its semantic interpretation. In this way, categorical logic offers a robust framework for representing world states by providing a formal mechanism for translating between syntactic descriptions and their corresponding semantic models. In the remainder of this section, we will introduce the

technical components of our categorical logic-based representation, followed by a case study on a manufacturing assembly task. A detailed comparison of its expressivity with PDDL will be presented in the section following that.

6.4.1 World States as C-sets

In our approach, we represent world states as structured collections of elements, attributes, and relationships, grounded in a domain-specific ontology. This allows us to capture both the physical and semantic properties of items, as well as their interactions within a world state.

$$\text{Box} \xrightarrow[\text{IsOn}]{} \text{Surface}$$

Box	**IsOn**
a	table
b	chair

```
(a::Box, ::IsOn, table::Surface)
(b::Box, ::IsOn, chair::Surface)
```

Figure 6.3: A BoxOnSurface-set, namely a C-set for the schema category BoxOnSurface.

To formalize this, we propose representing world states using **C-sets** (refer to Section 3.5.2.2 for a brief introduction). Let C denote a category, called a **schema category**, which defines the types (objects) and predicates (morphisms) in an ontology. A C-set is a functor from C to the category Set (see Example 1 in Chapter 3), where Set is the category of sets and functions. Thus, a C-set assigns types in C to sets and predicates to functions between the underlying sets. These functions can be compactly expressed as tables, resembling relational databases, where the leftmost column header refers to the schema object and the remaining column headers refer to the morphisms for which the schema object is the source. The map for each element of the set underlying each object is defined using the rows of the table. An example of a C-set schema and database is shown in Figure 6.3. Formally:

Definition 20 (C-set)**.** A C-set is a functor from the C to the category Set. The category C is a finitely generated category which is often referred to as the **schema category**.

Here, finitely generated means that the objects and morphisms in C are generated by a finite set of objects and morphisms. An example of this categorical representation is the functor Grph-set, which models directed graphs as C-sets, capturing both the nodes and edges as sets.

Example 14 (Grph-set)**.** Let $\mathsf{Grph} = E \rightrightarrows V$ be the schema category finitely generated by two objects, E and V, and two parallel morphisms, $\mathrm{src}, \mathrm{tgt} : E \to V$. Then, a Grph-set is a directed multigraph [152].

While we do not use Grph-sets to model world states and actions in our framework, this example serves to demonstrate that C-sets can generalize graphs to a wide range of combinatorial data structures. Combinatorial data structures, such as graphs, trees, hypergraphs, and, in our case, ontologies, systematically organize discrete elements and their relationships. By generalizing vertices and edges in graphs to objects in a category, we can see how C-sets allow us to model complex relational data using an arbitrary schema.

In practice, C-sets are constructed using colimits of representable functors [42]. Mathematically, a C-set can be decomposed into a colimit of representables (see Proposition 162 in [153]). Computationally, rather than instantiating all data at the time of construction—which would be memory-intensive—using colimits of representables provides an efficient alternative for concise data representation.

Recall from Definition 7 (see Section 3.1.1) that a (covariant) representable functor [103], $\mathsf{C}(A, -)$, for a given category C and object $A \in \mathsf{C}$, maps each object $X \in \mathsf{C}$ to the set of morphisms from A to X. The colimit of these functors combines relevant data by respecting existing relationships in the category, ensuring that whenever an object $A \in \mathsf{C}$ is referenced, any morphisms

originating from A are preserved in the resulting structure.

For example, in the BoxOnSurface-set, if the object Box is explicitly identified as part of a state, then the morphism IsOn (from Box to Surface) and any associated data of Surface are automatically included, since IsOn establishes a connection between Box and Surface. This ensures that both Box and Surface are consistently represented, maintaining the structural integrity of the BoxOnSurface relationships in any state or action involving Box.

The use of colimits of representable functors allows for both rich and concise expressions of states by integrating the connected knowledge at the time the state or action is constructed. A full-featured implementation of C-sets, designed for practical use in areas like robotics and planning, is available in the AlgebraicJulia ecosystem [154].

6.4.1.1 C-Set and Knowledge Graphs

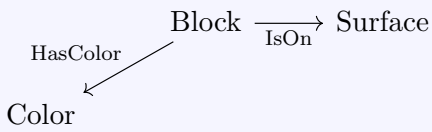
The relationship between C-sets and knowledge graphs is formalized through the *category of elements* construction, denoted as $\int X$ [155] (see Definition 11 in Section 3.3). This construction organizes the data of a C-set into a category that resembles a knowledge graph. Specifically, objects in the category of elements can be interpreted as RDF triples [156], which is, as previously mentioned, a widely used format for serializing knowledge and scene graphs. Figure 6.3 shows an example of a C-set encoded as list of RDF triplets, e.g. `(a::Box, ::IsOn, table::Surface)`, making this representation interoperable with knowledge graphs specified using RDF.

6.4.1.2 Attributed C-Set

This representation can be extended to handle non-combinatorial values, like numerical values and text, using **attributed C-sets** [41, 157]. Formally, we define an attributed C-Set as follows:

Definition 21 (Attributed C-set)**.** An attributed C-set is a functor F from a finitely presented schema category C to Set, where C is partitioned by a functor $S : \mathsf{C} \to 2$. The category, 2, has two objects, 0 and 1, and one morphism, $0 \to 1$ between them. The preimage $S^{-1}(0)$ captures the combinatorial structure (e.g., relationships between entities), while the preimage $S^{-1}(1)$ captures the attribute structure (e.g., properties of the entities). The preimage $S^{-1}(0 \to 1)$ isolates the morphisms connecting the combinatorial and attribute structures.

Example 15 (Attributed ColorBoxOnSurface-set)**.** We can extend the BoxOnSurface schema to include an attribute HasColor : Box $\to$ Color. Here, Color is an attribute object representing a set of string values (e.g., color names like "red" or "blue") that are not part of the combinatorial structure of BoxOnSurface, but provide additional descriptive information. The resulting schema, which we call ColorBoxOnSurface, includes this attribute and represents the extended structure:



Here, an attributed ColorBoxOnSurface-set, C, describes the color and location of each box in a specific world state.

Box	**IsOn**	**HasColor**
a	table	"pink"
b	chair	"yellow"

Attributed C-sets generalize C-sets by incorporating attributes alongside the combinatorial data. For the purposes of this chapter, we will assume that all referenced C-sets are attributed.

6.4.2 Actions as Spans in C-Set

In our categorical approach to planning, we model actions as spans in the category C-Set [158]. A formal definition for C-Set is provided in Definition 22.

Definition 22. The category C-Set is a category whose objects are functors $\mathsf{C} \to \mathsf{Set}$ and whose morphisms are natural transformations.

Grph-Set is an example of a C-Set category for the schema Grph from Example 14.

Example 16. The category Grph-Set is a category whose objects are Grph-set functors and whose morphisms are graph homomorphisms between directed multigraphs.

Actions are spans in C-Set. Actions extend the classical action model by introducing the additional concept of Keep, which tracks elements that remain consistent between the precondition and effect. Specifically, an action rule is a span (Pre $\hookleftarrow$ Keep $\to$ Eff), where:

- Pre represents the **precondition**, describing the requirements for the action to be applied.
- Eff represents the **effect**, describing the changes to the world state after the action is applied.
- Keep describes what remains invariant between the precondition and the effect.

Formally:

Definition 23. An **action** is a span in C-Set:

$$\text{Pre} \hookleftarrow \text{Keep} \to \text{Eff}$$

Pre refers to the precondition of the action, Eff to the effect, and Keep captures the conditions that remain unchanged between Pre and Eff.

6.4.3 Applicability using Monomorphisms

In classical planning, a precondition is satisfied by a world state if the literals in the precondition are a subset or are logically entailed with the world state. In the category-theoretic context, this notion is generalized using monomorphisms, typically denoted using a hooked arrow, $\hookrightarrow$. Monomorphisms are categorical analogs of injective functions in Set and play a key role in establishing the applicability of actions.

In $\mathsf{C\text{-}Set}$, a monomorphism is a natural transformation where every component is an injective function. For example, in the category of graphs, $\mathsf{Grph\text{-}Set}$, a monomorphism is a graph homomorphism in which both the vertex and edge maps are injective.

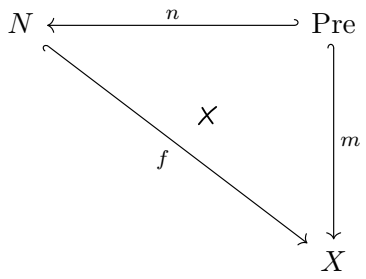
The importance of monomorphisms in planning stems from their injective nature: they ensure that no two distinct entities in the precondition can be mapped to the same entity in the world state. Thus, in categorical planning, an action is applicable if there exists a monomorphism $m : \text{Pre} \hookrightarrow X$ in $\mathsf{C\text{-}Set}$, where Pre is the action precondition and X is the current world state, we will refer to as the **match morphism**.

6.4.3.1 Actions With Negative Preconditions

In this representation language, negative preconditions—conditions that must *not* hold in the world state for an action to be applied—can be expressed using negative application conditions (NACs), a concept originating from graph rewriting [159]. NACs extend the action rule by specifying patterns that must not be present in the world state before the action can be applied. This concept generalizes negative preconditions from classical planning to the categorical setting.

Formally, a NAC is defined by a monomorphism $n : \text{Pre} \hookrightarrow N$, where N is a structure that should not appear in the world state. The action is considered **not applicable** if there exists a commuting triangle involving n, the match morphism m, and a morphism $f : N \hookrightarrow X$ which

captures the embedding of N into the current world state X. This non-commutative condition is expressed as $f \circ n \neq m$ in C-Set. Diagrammatically, this can be understood as:

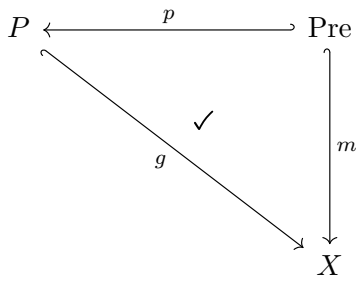


In this diagram, the morphism f represents the presence of the forbidden structure N in the world state X. The action is only applicable if there is no way to complete the commuting triangle between Pre, N, and X via f. In other words, the NAC ensures that certain configurations of the world state are avoided, preventing the action from being applied in inappropriate contexts.

Furthermore, negative conditions can be composed, allowing for the specification of distinct negative preconditions that must simultaneously be absent from the world state for an action to be applied. This compositionality provides flexibility in defining more complex applicability constraints.

6.4.3.2 Extending Preconditions

Positive application conditions (PACs) serve as the complement to negative application conditions (NACs). PACs extend preconditions by introducing additional conditions that must hold in the world state. Formally, this is achieved by defining a morphism $p : \text{Pre} \hookrightarrow P$, where P is the structure that must be present in the world state, such that $g \circ p = m$, where $g : P \hookrightarrow X$ commutes. Diagrammatically, this can be understood as:



This construction is particularly useful in scenarios where specific application preconditions need to be added, without altering the general specification of the rule itself. PACs allow for modular and flexible rule extensions, ensuring that the core rule remains intact while accommodating additional constraints relevant to the current context or environment.

6.4.4 Transition using Double-Pushout (DPO) Rewriting

Double-pushout (DPO) rewriting is a formal approach in graph rewriting that uses category theory to define transformations of graphs through two pushout constructions [160]. DPO rewriting is a mathematically rigorous approach within the broader field of graph rewriting, where graphs are systematically transformed according to predefined rules. This method naturally extends graph rewriting to C-set rewriting, allowing for transformations as it relates to any schema category C [42]. In the context of planning, action rules are modeled as DPO rewrite rules. For a given action rule, the DPO algorithm finds a monomorphism from the precondition to the world state. Two pushout squares are then computed which results in a transformed C-set.

6.4.4.1 Pushouts in C-Set

DPO rewriting relies on the fundamental category-theoretic concept of a pushout (see Definition 10). A pushout is the colimit of a diagram shaped as a span ($\bullet \leftarrow \bullet \rightarrow \bullet$). Given a span $X \leftarrow Z \rightarrow Y$, the pushout produces an object that effectively joins X and Y along their shared subobject Z, resulting in the object $(X \cup Y)/Z$. In C-Set, a pushout is computed by taking the

disjoint union of the underlying sets, then quotienting by the set Z. This allows for consistent and coherent merging of world states. An example of a pushout in C-Set is shown in Example 17.

Example 17 (Pushout in C-Set: Merging Worlds in BlocksOnSurface-Set)**.** Consider the schema category, BoxOnSurface. This schema models the data about blocks on surfaces.

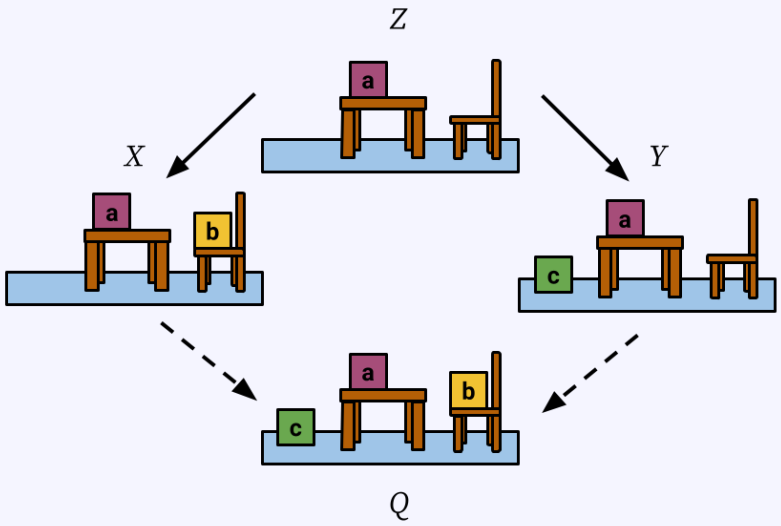


Figure 6.4: An illustration of a single pushout in BoxOnSurface-set.

We define three objects in BoxOnSurface-Set, denoted X, Y, and Z, which are functors from the schema BoxOnSurface to Set. These objects represent different configurations of blocks and surfaces, along with the relationships between them using the morphism IsOn.

Let X be a BoxOnSurface-set specifying two blocks and two surfaces. We define $X(\text{Box}) = \{a, b\}$ and $X(\text{Surface}) = \{\text{table}, \text{chair}\}$. The morphism $X(\text{IsOn})$ is defined as:

$$\text{IsOn}(a) = \text{table} \quad \text{and} \quad \text{IsOn}(b) = \text{chair}$$

Let Y be a BoxOnSurface-set representing two blocks and two surfaces. We define $Y(\text{Box}) = \{a, c\}$ and $Y(\text{Surface}) = \{\text{table}, \text{floor}\}$. The morphism $Y(\text{IsOn})$ is defined as:

$$\text{IsOn}(a) = \text{table} \quad \text{and} \quad \text{IsOn}(c) = \text{floor}$$

Let Z be a BoxOnSurface-set representing a configuration with one block and one surface. Specifically, $Z(\text{Box}) = \{a\}$ and $Z(\text{Surface}) = \{\text{table}\}$, with the morphism $Z(\text{IsOn})$

defined as:

$$\text{IsOn}(a) = \text{table}$$

The morphism $Z \to X$ maps the block a in Z to the block a in X, and the surface table in Z to the surface table in X. The morphism $Z \to Y$ maps the block a in Z to the block a in Y, and the surface table in Z to the surface table in Y.

The pushout of the span $X \leftarrow Z \to Y$ in BoxOnSurface-Set is constructed as follows:

1. **Take the Disjoint Union**. The disjoint union of X and Y results in the following sets:

   $$\text{Blocks: } \{a_X, a_Y, b, c\}, \quad \text{Surfaces: } \{\text{table}_\text{X}, \text{table}_\text{Y}, \text{chair}, \text{floor}\}.$$

2. **Glue Along** Z. We identify the blocks $a_X = a_Y$, as both are mapped from the block a in Z. Similarly, we identify the surfaces $\text{table}_X = \text{table}_Y$, as both are mapped from the surface table in Z.

3. **Compute Pushout Object**. The resulting pushout object is a BoxOnSurface-set, denoted as Q, with:

   $$Q(\text{Box}) = \{a, b, c\}, \quad Q(\text{Surface}) = \{\text{table}, \text{chair}, \text{floor}\}.$$

   The morphism $\text{IsOn} : Q(\text{Box}) \to Q(\text{Surface})$ is defined as:

   $$\text{IsOn}(a) = \text{table}, \quad \text{IsOn}(b) = \text{chair}, \quad \text{IsOn}(c) = \text{floor}$$

   The following commutative diagram represents the pushout construction:

$$\begin{array}{ccc} Z & \longrightarrow & X \\ \downarrow & \ulcorner & \downarrow \\ Y & \longrightarrow & Q \end{array}$$

The pushout in this example combines the structures of X and Y by identifying the common block a and surface table. The resulting pushout object reflects a combination of the relationships in both X and Y, where block a is on the table, block b is on the chair, and block c is on the floor.

The ability to compute pushouts within C-Set is crucial for planning, as it enables systematic

merging of state data that respect the structural relationships enforced by the world state's schema. An important mathematical fact is that the category C-Set is a *topos* [161], making it an especially well-behaved category in which all limits and colimits exist.

6.4.4.2 DPO Rewriting Algorithm

In Example 17, we demonstrated how pushouts can be used to combine two structures, X and Y, along a shared substructure Z within the context of C-sets. This process allowed us to merge the relationships between blocks and surfaces, preserving the schema's structure by forming a new object through the pushout operation.

We now extend this idea by applying it to DPO rewriting, where two consecutive pushout operations transform a world state through the application of an action rule. The diagram below illustrates this process:

$$\begin{array}{ccccc} \text{Pre} & \xleftarrow{l} & \text{Keep} & \xrightarrow{r} & \text{Eff} \\ {\scriptstyle m}\downarrow & & {\scriptstyle f}\downarrow & & \downarrow \\ X & \longleftarrow & Z & \longrightarrow & Y \end{array}$$

Here, the left pushout square represents the deletion of part of the world state (X) based on the preconditions (Pre), while preserving certain elements (Keep) and stores this result in Z. The right pushout square then merges the effects (Eff) of the action and Z, resulting in the new world state (Y). This approach ensures that structural integrity is maintained throughout the transformation, as the pushout construction guarantees that the relationships in the schema C are respected.

More specifically, the first step is to find a monomorphism, m, that matches Pre in X, as described in the previous subsection. The next step is to compute the **pushout complement**, $f : \text{Keep} \to Z$, given the morphisms l and m. The pushout complement manages the deletion of entities that form the complement Keep/Pre. Because m is a monomorphism, the pushout

Algorithm 2: Double-Pushout (DPO) Rewriting

Require: (action rule) Pre $\overset{l}{\hookleftarrow}$ Keep $\overset{r}{\to}$ Eff $\in$ C-Set
Require: (world) $X \in$ C-Set
m = `FindHomomorphism`(Pre,X,`monic`=True)
f = `ComputePushoutComplement`(l, m)
Y = `ComputePushout`(f, r)

complement exists and is unique up to isomorphism, [42] provided that the **identification** and **dangling conditions** are satisfied when defining an action.

Identification condition. The identification condition requires that when an element in the component of an action is uniquely identified across all structures involved in the rewrite. Specifically, this condition prevents any unintended overlaps or merges of distinct elements during the rewriting process. For example, if a robot picks up a block, the identification condition ensures that only the intended block is removed from the table and that no other blocks, that may also have been on the table, are incorrectly removed.

Dangling condition. The dangling condition, on the other hand, requires that when elements are deleted, the deletion does not leave any "dangling edges" or unresolved references in the resulting structure. For example, in a C-set representing items in a room, if an item is deleted, all relationships involving that item must also be deleted. Otherwise, there would be references to non-existent objects, which would lead to an invalid world state.

Once the three sides of the square (l, m, and the pushout complement f : Keep $\to Z$) are constructed, the pushout square is completed by a unique map $Z \to X$. The final step involves computing the right pushout square given Keep $\to Z$ and r, which produces the new world state. Pseudocode for the DPO rewriting procedure is given in Algorithm 2.

Complexity of DPO Rewriting. A subroutine, `FindHomomorphism()`, is used to identify specific morphisms when a morphism has not been manually specified, such as in $m : \text{Pre} \hookrightarrow X$. The runtime complexity of current implementations of `FindHomomorphism()` is $\mathcal{O}(n^k)$, where k is the size of Pre and n is the size of X, restricted by the subschema involved in Pre [42]. Here, the size of a C-set refers to the total number of elements across all types in the schema; mathematically, this corresponds to the number of objects in the category of elements.

The subroutines, `ComputePushoutComplement()` and `ComputePushout()`, leverages the union-find algorithm, which has a complexity of $\mathcal{O}(p \cdot \alpha(s))$ (see Section 21.4 in [162]). For a span $X \leftarrow Z \rightarrow Y$, p represents the number of union-find operations required to manage equivalences among elements in X and Y as specified by the relationships in Z. Here, s is the total number of elements across the sets X and Y, and $\alpha(s)$ is the inverse Ackermann function, which is near-constant in practice, providing effectively linear runtime complexity for computing pushouts in typical applications.

6.4.4.3 Use in Planning

In summary, the categorical approach provides a more structured and relational representation through C-sets, spans, and DPO rewriting, while the classical approach relies on simpler set-based operations and logical conjunctions. These differences impact the expressiveness and scalability of each method in handling structured knowledge in planning. Table 6.1 provides a map between concepts in the categorical represention and the classical representation aligned with the state transition model for planning.

While both the categorical and classical approaches differ significantly in how they represent states, actions, applicability, and transitions, they rely on the same underlying state transition system model. Because of this, planning algorithms that operate on this model can be naturally

Representation	Categorical	Classical
State, S	C-set	Conjunction of literals
Action, A	Spans in C-Set containing: Preconditions, Keep, Effects	Action model containing: Preconditions, Effects
Applicability	Monomorphisms in C-Set	Subset inclusion
Transition, γ	DPO rewriting	Set-based addition and subtraction

Table 6.1: Terminology map between the categorical representation and classical representation for planning

extended to work within this categorical framework. In practice, this means that classical planning algorithms, such as search-based methods (e.g., A* or breadth-first search) and heuristic-based approaches, can be adapted to operate within this categorical setting. Algorithm 4 in Appendix A.2 provides pseudocode for implementing a depth-first search algorithm for forward planning within the categorical framework.

6.5 Comparison for a Manufacturing Assembly Task

In this section, we highlight the representational differences between the categorical and classical approaches to planning through an example in a small robotic assembly domain. Imagine a robot tasked with assembling objects, such as a table, using parts like legs, screws, and tabletops. This example involves assembling these components into a complete table, where implicit relationships between the parts (e.g., parts and assemblies being colocated) must be preserved throughout the process. An illustration of this domain is found in Figure 6.5.

6.5.1 Planning Domain

The categorical representation of a planning domain requires defining a schema category prior to defining states and actions with respect to that domain.

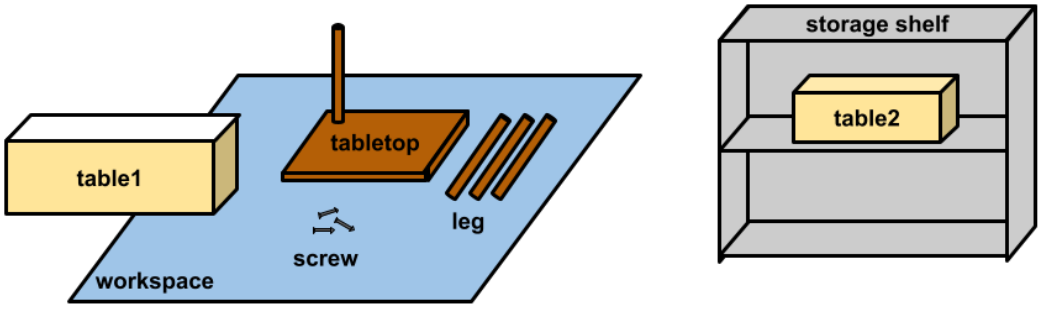


Figure 6.5: An illustration of the Assembly domain.

6.5.1.1 Categorical Representation, Schema Category Assembly

The schema for the robot assembly domain, called Assembly, consists of three objects representing the different components involved in the assembly process:

- **Part**: Represents any physical component.
- **Type**: Represents the type of the part (e.g., tabletop, leg, screw).
- **Assembly**: Represents the collection of parts that get assembled into a product (e.g., a table).
- **Location**: Represents spatial areas where parts or assemblies can be placed (e.g., workspace, storage shelf).

The schema also includes morphisms representing the relationships between these objects:

- **HasType**: A morphism representing that specifies the type of the part (e.g., "part HasType leg")
- **IsPartOf**: A morphism representing that a part is part of an assembly (e.g., "leg IsPartOf table").
- **PIsLocatedAt**: A morphism representing the spatial attribute of parts, hence the prefix "P", and locations (e.g., "screw PIsLocatedAt workspace").
- **AIsLocatedAt**: A morphism representing the spatial attribute of assemblies, hence the prefix

"A", and locations (e.g., "table AIsLocatedAt workspace").

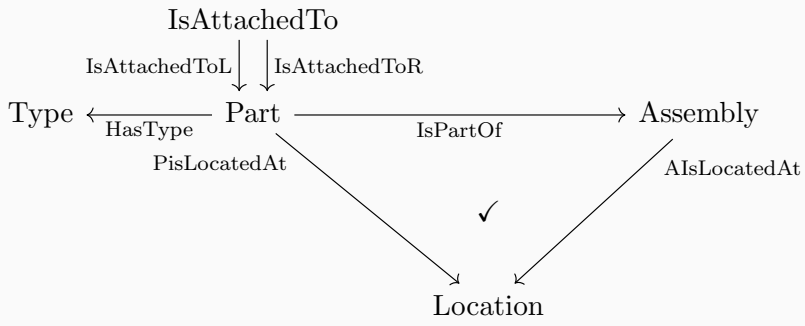
Lastly, the schema includes objects and morphisms that take the shape of a span. This allows the schema to encode relations between elements while within C-Set. The object at the apex represents the relation and the two morphisms represent the left and right-side of the relation. For this schema, there exists a relation:

- **isAttachedTo**: A relation representing one part as being attached to another part (e.g., "leg isAttachedTo tabletop").

This pattern allows for parts to not being involved in an attachment, or a parts to be attached to multiple parts.

Given this setting, we can define a schema category called Assembly that describes a robot assembly domain.

Definition 24 (Assembly)**.** The schema category, Assembly, for the robotic assembly domain has objects {Part, Assembly, Location, Type} and morphisms:



Notice that a commuting diagram $\text{AIsLocatedAt} \circ \text{isPartOf} = \text{PIsLocatedAt}$, symbolized by ✓, is enforcing a constraint that suggests that all the parts belonging to an assembly should be at the same location as the assembly in order for the diagram to commute.

6.5.1.2 Classical Representation, Domain Types and Predicates

The same robotic assembly domain can be represented in PDDL by defining types and predicates. In PDDL, the domain defines types of objects (e.g. `part`, `assembly`, `location`, `type`) and predicates that capture relationships and properties. These elements correspond directly to the objects and morphisms in the categorical schema but are expressed using the syntax and semantics of classical planning.

```
(define (domain assembly)

  (:types
    part
    assembly
    location
    type
  )

  (:predicates
    (HasType ?p - part ?t - type)
    (IsPartOf ?p - part ?a - assembly)
    (PIsLocatedAt ?p - part ?l - location)
    (AIsLocatedAt ?a - assembly ?l - location)
    (IsAttachedTo ?p1 - part ?p2 - part)
  )
  ...
```

In the predicates, types are declared as a way to constrain the possible elements that can be involved in their grounded versions. For example, `HasType` captures the relationship between a part and its type, similar to how the morphism HasType operates in the categorical schema. Relationships between predicates, such as `IsPartOf` and `AIsLocatedAt`, based their common argument types are not considered explicitly.

6.5.2 Initial State

Now that we have defined the planning domain, we can define an initial state to a planning problem.

6.5.2.1 Categorical Representation, Object X in Assembly-Set

A state in this domain is an object in Assembly-Set. In this example, we take a state, X, such that the sets underlying each object are:

- $X(\text{Part}) : \{\text{part\_}i\}$ where $i = 1 : 10$
- $X(\text{Type}) : \{\text{leg}, \text{screw}, \text{tabletop}\}$
- $X(\text{Assembly}) : \{\text{table\_1}, \text{table\_2}\}$
- $X(\text{Position}) : \{\text{workspace}, \text{storage\_shelf}\}$

The functions in Set that underpin this state specify the starting locations of each part and the assembly that each part belongs to. For this example, we say that one leg (leg_1) of the table is already attached to tabletop (tabletop) via a screw (screw_1). This scenario is expressed in Table 6.2. Because Type does not have an outgoing morphism, it does not need a table defining the element maps.

Table 6.2: The state X in Assembly-Set.

Part	**HasType**	**IsPartOf**	**PIsLocatedAt**
part_1	leg	table_1	workspace
part_2	leg	table_1	workspace
part_3	leg	table_1	workspace
part_4	leg	table_1	workspace
part_5	leg	table_2	storage_shelf
part_6	screw	table_1	workspace
part_7	screw	table_1	workspace
part_8	screw	table_1	workspace
part_9	screw	table_1	workspace
part_10	tabletop	table_1	workspace

Assembly	**AIsLocatedAt**
table_1	workspace
table_2	storage_shelf

IsAttachedTo	**L**	**R**
1	part_6	part_10
2	part_6	part_1
3	part_1	part_10

6.5.2.2 Classical Representation, PDDL Problem file

In a PDDL problem file for a robotic assembly domain, the objects must include specific parts like `leg_1`, `table_1`, and locations like `workspace` and `storage_shelf`. The initial state

would list facts such as `leg_1` being part of `table_1` and currently located in the workspace. The PDDL syntax for the initial state is:

```
(define (problem assembly-problem)
  (:domain assembly)

  (:objects
    part_1 part_2 part_3 part_4 part_5
    part_6 part_7 part_8 part_9 part_10 - part
    leg screw tabletop - type
    table_1 table_2 - assembly
    workspace storage_shelf - location
  )

  (:init
    ;; Type assignments
    (HasType part_1 leg) (HasType part_2 leg) (HasType part_3 leg)
    (HasType part_4 leg) (HasType part_5 leg) (HasType part_6 screw)
    (HasType part_7 screw) (HasType part_8 screw) (HasType part_9 screw)
    (HasType part_10 tabletop)

    ;; Part-of relationships
    (IsPartOf part_1 table_1) (IsPartOf part_2 table_1)
    (IsPartOf part_3 table_1) (IsPartOf part_4 table_1)
    (IsPartOf part_5 table_2) (IsPartOf part_6 table_1)
    (IsPartOf part_7 table_1) (IsPartOf part_8 table_1)
    (IsPartOf part_9 table_1) (IsPartOf part_10 table_1)

    ;; Location assignments
    (PIsLocatedAt part_1 workspace) (PIsLocatedAt part_2 workspace)
    (PIsLocatedAt part_3 workspace) (PIsLocatedAt part_4 workspace)
    (PIsLocatedAt part_5 storage_shelf) (PIsLocatedAt part_6 workspace)
    (PIsLocatedAt part_7 workspace) (PIsLocatedAt part_8 workspace)
    (PIsLocatedAt part_9 workspace) (PIsLocatedAt part_10 workspace)

    (AIsLocatedAt table_1 workspace)
    (AIsLocatedAt table_2 storage_shelf)

    ;; Attachment relationships
    (IsAttachedTo part_6 part_10) ; screw_1 attaches tabletop to screw_1
    (IsAttachedTo part_6 part_1)  ; screw_1 attaches leg_1 to screw_1
    (IsAttachedTo part_1 part_10) ; leg_1 attaches to tabletop
  )
)
```

It is theoretically possible to define semantically invalid literals within this syntax. For example, there is nothing preventing a problem designer from defining `(PIsLocatedAt part_3 storage_shelf)`, which could technically invalidate the state if the parts and their assembly (e.g., `(AIsLocatedAt table_1 workspace)`) are required to be in the same location.

While it is possible to enforce such constraints within the PDDL domain file using additional constructs or extensions, such as existential quantifiers, like so:

```
(:constraints
  (forall (?p - part ?a - assembly ?l - location)
    (implies
      (and (IsPartOf ?p ?a) (PIsLocatedAt ?p ?l))
      (AIsLocatedAt ?a ?l)
    )
  )
)
```

These constraints are only validated during the planning process and do not prevent a domain and problem designer from defining an initial state with this error. Moreover, each PDDL extension introduces additional computational complexity to the search space (see Chapter 2.4.3 in [23]). For this specific constraint, the time complexity of checking whether the constraint is respected is $\mathcal{O}(|P| + |A| + |L|)$, where $|P|$, $|A|$, and $|L|$ represent the number of parts, assemblies, and locations, respectively. This time is added to the runtime required to check the applicability of the action.

This added complexity can make the planning process more resource-intensive, as the planner must enforce these constraints at each state transition, potentially slowing down the search and making it more difficult to find a solution. In contrast, enforcing the constraint in the categorical approach incurs no additional complexity, as it is integrated into the construction of the states themselves.

6.5.3 Action

Now that the domain has been defined, it is possible to define an action in this domain. We choose to model the action of attaching a leg to the tabletop, which we call, **AttachPart**.

6.5.3.1 Categorical representation, Span in Assembly-Set

To define an action, we must define the Pre, Keep, and Eff objects and the morphisms $l : \text{Keep} \hookrightarrow \text{Pre}$ and $r : \text{Keep} \to \text{Eff}$.

Precondition. The precondition of this action is that there must exist a *free* leg, a *free* screw, and a tabletop all belonging to the same assembly. We assume that (i) each assembly has a fixed number of legs and screws assigned to it and (ii) a part being free means that it is *not* captured under the IsAttachedTo relation. To represent this condition, we need to not only define Pre, but also a NAC, $n : \text{Pre} \hookrightarrow N$ (see Section 6.4.3.1). The precondition Pre is defined in Table 6.3. The negative condition N includes the Pre with the addition of the conditions in Table 6.4.

Table 6.3: Pre (and Keep) for action AttachPart

Part	**HasType**	**IsPartOf**	**PIsLocatedAt**
part_x	leg	assembly_x	?x
part_y	screw	assembly_x	?x
part_z	tabletop	assembly_x	?x

Assembly	**AIsLocatedAt**
assembly_x	?x

Table 6.4: NACs (in addition to Pre) for action AttachPart.

IsAttachedTo	**L**	**R**
1	part_x	part_y
2	part_x	part_z
3	part_y	part_z

The _x, _y, _z suffix appended to each part's label symbolizes that the precondition is not referring to specific parts such as a particular leg or screw (e.g., part_1 or part_2) but rather to arbitrary yet distinct parts. The assembly, however, must be the same for all three parts, so its label is consistently appended with the suffix _x to indicate this shared reference.

Additionally, observe that in this precondition, the location of each part and the location of the assembly are left as variables (`?x`). This occurs as a result of taking a colimit over representable functors, specifically $\text{Hom}(\text{Part}, -)$ and $\text{Hom}(\text{Assembly}, -)$. The colimit unifies information about

the locations by creating a minimal representation of the relationships across objects. The use of the same variable `?x` reflects this shared structure, as the colimit allows overlapping data from the relationships between parts and assemblies to represent their common locations.

We define the morphism $n : \text{Pre} \hookrightarrow N$ to simply be the self-identification of the rows of Pre within N. For instance:

$$(\text{leg\_x}, \text{assembly\_x}, ?x) \in \text{Pre} \mapsto (\text{leg\_x}, \text{assembly\_x}, ?x) \in N$$

Effect. Because the desired Eff is to attach these parts to the tabletop, we can set the Eff to be equal to N, namely Eff $= N$. The object N specifies the attachment relationships between the parts, and therefore, can be used as-is to specify the desired effect.

Keep. The Keep condition identifies the parts of Pre and Eff that remain the same. Given that we would like the free parts, e.g. legs, screws, and tabletops, to be the same parts we attach via Eff, we can set Keep to be equal to Pre, namely Keep = Pre. This is because the conditions expressed in Pre are exactly the conditions that should remain the same. Because Eff $= N$ and Keep = Pre, we can define the morphism $l : \text{Keep} \hookrightarrow \text{Pre}$ to be equal to the identity morphism $id_{\text{Pre}} : \text{Pre} \hookrightarrow \text{Pre}$ and the morphism $r : \text{Keep} \to \text{Eff}$ to be equal to n.

6.5.3.2 Classical Representation, Action in PDDL Domain

In PDDL, the action `AttachPart` can be specified with preconditions and effects that mirror the categorical representation's span in Assembly-Set. The `AttachPart` action in this representation models the process of attaching a leg to a tabletop in the same robotic assembly domain. The action specifies a set of parameters that include the leg, screw, tabletop, assembly, and location. These parameters define the objects that the action will interact with during execution. The PDDL syntax for the action is:

```
(:action AttachPart
  :parameters (?leg - part ?screw - part ?tabletop - part
  ?assembly - assembly ?loc - location)
  :precondition (and
      (HasType ?leg leg)
      (HasType ?screw screw)
      (HasType ?tabletop tabletop)
      (IsPartOf ?leg ?assembly)
      (IsPartOf ?screw ?assembly)
      (IsPartOf ?tabletop ?assembly)
      (PIsLocatedAt ?leg ?loc)
      (PIsLocatedAt ?screw ?loc)
      (PIsLocatedAt ?tabletop ?loc)
      (AIsLocatedAt ?assembly ?loc)
      (not (IsAttachedTo ?leg ?screw))
      (not (IsAttachedTo ?leg ?tabletop))
      (not (IsAttachedTo ?screw ?tabletop))
  )
  :effect (and
      (IsAttachedTo ?leg ?screw)
      (IsAttachedTo ?leg ?tabletop)
      (IsAttachedTo ?screw ?tabletop)
  )
)
```

The preconditions for this action ensure that the leg, screw, and tabletop involved in the action are part of the same assembly and are all located at the same place (`?loc`). Specifically, the preconditions check that the leg is of type leg, the screw is of type screw, and the tabletop is of type tabletop. Additionally, it verifies that each of these parts is part of the specified assembly (`?assembly`) and that they are all located at the same location (`?loc`). Crucially, the preconditions also include negative conditions (`not`), which ensure that the leg, screw, and tabletop are not already attached to each other before the action is executed.

The effect of the action specifies the changes to the world state that result from successfully performing the action. In this case, the action results in the leg being attached to both the screw and the tabletop, and the screw being attached to the tabletop. These new relationships are represented by the IsAttachedTo predicate, which updates the state to reflect that the parts are now physically connected as part of the assembly.

Notice that the constraint enforcing the colocation of parts and assemblies need to be made

explicit in the action if a PDDL constraint is not specified.

6.5.4 Next State

Now that the action **AttachPart** has been defined, we can check the applicability of this action and apply it to the initial state defined earlier in this section.

6.5.4.1 Categorical Representation, $Y \in$ Assembly-Set

To determine the applicability of the action in the categorical representation, we need to search for a match, m : Pre $\hookrightarrow X$, taking into account any NACs or PACs. This can be done using an implementation of `FindHomomorphism()` (see Section 6.4.4.2). A reasonable candidate for m is shown in Table 6.5. The rows highlighted in yellow identify the valid matches for each precondition (i.e. per row in the Pre database).

Table 6.5: Match m from Pre to the initial state X.

Part	HasType	IsPartOf	PIsLocatedAt
part_1	leg	table_1	workspace
part_2	leg	table_1	workspace
part_3	leg	table_1	workspace
part_4	leg	table_1	workspace
part_5	leg	table_2	storage_shelf
part_6	screw	table_1	workspace
part_7	screw	table_1	workspace
part_8	screw	table_1	workspace
part_9	screw	table_1	workspace
part_10	tabletop	table_1	workspace

Assembly	AIsLocatedAt
table_1	workspace
table_2	storage_shelf

After executing the DPO rewriting algorithm (see Algorithm 2), the new state contains the facts about the new attachments as shown in Table 6.6.

Table 6.6: Modified IsAttachedTo table within the new state Y.

IsAttachedTo	**L**	**R**
1	part_6	part_10
2	part_6	part_1
3	part_1	part_10
4	part_7	part_10
5	part_7	part_2
6	part_2	part_10

6.5.4.2 Classical Representation, Applicability and State Update

In PDDL, the applicability of an action is determined by evaluating the action's preconditions against the current state of the world. This done by checking that there is a consistent match between the types of the parameters in the preconditions and the objects in the state. The match is determined by binding the parameters in the action's preconditions to specific objects in the state that satisfy all the conditions, which is similar to finding a monomorphism in the categorical approach. However, in PDDL, this process is entirely dependent on the domain designer's definition of the action, without the inherent structural guarantees provided by the categorical framework.

After the action is determined to be applicable, a new state is generated by adding the effects into the new state. This would add the following attachment relationships to the world state:

```
...
;; Attachment relationships
(IsAttachedTo part_6 part_10) ; screw_1 attaches tabletop to screw_1
(IsAttachedTo part_6 part_1)  ; screw_1 attaches leg_1 to screw_1
(IsAttachedTo part_1 part_10) ; leg_1 attaches to tabletop
(IsAttachedTo part_7 part_10) ; screw_2 attaches tabletop to screw_2
(IsAttachedTo part_7 part_2)  ; screw_2 attaches leg_2 to screw_2
(IsAttachedTo part_2 part_10) ; leg_2 attaches to tabletop
```

Both the categorical representation and the PDDL approach ultimately arrive at the same next state, demonstrating their equivalence in expressivity for this example. However, the categor-

ical representation inherently enforces structural constraints throughout the process. In PDDL, the planner must check individual conditions at each step. While PDDL is highly flexible and well-suited for a wide range of planning problems, enforcing such complex constraints can result in increased computational complexity. In contrast, the categorical approach automatically enforces the structural constraints, such as typing and transitive closure, by construction. Although operations like finding homomorphisms can introduce higher computational demands (especially for large structures), this complexity is more predictable and fixed.

6.6 Discussion

The comparison between categorical and classical (PDDL) representations highlights distinct strengths and limitations in how each approach addresses planning problems, particularly in specifying states and actions, enforcing implicit conditions based on relational structure, and facilitating analogical plan transfer.

6.6.1 Advantages of the Categorical Structure for Planning

Enforcement of Structural Integrity. A key advantage of the categorical representation is its ability to automatically manage implicit relationships and guarantee structural integrity. By leveraging the structure defined within the schema category, the categorical approach naturally captures dependencies without having to explicitly enumerate of every condition and relationship, as required in PDDL. In contrast, PDDL demands that all relationships and conditions be manually specified and managed. This increases the likelihood of errors in domains with complex relational structures where implicit dependencies might be overlooked.

Preservation of Structure During State Transitions. In classical approaches, it is critical to explicitly define and track all relevant state changes in PDDL actions, which can increase the potential

for inconsistency or contradiction, especially as the domain's complexity grows. The categorical approach, through the use of DPO rewriting, provides a rigorous and structured mechanism for applying actions and updating states.

Predictable Computational Complexity. Another advantage of the categorical representation is more predictable computational complexity, which arises from well-defined algebraic operations such as DPO rewriting. These operations are grounded in the semantics of category theory, leading to a more predictable computational analysis. In contrast, PDDL, while highly flexible, can introduce unpredictable interactions between its extensions—such as constraints, type systems, or derived predicates—resulting in variable and often higher computational complexity when determining applicability of actions and computing new states. Although PDDL's flexibility supports a wide range of customizations, this frequently comes at the cost of increased and less predictable computational demands.

Support for Analogical Plan Transfer. The classical PDDL representation for planning lacks built-in support for isolating domain ontologies in a structured way. In contrast, the categorical representation enforces a robust formalism within which maps between domains can be rigorously defined, thereby better supporting analogical transfer.

6.6.2 Limitations of the Categorical Structure for Planning

Reliance on Advanced Mathematical Theory. The categorical representation is grounded in abstract mathematical concepts such as categories, functors, and pushout constructions, which can present a steep learning curve for practitioners. In contrast, the classical representation relies on more accessible set-based operations. In simpler planning domains, where relationships and dependencies are straightforward, the categorical approach may introduce unnecessary overhead,

making the simplicity and directness of PDDL a more practical choice. The next chapter will discuss a syntax and programming framework designed to alleviate the need for deep mathematical expertise in category theory in order to make use of this representation language.

Rigid Mathematical Semantics. While the structure provided by the categorical representation offers significant benefits, it also imposes limitations on the expressivity of domain ontologies to ensure mathematical coherence. One notable restriction is that predicates are mathematically tied to functions, requiring patterns such as spans for relations to accommodate non-function-based semantics. In contrast, the classical representation imposes no rigid formalism on the definition of domains and states, allowing for greater flexibility and fewer design constraints.

Higher Computational Complexity. Planning algorithms that align with the state transition model can generally be adapted to use the categorical representation, but additional computational complexity is introduced when using the DPO rewriting technique. The `FindHomomorphism()` subroutine, analogous to determining the applicability of actions in PDDL, involves more complex structural matching. Applicability checks in PDDL typically involve subset checks or logical entailment, which are generally bilinear in the sizes of the preconditions and the current state. In contrast, the exponential time complexity of the `FindHomomorphism()` subroutine reflects the additional computational effort required to find an injective match in a categorical setting, often resulting in higher computational costs compared to basic applicability checks in PDDL.

6.7 Conclusion

The chapter discussed the challenges and opportunities in planning with structured knowledge, focusing on the balance between abstraction and representation. The categorical approach was presented as a powerful alternative to classical planning methods, particularly in handling

complex relational structures through C-sets and DPO rewriting. The primary purpose of setting up this categorical language was to support analogical plan transfer which requires incorporating structured, highly connected knowledge into the planning process. The categorical framework offers several advantages aside from this, including the ability to manage implicit relationships and preserve structural integrity during state transitions. The comparison with classical methods, such as PDDL, highlighted the strengths of the categorical approach in enforcing relational structures automatically, contrasted with the explicit control and potential for error in classical representations. Ultimately, the chapter suggests that while the categorical approach may introduce overhead in simpler domains, it provides significant benefits that can enable analogical plan transfer and the use of structured knowledge in planning.

Chapter 7: Analogical Transfer using Data Migrations

Analogical transfer presents an advanced method of plan reuse, where a plan π_S generated within source planning domain D_S is adapted to a new target planning domain D_T by forming an analogy between the two domains. Unlike sequential transfer, which involves the direct composition of plans, or hierarchical transfer, which integrates plans within predefined modular structures, analogical transfer focuses on identifying and exploiting underlying patterns that are common to both the source and target domains. This type of transfer transforms the source plan, π_S, into a new, contextually appropriate plan, π_T—tailored to the target domain's ontology. Formally:

Definition 25 (Analogical Transfer)**.** Let π_S be a task plan that is the solution to a planning problem, P_S, expressed using the source domain ontology, D_S. Let D_T be the target domain ontology. Analogical transfer involves finding a structure-preserving map $F : \mathsf{D}_T \to \mathsf{D}_S$ that maps the ontological types and predicates of D_T to corresponding types and predicates in D_S. The transfer function Δ_F, induced by F, maps the planning problem, P_S, and plan, π_S, as follows:

$$\Delta_F(\mathrm{P}_S) = \mathrm{P}_T, \quad \Delta_F(\pi_S) = \pi_T$$

where P_T is the transferred planning problem and π_T is the transferred task plan that respects the target domain ontology D_T.

Valid Transfer. Analogical transfer is considered successful if the transferred plan π_T solves the transferred problem P_T in the target domain:

$$\mathrm{valid}(\mathrm{Final}(\pi_T), \mathrm{Goal}(P_T)) = \mathrm{True},$$

where $\mathrm{Final}(\pi_T)$ denotes the final state reached by the execution of π_T, and $\mathrm{Goal}(P_T)$ refers to the goal state associated with P_T.

7.1 Motivation

Analogical transfer excels in scenarios where the target environment differs significantly from the source environment yet shares key structural similarities. For example, assembling a table is analogous to a robot setting up a temporary communication network—both involve connecting essential parts to form a stable system. Similarly, the task of stacking blocks in certain scenarios can resemble stacking ingredients to assemble a sandwich. These analogies highlight the potential of adapting plans to fit different contexts—a value that becomes more pronounced as robots are deployed in increasingly diverse environments.

Existing approaches to transferring plans in robotics primarily focus on transferring motions between problems or environments [7, 73, 96, 97, 163], and pay little attention to transferring task plans *across* planning domains. In contrast, our approach facilitates well-defined plan transfer across different domains by leveraging *functorial data migration*. This method enables the automatic generation of analogous plans in new domains eliminating the need for replanning. A schematic of our approach is shown in Figure 7.1. In this figure, dotted boxes represent the data that is automatically generated by the plan transfer process.

In the remainder of this chapter, we provide the necessary background, covering topics such as planning domain ontologies, ontology alignment, and functorial data migration. In Section 7.3, we introduce novel definitions for two specific types of analogies, which are fundamental to our

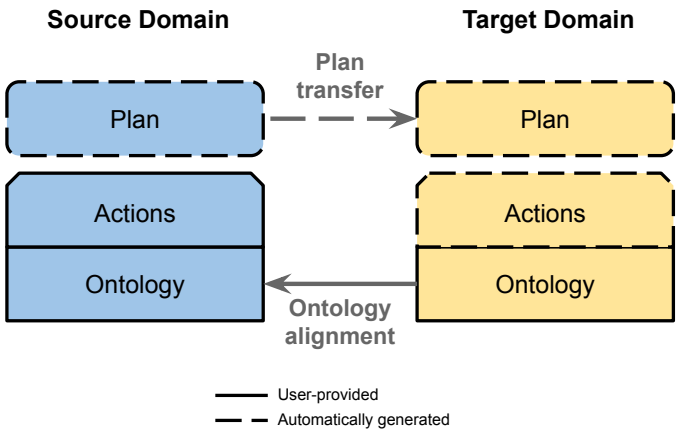


Figure 7.1: A schematic of analogical plan transfer.

approach to analogical plan transfer. We will then delve into our categorical approach to plan transfer in Section 7.4. Section 7.5 introduces key programming constructs from AlgebraicJulia, essential for implementing this approach. We will then examine two case studies: a direct analogy between table assembly and emergency communication planning in Section 7.6, and a structural analogy between Blocksworld and AI2-THOR Kitchenworld in Section 7.7. Finally, in Section 7.9, we will discuss practical considerations, benefits, and limitations of analogical plan transfer.

7.2 Background

In this section, we delve into the foundational concepts that underpin these approaches. Specifically, we will introduce the essential background as it relates to planning domain ontologies, ontology alignment, and functorial data migration.

7.2.1 Planning Domain Ontology

As we have seen in Chapter 4, a planning scenario can be broken down into two key components: the planning domain and the planning problem [23]. A planning domain has two parts: the **domain ontology** and the **domain actions**. The domain ontology specifies **types** and **predicates**

that describe the environment within which planning occurs. The domain actions are formal, predefined descriptions of operations that can be performed within this environment, with both the preconditions and effects of these actions expressed using the language provided by the domain ontology. Together, the domain ontology and domain actions constitute the planning domain. The design of the domain ontology is crucial, as even minor changes can significantly impact the design of the domain actions. For the purposes of analogical plan transfer, we will treat the domain ontology as the medium for establishing an analogical map between two domains.

7.2.2 Ontology Alignment

As introduced in Section 2.4.6, ontology alignment is a process in the field of knowledge representation that involves generating a set of correspondences between entities in different ontologies—these entities can be concepts, properties, or instances [99, 164, 165]. The need for ontology alignment arose from the rapid development of the Internet and the Semantic Web ontologies as they were developed independently to represent diverse, domain-specific knowledge. Since a single, universal ontology that covers all frameworks is impractical, the ability to align ontologies became essential for enabling interoperability. In the context of analogical plan transfer, we plan to use an approach that aligns closely with these ontology alignment techniques. Specifically, we will employ **manual ontology alignment** [165], where human experts manually create correspondences between entities in different ontologies based on their domain knowledge and understanding of the concepts. However, our approach does not preclude the possibility of incorporating automated alignment methods where appropriate.

7.2.3 Functorial Data Migration

Functorial data migration, as conceptualized by Spivak [43], is a mathematical approach to migrating C-set data under a schema category C to data under a different schema category, e.g. C' (For C-set, refer to Definition 22). At its core, this method uses functors to systematically map data from one schema to another such that the relationships within the data are preserved throughout the migration process. Functorial data migration has been applied in several areas, including database integration [166], model-management in systems engineering [167], and knowledge representation [168].

A functorial data migration depends on two components:

1. **A translation functor,** F: A functor F maps objects and morphisms from one schema category to another schema category. Specifically, the functor maps objects from the target schema (with respect to the direction of transfer) to the source schema, $F : \mathsf{C}' \to \mathsf{C}$. The translation functor forces one to systematically relate the target schema to the source schema in a way that preserves the compositional structure of the target schema. This means that for every object in C', F assigns it to a corresponding object in C, and for every morphism in C', F assigns a corresponding morphism in C, while preserving composition found within the target schema.

2. **A migration functor,** Δ_F: Given the translation functor F, the (delta) migration functor, Δ_F, systematically transforms instances of the source schema C into instances of the target schema C'. Specifically, for a given C-set functor, $X : \mathsf{C} \to \mathsf{Set}$, $\Delta_F(X)$ is an object in C'-Set, which is obtained by pre-composing X with F, i.e. $X \circ F : \mathsf{C}' \xrightarrow{F} \mathsf{C} \xrightarrow{X} \mathsf{Set}$. This means that the image for every object in C' under Δ_F is determined by applying F and then X, and similarly for morphisms.

Formally:

Definition 26 (Delta Data Migration)**.** Given a functor $F : \mathsf{C}' \to \mathsf{C}$, called a **translation functor**, a **delta migration functor**, $\Delta_F : \mathsf{C}\text{-Set} \to \mathsf{C}'\text{-Set}$, is constructed as follows:

For a given object X in C-Set,

$\Delta_F(X) = X \circ F$

For a given morphism $\alpha : X \to Y$ in C-Set, Δ_F acts on α to produce the following morphism in C'-Set—

$\Delta_F(\alpha)_{c'} : X(F(c')) \to Y(F(c'))$ for $c' \in \mathsf{C}'$

Notice that $\Delta_F(\alpha)$ is a natural transformation. This means that a component for $\Delta_F(\alpha)$ must be defined for every object in C', hence $\Delta_F(\alpha)_{c'}$.

An important fact about delta data migrations is that they preserve pushouts [43]. Because pushouts are inherent to how new starts are computed in our planning representation language (see Section 6.4.4), migrating actions defined as pushout squares is well-defined.

7.2.3.1 Conjunctive Query Data Migration

A delta data migration depends on the simplest construction of a translation functor. It is also possible to define a translation functor that maps types in the target domain to finite diagrams (Refer to Definition 5 for diagrams.) of types, otherwise known as *conjunctive queries*, in the source domain. This gives an opportunity to express more elaborate structures when assigning types from one domain to another. A data migration with a translation functor that uses conjunctive queries is called a **conjunctive query data migration** [169, 170]. Formally:

Definition 27 (Conjunctive Query Data Migration)**.** Given a translation functor $F : \mathsf{C}' \to \mathsf{Diag}^{\mathsf{op}}(\mathsf{C})$ and a **conjunctive query migration functor**, $\Delta_F : \mathsf{C}\text{-Set} \to \mathsf{C}'\text{-Set}$, is constructed as

follows[a]:

For a given object X in C-Set,

$$\Delta_F(X) = F^*(X) \circ F, \quad \text{where } F^*(X) : \mathsf{Diag}^{\mathsf{op}}(\mathsf{C}) \xrightarrow{\mathsf{Diag}^{\mathsf{op}}(X)} \mathsf{Diag}^{\mathsf{op}}(\mathsf{Set}) \xrightarrow{\lim} \mathsf{Set}$$

Here, $F^*(X)$ is the limit of the diagram computed in the category Set.

For a given morphism $\alpha : X \to Y$ in C-Set, Δ_F acts on α to produce the following morphism in C'-Set:

$$\Delta_F(\alpha)_{c'} : F^*(X)(c') \to F^*(Y)(c') \text{ for } c' \in \mathsf{C}'$$

[a] $\mathsf{Diag}^{\mathsf{op}}(\mathsf{C})$ is a category whose objects are diagrams and whose morphisms are natural transformations between these diagrams, with the direction of the natural transformations reversed. This reversal corresponds to taking the opposite category, hence the "op" [103].

A limitation to the use of conjunctive query data migrations is that functoriality of pushouts is not guaranteed. This primarily affects cases where multivariate equations are used to determine attribute values within the source domain. Practical implications of this limitation remains an area of future exploration.

7.3 Types of Analogies and Case Studies

The two varieties of functorial data migrations, delta and conjunctive query migrations, allow us to establish two types of analogies for plan transfer as part of this work. In the field of cognitive psychology, analogies are often categorized by surface similarities—pertaining to superficial features—and structural similarities, which involve deeper, relational correspondences [171]. Inspired by this distinction, we introduce two types of analogies central to our approach: **direct analogies**, which align concepts one-to-one, and **structural analogies**, which align concepts with patterns of concepts. An illustration of this distinction can be seen in Figure 7.2. In the following sections, we elaborate on each type of analogy and motivate the use of each type by introducing case studies that demonstrate their effectiveness.

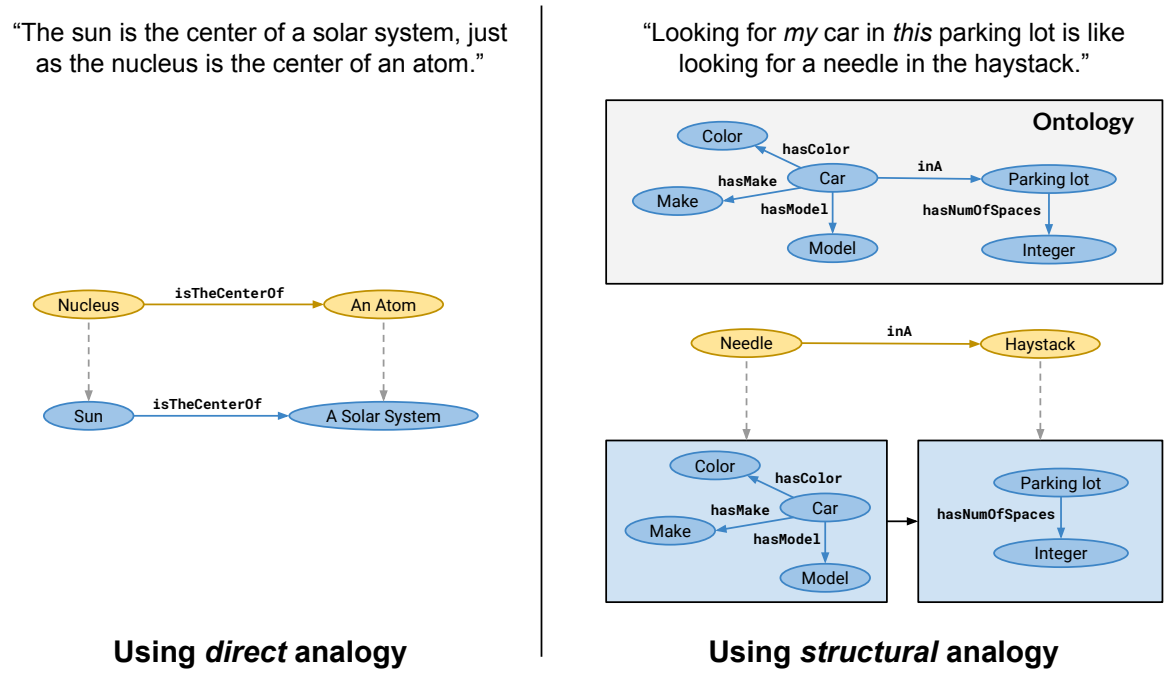


Figure 7.2: Examples of different types of analogies–direct and structural.

7.3.1 Direct Analogy

Direct analogies involve a straightforward, one-to-one correspondence between concepts in different domains, facilitating the direct transfer of solutions across domains with similar structural elements. For instance, consider the analogy: *"The sun is the center of the solar system, just as the nucleus is the center of an atom."* Here, the parallelism highlights a clear alignment between the sun and the nucleus, and between the solar system and the atom. This is a direct analogy because each concept in the topic of the solar system is matched one-to-one with a concept in the topic of the atom.

Case Study for Direct Analogy. In this chapter, we apply a direct analogy to transfer a plan for assembling a table to a plan designed for setting up an emergency communication network during disaster relief efforts. In both cases, the core challenge is managing the placement and connection of multiple components to create a functional system. For the table assembly, this involves positioning

and attaching legs, screws, and a tabletop to build a stable structure. Similarly, setting up an emergency communication network requires deploying and connecting communication nodes, relay stations, and a central hub to establish reliable communication across a disaster area. Table assembly highlights the potential for robots to construct physical structures at scale [172, 173], while network recovery is crucial for search and rescue operations, where robots could be deployed to establish communications in hazardous environments [174]. This analogy is compelling because it challenges conventional boundaries, showing how a seemingly simple task like table assembly can inform critical processes in disaster management.

7.3.2 Structural Analogy

Structural analogies differ from direct analogies by mapping concepts from one domain to patterns of concepts in another. This enables the transfer of plans across domains that, while not identical, share deeper relational structures. Unlike direct analogies, which involve a one-to-one mapping of individual elements, structural analogies identify and align broader patterns of similarity. For example, consider the analogy: *"Looking for my car in this parking lot is like looking for a needle in a haystack."* Though subtle, the comparison emphasizes the complexity of locating a small, specific object (a car, defined by its make, model, and color) within a larger, cluttered space (a parking lot with a certain number of spaces). These deeper patterns—based on the car's characteristics and the parking lot's layout—are aligned with the concepts of the needle and the haystack.

Case Study for Structural Analogy. In this chapter, we explore how a plan for stacking blocks can be adapted to a plan for assembling a sandwich through a structural analogy. This analogy demonstrates the transfer of robot task plans from a simplified domain like Blocksworld to a more complex and expressive environment such as Kitchenworld, inspired by the AI2-THOR

simulation environment [38]. While Blocksworld offers generality, Kitchenworld provides the detailed realism needed for practical applications. However, planning in descriptive environments, like Kitchenworld, can be computationally intensive—making it advantageous to first create plans in an abstract domain and then translate them to a more detailed one. This strategy for using simple, abstract problems to solve complex real-world problems highlights the value structural analogies can provide.

In this section, we have examined both direct and structural analogies, introducing how they can be leveraged to transfer task plans between different domains. By exploring these types of analogies, we aim to demonstrate their broad applicability and the potential they have for streamlining planning processes across diverse and complex scenarios.

7.4 Categorical Approach to Plan Transfer

Building on the foundational concepts outlined in the previous section and the planning representation language introduced in Chapter 6, we introduce the categorical approach to analogical plan transfer. In this section, we will first introduce the high-level steps of analogical plan transfer and then elaborate on its parts.

High-level Steps in Plan Transfer. Here, we list the practical steps required to perform analogical plan transfer using functorial data migrations.

1. **Define the Source Domain**: Start by defining the ontology, D_S, and actions, A_S, for the source planning domain.
2. **Define the Planning Problem**: Establish the initial and goal states, $P_S = (I, G)$, within the source planning domain.
3. **Plan**: Solve for a valid plan, π_S, that satisfies P_S within the source domain.

4. **Define the Target Domain Ontology**: Specify the target ontology, D_T, in the target planning domain.
5. **Align Ontologies**: Specify a translation functor, $F : \mathsf{D}_T \to \mathsf{D}_S$, linking the types and predicates from the target domain ontology to the source domain ontology.
6. **Transfer the Plan**: Use a functorial data migration, Δ_F, to transfer the grounded plan, resulting in π_T.
7. **Validate the Plan**: Check that the end state of π_T contains $\Delta_F(G)$.

In the following sections, we will elaborate on how to construct an ontology alignment map and discuss the effect of functorial data migrations on plan actions.

7.4.1 Ontology Alignment Map, $F : \mathsf{D}_T \to \mathsf{D}_S$

Ontology alignment is achieved by defining a translation functor. The translation functor, $F : \mathsf{D}_T \to \mathsf{D}_S$, is a map from the target domain to the source domain, ensuring that types and predicates are correctly translated. The nature of the translation functor dictates the type of analogy it forms. In the case of delta data migration, the translation functor gives rise to a direct analogy by mapping objects and morphisms from one schema to another directly. Conversely, the translation functor in a conjunctive query data migration gives rise to a structural analogy, as it maps objects to diagrams of objects and morphisms in the schema category.

7.4.1.1 For a Direct Analogy

In the context of a direct analogy, the translation functor, F, operates as a straightforward mapping from the target domain to the source domain. The simplicity of this mapping allows for a one-to-one correspondence between types, predicates, and attributes in the two domains. To ensure the integrity of this direct analogy, the following guidelines must be followed:

1. **Map each type to a type**. Each type in the target domain should map to a single type in the source domain.
2. **Map each predicate to a predicate**. Each predicate in the target domain should map to a corresponding predicate or composite of predicates in the source domain while preserving compositional relationships.
3. **Map each attribute to a derived value**: Each attribute in the target domain must map to a function determining the non-combinatorial value, e.g. a value of type string, floating point, Boolean, or integer.

7.4.1.2 For a Structural Analogy

While the direct analogy offers a simple and effective means of plan transfer, it may not be sufficient for more complex domains where relationships and dependencies between types are intricate and interdependent. In such cases, a structural analogy, facilitated by a conjunctive query data migration, is more appropriate. The following guidelines apply:

1. **Map each type to a diagram**. Instead of mapping each type in the target domain directly to a single type in the source domain, map it to a finite diagram (conjunctive query) in the source domain, representing a more complex structure or relationship between multiple types. The diagram can contain a single type, if appropriate.
2. **Map each predicate to a morphism between diagrams**. Each predicate in the target domain should map to a morphism between diagrams or composite of morphisms between diagrams in the source domain while preserving the relationships between diagrams of types.
3. **Map each attribute to a derived value**. Similar to a direct analogy, each attribute in the target domain must map to a function determining the non-combinatorial value, e.g. a value of type string, floating point, Boolean, or integer.

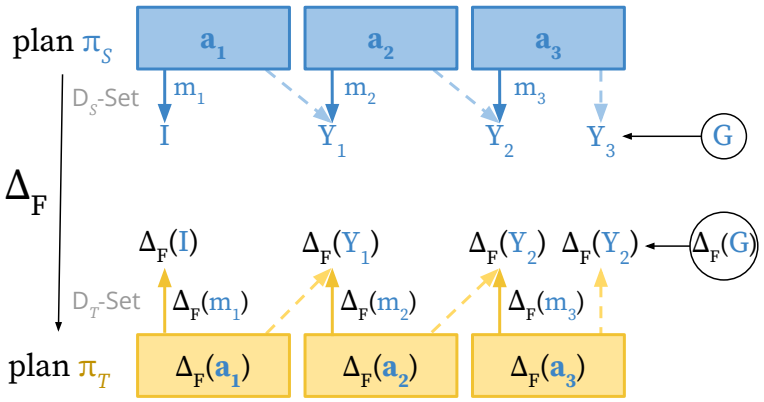


Figure 7.3: Schematic of analogical plan transfer applied to plan π_S.

The choice between a direct or structural analogy should be guided by the complexity of the domains involved and the nature of the relationships that need to be preserved during plan transfer.

7.4.2 Plan Transfer Migration Functor, $\Delta_F : \mathsf{D}_S\text{-Set} \to \mathsf{D}_T\text{-Set}$

Once a translation functor, $F : \mathsf{D}_T \to \mathsf{D}_S$, is defined, we can evaluate the plan transfer functor, $\Delta_F : \mathsf{D}_S\text{-Set} \to \mathsf{D}_T\text{-Set}$. Recall from Chapter 6, the objects of $\mathsf{D}_S\text{-Set}$ can be interpreted as states in the source domain. Morphisms in $\mathsf{D}_S\text{-Set}$ are used to define (ungrounded) action spans. Throughout the planning process, match morphisms, m_i, ground actions. After an action is grounded, the new state is computed using DPO rewriting. A sequence of action spans and their match morphisms form the plan that is to be transferred.

To illustrate the process of plan transfer, Algorithm 3 provides an algorithmic view, and Figure 7.3 depicts this process graphically. In Figure 7.3, the top (blue) sequence represents a plan in $\mathsf{D}_S\text{-Set}$. The bottom (yellow) sequence represents a plan in $\mathsf{D}_T\text{-Set}$. Rectangles in the schematic correspond to actions within each plan. Arrows emerging from the left side of the rectangles are match morphisms. Dotted arrows emerging from the right side of the rectangles point to the resultant state after each action is applied. Each morphism in the action span and the action's

Algorithm 3: Plan Transfer Procedure

Input:
Ontology map, $F : \mathsf{D}_T \to \mathsf{D}_S$
Source plan, π_S
Goal, G

Output: Transferred plan π_T

```
Function TransferPlan(F, π_S, G):
    π_T = [ ];
    // Transfer plan
    foreach (action a, match m) in π_S do
        Δ_F ← construct plan transfer functor using F;
        a' ← Δ_F(a);
        m' ← Δ_F(m);
        put (a', m') in π_T;
    end
    Y ← DPO(π_T[end]) // Get final state of the plan
    if assert Δ_F(G) in Y // Check that goal is satisfied
     then
        return π_T;
    end
    else
        return "Invalid Plan" ;
    end
```

match morphism is transferred using a functorial data migration. The result is new states and actions that respect the structure imposed by the schema category of the target domain.

7.4.2.1 Validation

A transferred plan is considered valid if the transferred goal is satisfied by the final state of the transferred plan. When plan transfer is performed via the delta data migration Δ_F, the resulting plan is guaranteed to be valid in the target domain because Δ_F preserve pushouts, as established in Theoreom 7.4.1.

Theorem 7.4.1 (Universally Valid Analogical Transfer via Delta Data Migrations)**.** Let D_S and D_T be schema categories representing ontologies of the source and target planning domains, respectively. Let $\Delta_F : \mathsf{D}_S\text{-}\mathsf{Set} \to \mathsf{D}_T\text{-}\mathsf{Set}$ be a delta data migration functor. For any source

plan π_S achieving a goal state G, the transferred plan $\Delta_F(\pi_S)$ achieves the transferred goal $\Delta_F(G)$ in the target domain.

Proof. First, consider the final ground action in the source plan π_S. The grounding morphism of the final action, $m_n : \text{Pre} \to Y_{(n-1)}$, consists of the object, Pre, of the final action, and object $Y_{(n-1)}$, where n is the total number of actions in the plan and $\Delta_F(Y_{(n-1)})$ is the second-to-last state of the plan. Transferring this morphism transfers its domain and codomain objects with it, which produces the morphism:

$$\Delta_F(m_n) : \Delta_F(\text{Pre}) \to \Delta_F(Y_{(n-1)})$$

Computing the DPO for this grounded action yields the final state in the target domain, which we denote by Y'_n.

To verify that the goal has been satisfied in the target domain, it is necessary to check whether there exists a monomorphism in the target domain, namely:

$$g' : \Delta_F(G) \hookrightarrow Y'_n,$$

where $\Delta_F(G)$ is the goal state in the target domain. Given that the source plan satisfied the source goal, there must exist a monomorphism in the source domain:

$$g : G \hookrightarrow Y_n.$$

Because delta data migrations preserve pushouts [43], we can conclude that:

$$\Delta_F(Y_n) = Y'_n.$$

By functoriality, the image of the monomorphism $g : G \hookrightarrow Y_n$ under Δ_F is:

$$\tilde{g} : \Delta_F(G) \hookrightarrow \Delta_F(Y_n).$$

Since $\Delta_F(Y_n) = Y'_n$, it follows that:

$$\tilde{g} : \Delta_F(G) \hookrightarrow Y'_n.$$

Thus, we have shown that $\tilde{g} = g'$ implying that there exists a monomorphism from the transferred goal $\Delta_F(G)$ to the final state Y'_n in the target domain. □

In contrast, conjunctive query migrations do not necessarily guarantee that pushouts are preserved which means that the transferred plan is not guaranteed to satisfy the analogous goal. As a byproduct, structural analogies require additional validation or adjustments to ensure the plans validity in the target domain. This is done by evaluating whether there is a monomorphism from $\Delta_F(G)$ into the final state of π_T. This can be done by using a homomorphism search algorithm [42], restricting matches to morphisms that are monic.

Having established the foundations of plan transfer for both delta and conjunctive query migrations, we now turn to a concrete examples to illustrate their practical application.

7.5 Software Constructors found within AlgebraicJulia

In the implementation of the domain ontologies for this dissertation, we utilize constructors found within AlgebraicJulia [154]. These Julia-based packages are designed for category-theoretic modeling and offer several useful tools for defining domain ontologies and actions within planning domains. Its constructors allow us to express schema categories, functors, and data migrations concisely, making it easy for readers to comprehend complex examples with greater efficiency. While these constructs are crucial to our implementation, they are part of an existing software tool and are not original contributions of this thesis. Nonetheless, understanding their function is essential for following the implementations discussed in subsequent sections.

Two constructs from these packages, `Hom` and `Attr`, are particularly important for formalizing the domain ontologies used in this work. Additionally, syntactic sugar constructors for defining translation functors and objects in C-Set are also described in this section.

The `Hom` Construct. The `Hom` construct, short for "homomorphism," is a fundamental concept in category theory, where it typically denotes a structure-preserving map between two objects. In

the context of this work, `Hom` is used to define predicates between different types within a domain ontology. These predicates are directional, where the first argument refers to the source of the predicate and the second argument refers to the target. For example, in the ColorBlocksworld domain (which we will define in the next section), the predicate stating that "A Gripper is holding a Block" is expressed as:

```
isHolding::Hom(Gripper,Block)
```

This specification indicates that a `Gripper` type can hold something of type `Block`. This predicate can be incorporated in domain ontologies where a robot gripper might manipulate blocks.

The `Attr` and `AttrType` Construct. The `Attr` construct is used to define attributes associated with types, namely `AttrType`, in the domain ontology. Attributes, as in attributed C-sets (see Definition 21 in Section 6.4.1.2), allow a non-combinatorial data type to be specified. Some examples include item weight and color taking values of type `Float64` or `String`. For instance, to describe the color of a block in the ColorBlocksworld domain, the `Attr` construct is used:

```
Color::AttrType
hasColor::Attr(Block, Color)
```

This statement specifies that each `Block` type has an attribute `hasColor` which specifies an attribute of type `Color`. Attribute types are mapped to non-combinatorial data types at the time of construction such as `String`, `Float64`, etc. In this case, if `Color` maps to `String`, it can take on values like `"red"` and `"blue"`.

The :: Operator. The :: operator is used to assign an object or type in a schema to a specific set as per the C-Set theory. The right side of the :: operator specifies the object or type within the schema category C, while the left side specifies the set that is the image under the C-set functor. For example, in the expression `gripper::Gripper`, the type Gripper is assigned to the set {gripper} and in (block1,block2)::Block, the type Block is assigned to the set $\{\text{block}_1, \text{block}_2\}$.

The `=>` Operator. The `=>` operator is used to represent a functor assignment. Specifically, it captures the mapping of an object or morphism in one category to a corresponding object or morphism in another category. This operator is a crucial part of defining how entities from the source domain (represented as a category) correspond to entities in the target domain (another category). For example, when we write `Part => Block`, we are specifying that the object Part in the source category is mapped to the object Block in the target category. Similarly, the expression `isPartOf => isOn` for predicates `isPartOf::Hom(Part, Assembly)` in the Assembly domain and `isOn::Hom(Block, Block)` in the ColorBlocksworld domain specifies that the predicate `isPartOf` is mapped to the predicate `isOn`[1].

A `(f(x) == a)` expression. An expression, `f(x) == a`, specifies how a C-set functor maps a morphism f in the schema category to a corresponding function in the category Set. Specifically, $f(x)$ represents the application of the function to an element x in the set corresponding to the domain object of $f \in \mathrm{Hom}(\mathsf{C})$. The result, a, is the element in the set corresponding to the codomain object of f. For example, in the expression `IsPartOf(part_1) == table_1`, the function under the (IsPartOf : Part $\to$ Assembly) morphism maps $\mathrm{part}_1 \mapsto \mathrm{table}_1$, which are elements of the sets underlying Part and Assembly, respectively.

Macro programs, such as `@acset_colim` for computing the colimit of representable functors, `@migration` for defining data migrations, and `@join` for defining conjunctive queries have also been implemented in AlgebraicJulia. For those interested in the implementation of these constructors, we refer the reader to the AlgebraicJulia GitHub repository[2].

[1] Notice that this would only be a valid functor assignment if `Assembly => Block`.
[2] `https://github.com/AlgebraicJulia`

7.6 A Case Study of Direct Analogy: From Assembly to Emergency Communications

To explore the concept of direct analogy in a practical context, we will examine the transfer of plans between two distinct yet structurally identical domains. Specifically, we focus on an **Assembly** domain, where the task involves constructing an assembly from various parts, and an **EmergencyComms** domain, where the task is to establish an emergency communication network using various network components. In the Assembly domain, each part has a specific location, type, and relationship with other parts. In the EmergencyComms domain, the network components also have specific locations, types, and relationships with other components, akin to how parts relate within an assembly. The direct analogy between these domains lies in the structural correspondence between attaching parts in an assembly and connecting components in a communication network. Both domains involve constructing a larger system from individual components, where each component has a defined role and relationship within the broader structure. In this section, we will formalize this direct analogy and demonstrate how to transfer plans between these domains using the principles of functorial data migration. An overview of the plan transfer scenario is shown in Figure 7.4.

7.6.1 Domains

Planning domain ontologies are user-defined, therefore, we begin by identifying important ontological features for each domain. Both the Assembly and EmergencyComms domains must be able to express predicates that capture the relationships between different components and their roles within larger systems. In addition, these domains should specify the location of components and assemblies. These concepts can be captured using schema categories, denoted as $\mathsf{Assembly}$ for the Assembly domain and $\mathsf{EmergencyComms}$ for the EmergencyComms domain. The

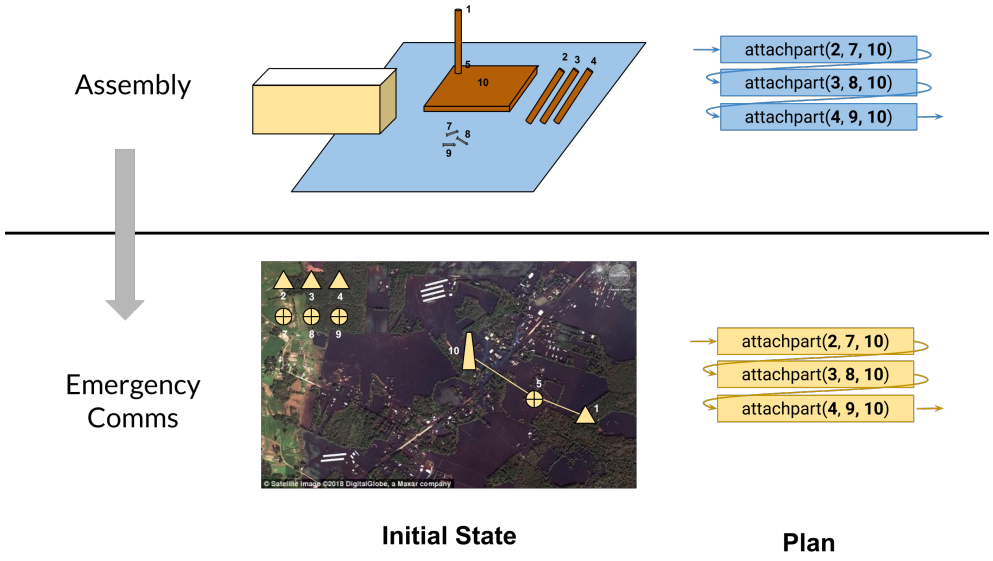


Figure 7.4: Overview of the direct analogy case study.

data migration functor transfers a plan from the source domain, Assembly, to the target domain, EmergencyComms.

7.6.1.1 Assembly Domain Ontology

Recall the Assembly domain introduced in Chapter 6 (see Section 6.5). This schema category contains types:

```
Part, Assembly, Location, PartType
```

These types describe the fundamental entities in the Assembly domain, where `Part` refers to individual components, `Assembly` represents the larger structure formed by these parts, `Location` denotes the spatial context, and `PartType` classifies the different kinds of parts.

In this schema category, we include predicates such as:

```
IsPartOf::Hom(Part, Assembly)
PIsLocatedAt::Hom(Part, Location)
```

which express that a `Part` is part of an `Assembly` and that a `Part` is located at a specific `Location`. The type of a part is expressed using:

```
HasType::Hom(Part, PartType)
```

where example part types can be screw, leg, and tabletop for a table assembly.

The schema also includes relations that express connections between parts, such as:

```
IsAttachedTo::Ob
IsAttachedToL::Hom(IsAttachedTo, Part)
IsAttachedToR::Hom(IsAttachedTo, Part)
```

where `IsAttachedTo` represents a relationship between parts, with `IsAttachedToL` and `IsAttachedToR` mapping the relationship to the respective parts involved. A full specification of the Assembly schema can be found in Appendix A.2.1. This full specification of the schema is important as it provides the foundation for understanding how the ontology is structured and how its structure supports the analogical plan transfer processes discussed in later sections.

7.6.1.2 EmergencyComms Domain Ontology

The EmergencyComms domain differs from Assembly in its types and predicates, reflecting the distinct nature of emergency communication systems as opposed to physical assembly processes. This schema category models components of communication networks, their connections, and the areas they serve. In EmergencyComms, key types include:

```
NetworkComponent, CommunicationNetwork, Area, ComponentType
```

These types describe the core entities in the EmergencyComms domain, where `NetworkComponent` refers to individual network elements (e.g., communication nodes, relay nodes), `CommunicationNetwork` represents the entire communication system, `Area` denotes the disaster region served, and `ComponentType` classifies the different kinds of network components.

Similar to Assembly, this domain also includes predicates that describe how components relate to the overall network and the areas they serve, such as:

```
IsPartOf::Hom(NetworkComponent, CommunicationNetwork)
ComponentIsServingArea::Hom(NetworkComponent, Area)
```

which express that a `NetworkComponent` is part of a `CommunicationNetwork` and that it serves a specific `Area`.

The schema also includes relations that describe connections between network components:

```
IsConnectedTo::Ob
IsConnectedToL::Hom(IsConnectedTo, NetworkComponent)
IsConnectedToR::Hom(IsConnectedTo, NetworkComponent)
```

where `IsConnectedTo` represents a connection between network components, with `IsConnectedToL` and `IsConnectedToR` mapping the connection to the respective components involved.

Similar to Assembly, EmergencyComms includes a predicate to describe the type of components, such as:

```
HasType::Hom(NetworkComponent, ComponentType)
```

which specifies the type of the network component, such as a communication node, relay node, and central hub.

EmergencyComms shares some structural features with Assembly, maintaining a sense of analogy between the two domains. For example, both domains involve components (whether parts or network components) that are part of a larger system and are located in specific areas. This shared structure allows for a direct analogy to be formed between these domains, as will be explored in subsequent sections. A full specification of the EmergencyComms schema can be found in Appendix A.2.2.

7.6.2 Ontology Alignment

To explore the concept of direct analogy, we examine the translation of plans between the Assembly and EmergencyComms through ontology alignment. Despite the apparent differences in these domains, their underlying structure allows for a meaningful analogy, which we can capture using a translation functor, F : EmergencyComms $\to$ Assembly. This functor maps types and

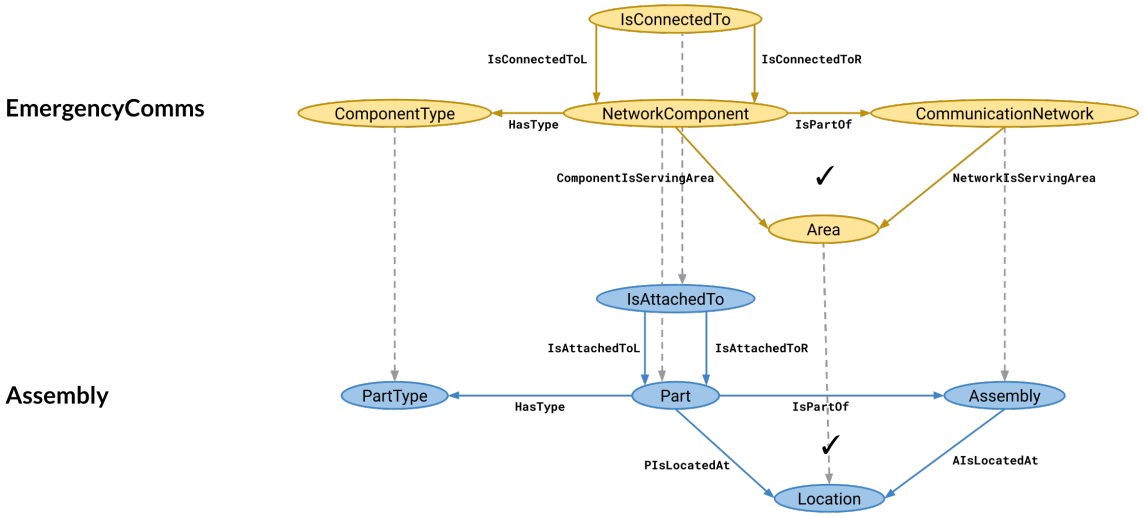


Figure 7.5: Ontology alignment using a direct analogy.

predicates from the target domain EmergencyComms to corresponding types and predicates in the source domain Assembly while preserving the composition of relationships.

We start by mapping types that share similar semantics in both domains. For example:

```
% types
NetworkComponent => Part
CommunicationNetwork => Assembly
Area => Location
ComponentType => PartType
```

Here, a `NetworkComponent` in the EmergencyComms domain is analogous to a `Part` in the Assembly domain, both representing fundamental units in their respective systems. Similarly, a `CommunicationNetwork` corresponds to an `Assembly`, as both represent a larger structure composed of these fundamental units. The `Area` and `Location` types map naturally, reflecting the geographic or spatial aspects in each domain, while `ComponentType` and `PartType` denote the classifications of these units.

Given this map between types, the predicates between types become obvious. This involves mapping predicates that describe the function, membership, and location of these components:

```
IsPartOf => IsPartOf
ComponentIsServingArea => PIsLocatedAt
```

```
HasType => HasType
NetworkIsServingArea => AIsLocatedAt
```

In this case, the `IsPartOf` relationship in both domains reflects the inclusion of a part or network component within a larger assembly or communication network. The `ComponentIsServingArea` predicate, which specifies the area a network component serves, maps to `PIsLocatedAt`, reflecting the location of a part within an assembly. The `HasType` predicate, which assigns a type to a network component or part, remains consistent across both domains. Lastly, `NetworkIsServingArea` in EmergencyComms maps to `AIsLocatedAt` in Assembly, preserving the relationship between the entire structure and its location or service area.

Finally, we address the relationships between components or parts in either domain, particularly how components or parts are connected or attached. The translation functor F maps relationships in the EmergencyComms domain to analogous relationships in the Assembly domain:

```
IsConnectedTo => IsAttachedTo
IsConnectedToL => IsAttachedToL
IsConnectedToR => IsAttachedToR
```

In this mapping, the `IsConnectedTo` relationship between network components is directly analogous to the `IsAttachedTo` relationship between parts.

The left and right connections (`IsConnectedToL` and `IsConnectedToR`) map directly to their counterparts (`IsAttachedToL` and `IsAttachedToR`), preserving the directional nature of these connections.

7.6.3 Plan Transfer

Having defined an ontology alignment map from EmergencyComms to Assembly, we can examine the results of applying a delta migration functor to a plan within the Assembly domain. In this subsection, we will first define the specific problem within the Assembly domain, including the initial and goal states. Next, we will describe how the actions in the Assembly domain were

Table 7.1: Table comparing Assembly-Set and EmergencyComms-Set

Assembly	EmergencyComms
`# Initial state` `(part_1, part_2, ... part6, ..., part_10)::Part` `(table_1, table_2)::Assembly` `(workspace, storageshelf)::Location` `(leg, screw, tabletop)::PartType` `(x1, x2, x3)::IsAttachedTo` `IsAttachedToL(x1) == part_6;` `IsAttachedToR(x1) == part_10;` `IsAttachedToL(x2) == part_6;` `IsAttachedToR(x2) == part_1;` `IsAttachedToL(x3) == part_1;` `IsAttachedToR(x3) == part_10;` `AIsLocatedAt(table_1) == workspace;` `AIsLocatedAt(table_2) == storageshelf;` `HasType(part_1) == leg;  # part_1` `IsPartOf(part_1) == table_1;` `PIsLocatedAt(part_1) == workspace;` `HasType(part_2) == leg;  # part_2` `IsPartOf(part_2) == table_1;` `PIsLocatedAt(part_2) == workspace;` `...`	`# Initial state (after transfer)` `(part_1, part_2, ... part_6,` `...., part_10)::NetworkComponent` `(table_1, table_2)::CommunicationNetwork` `(workspace, storageshelf)::Area` `(leg, screw, tabletop)::ComponentType` `(x1, x2, x3)::IsConnectedTo` `IsConnectedToL(x1) == part_6;` `IsConnectedToR(x1) == part_10;` `IsConnectedToL(x2) == part_6;` `IsConnectedToR(x2) == part_1;` `IsConnectedToL(x3) == part_1;` `IsConnectedToR(x3) == part_10;` `NetworkIsServingArea(table_1) == workspace;` `ComponentIsServingArea(table_2) == storageshelf;` `HasType(part1) == leg;` `IsPartOf(part1) == table_1;` `ComponentIsServingArea(part_1) == workspace;` `HasType(part2) == leg;` `IsPartOf(part2) == table_1;` `ComponentIsServingArea(part_2) == workspace;` `...`
`# AttachPart precondition` `(part_x, part_y, part_z)::Part` `(leg, screw, tabletop)::PartType` `assembly_x::Assembly` `IsPartOf(part_x) == assembly_x` `IsPartOf(part_y) == assembly_x` `IsPartOf(part_z) == assembly_x` `HasType(part_x) == leg` `HasType(part_y) == screw` `HasType(part_z) == tabletop`	`# AttachPart precondition (after transfer)` `(part_x, part_y, part_z)::NetworkComponent` `(leg, screw, tabletop)::ComponentType` `assembly_x::CommunicationNetwork` `IsPartOf(part_x) == assembly_x` `IsPartOf(part_y) == assembly_x` `IsPartOf(part_z) == assembly_x` `HasType(part_x) == leg` `HasType(part_y) == screw` `HasType(part_z) == tabletop`
`# Match morphism` `part_x::Part -> part_1::Part` `part_y::Part -> part_6::Part` `part_z::Part -> part_10::Part` `leg::PartType -> leg::PartType` `screw::PartType -> screw::PartType` `tabletop::PartType -> tabletop::PartType` `assembly_x::Assembly -> table_1::Assembly`	`# Match morphism (after transfer)` `part_x::NetworkComponent` `-> part_1::NetworkComponent` `part_y::NetworkComponent` `-> part_6::NetworkComponent` `part_z::NetworkComponent` `-> part_10::NetworkComponent` `leg::ComponentType -> leg::ComponentType` `screw::ComponentType -> screw::ComponentType` `tabletop::ComponentType` `-> tabletop::ComponentType` `assembly_x::CommunicationNetwork` `-> table_1::CommunicationNetwork`

specified. Finally, we will demonstrate the effects of transferring a plan from the Assembly domain to the EmergencyComms domain.

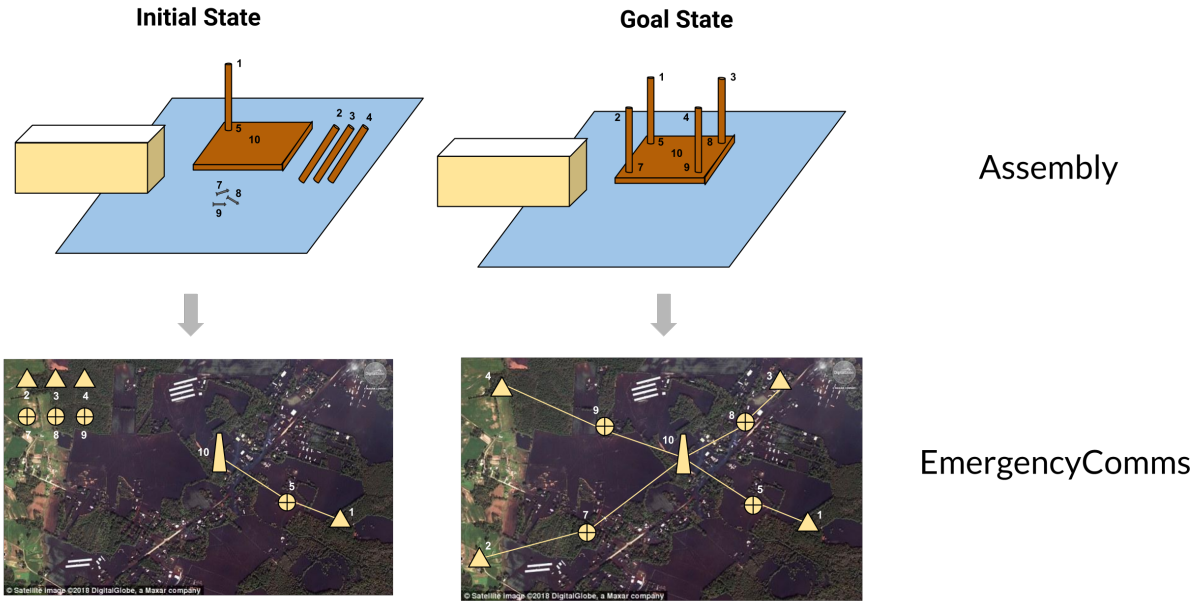


Figure 7.6: The initial and goal states in the Assembly and EmergencyComms domains.

7.6.3.1 Assembly Problem

Consider a problem in the Assembly domain where parts of a table are initially arranged in a workspace, as shown in Figure 7.6. The assembly consists of four legs (`part_1`, `part_2`, `part_3`, `part_4`), a tabletop (`part_10`), and four screws (`part_6`, `part_7`, `part_8`, `part_9`). The initial state indicates that one leg (`part_1`) is already attached to the tabletop (`part_10`) using a screw (`part_6`). The remaining legs and screws are positioned in the workspace but not yet attached to the tabletop. The goal state for this problem is to have all four legs (`part_1`, `part_2`, `part_3`, `part_4`) securely attached to the tabletop (`part_10`) using the four screws (`part_6`, `part_7`, `part_8`, `part_9`). This final configuration ensures that the table is fully assembled and stable. The initial and goal states can be defined as objects in Assembly-Set by mapping each type in Assembly to an element in Set. An example of the initial state specification is provided in Table 7.2. The goal state is defined using the same principles.

7.6.3.2 Assembly Domain Action

In the Assembly domain, there is one action called AttachPart, which attaches the leg onto the table using a screw that is attached to both the leg and the table. Refer to Section 6.5.3 for a detailed description of this action. Using our representation language and the syntax from AlgebraicJulia, the `attachpart` action can be expressed using the following action span:

```
Pre = Keep = @acset_colim yAssembly begin
  (part_x, part_y, part_z)::Part
  (leg, screw, tabletop)::PartType
  assembly_x::Assembly

  IsPartOf(part_x) == assembly_x
  IsPartOf(part_y) == assembly_x
  IsPartOf(part_z) == assembly_x

  HasType(part_x) == leg
  HasType(part_y) == screw
  HasType(part_z) == tabletop
end

N = Eff = @acset_colim yAssembly begin
  (part_x, part_y, part_z)::Part
  (leg, screw, tabletop)::PartType
  assembly_x::Assembly

  IsPartOf(part_x) == assembly_x
  IsPartOf(part_y) == assembly_x
  IsPartOf(part_z) == assembly_x

  HasType(part_x) == leg
  HasType(part_y) == screw
  HasType(part_z) == tabletop

  # Attachment relationships
  (x1,x2,x3)::IsAttachedTo
  (part_x, part_y, part_z)::Part
  IsAttachedToL(x1) == part_x; IsAttachedToR(x1) == part_y;
  IsAttachedToL(x2) == part_x; IsAttachedToR(x2) == part_z;
  IsAttachedToL(x3) == part_y; IsAttachedToR(x3) == part_z;
end

l = # self-identification from Keep => Pre, e.g. part_x -> part_x
r = # self-identification from Keep => Eff, e.g. part_x -> part_x
n = r

attachPart = Rule{:DPO}(l, r; ac=[NAC(n)])
```

In this example, the `Pre` condition defines the initial state of the assembly process, where three parts—identified as `part_x`, `part_y`, and `part_z`—are specified. These parts correspond to

a leg, a screw, and a tabletop, respectively. The `Pre` condition also indicates that each part is associated with an assembly, as captured by the `IsPartOf` predicate, and each part has a specific type, such as `leg`, `screw`, or `tabletop`, as indicated by the `HasType` predicate. The negative application condition (NAC), `N`, includes the attachment relationships that appear in `Eff`.

The `Keep` section identifies the entities that remain unchanged during the action. In this case, the entities include the parts and their types, ensuring that the leg, screw, and tabletop maintain their respective roles and relationships within the assembly.

The `Eff` condition describes the result of the action, where the `IsAttachedTo` predicates are used to express that the leg (`part_x`), screw (`part_y`), and tabletop (`part_z`) are now attached to each other. Specifically, the leg is attached to the screw, and both the leg and the screw are attached to the tabletop. The action, therefore, modifies the relationships between these parts by establishing the necessary connections to complete the assembly.

The maps $l : \text{Keep} \hookrightarrow \text{Pre}$, $r : \text{Keep} \hookrightarrow \text{Eff}$, and $n : \text{Pre} \hookrightarrow N$ serve as connections between entities that remain unchanged during the action. These maps make a self-identification, meaning that each element in the `Keep` set directly corresponds to the semantically identical element in both `Pre` and `Eff`, for example.

7.6.3.3 Transfer from Assembly to EmergencyComms

To transfer a plan, we must transfer every action and its match. Consider a plan where the first action is to attach a leg to the tabletop in the Assembly domain. A valid match for the `attachpart` action grounds `part_x` $\mapsto$ `part_2`, `part_y` $\mapsto$ `part_7`, and `part_z` $\mapsto$ `part_10`. The precondition and match in the Assembly domain can be seen in Table 7.2. The plan transfer functor, Δ_F, applies to the initial state, `attachpart(leg_1, screw_1, tabletop)`, and its match by sending X to $X \circ F$, which maps the types in Assembly through F and then to corresponding

sets in Set.

Table 7.2 shows the transferred initial state from the Assembly domain to the EmergencyComms domain. Although the names of instances (e.g., `leg_1`, `screw_1`, `tabletop`) remain the same, they are now instances of types in the EmergencyComms domain. The transferred precondition of the `attachpart` action is also displayed in Table 7.2. The `Keep` and `Eff` components undergo similar transfers. The result is that the types have been translated from the Assembly domain to the EmergencyComms domain, with the semantics of the predicates preserved.

After executing a forward backtracking search algorithm, a valid plan, π_{Assembly} for this problem in the Assembly is:

```
attachpart(part_2, part_7, part_10);
attachpart(part_3, part_8, part_10);
attachpart(part_4, part_9, part_10)
```

The same plan, when transferred to the EmergencyComms domain, $\pi_{\mathsf{EmergencyComms}}$ is:

```
attachpart(component_2, component_7, component_10);
attachpart(component_3, component_8, component_10);
attachpart(component_4, component_9, component_10)
```

The names of the instances are simply pointers to elements in a set and can, therefore, take on different semantic meaning without invalidating the plan. For this exposition, we have chosen to present the plan with instances relabeled to more contextually appropriate terms like `component_2` or `component_10` to indicate this plan is referring to network components.

7.6.4 Validation

To verify that the transferred plan, $\pi_{\mathsf{EmergencyComms}}$, satisfies the transferred goal in the target domain, we must check that the transferred goal is contained in the final state of $\pi_{\mathsf{EmergencyComms}}$. Because a delta data migration was used for this case study, it is guaranteed to be a valid plan per Theorem 7.4.1.

7.6.5 Summary

In this case study, we demonstrated the practical value of using direct analogy to solve a problem in the EmergencyComms domain by leveraging an existing plan from the Assembly domain. By establishing a direct one-to-one mapping between types, predicates, and relationships in both domains, we were able to faithfully transfer a plan originally designed for assembling a table to the task of setting up an emergency communication network.

The schemas and planning scenario were chosen in favor of a pedagogical exposition, but this approach can be extended to incorporate more complex features relevant to both manufacturing assembly and the establishment of emergency communication networks, or other mission-critical operations. Such extensions could include more sophisticated relationship and constraints that mirror the intricacies of real-world scenarios.

Another advantage of this approach lies in its potential for handling sensitive information in mission-critical scenarios. By initially planning in an abstracted or innocuous domain, such as manufacturing assembly, the approach allows for the development of strategies and solutions without directly exposing any sensitive information related to the actual mission. Once the plan is formulated in the safe, analogous domain, it can be transferred to the target, sensitive setting, ensuring that the plan remains executable while keeping critical details secure.

Overall, direct analogy provides a straightforward mechanism where each object in one schema is mapped directly to an object in another schema. While this may seem like a trivial recasting of concepts from one planning scenario to another, this technique provides a general and well-behaved approach to analogical plan transfer with potential applicability across a wide range of transfer scenarios.

7.7 A Case Study of Structural Analogy: From Blocksworld to Kitchenworld

Building on the effective application of direct analogy to transfer plans between the Assembly and EmergencyComms domains, we now turn our attention to a more complex scenario involving structural analogy. To explore the concept of structural analogy in a practical context, we will examine the transfer of plans between two distinct yet structurally analogous domains: Blocksworld and Kitchenworld. Specifically, we focus on a variant of Blocksworld [133] which we call **ColorBlocksworld** and a kitchen object rearrangement problem [175] referred to as **Kitchenworld**, based on the AI2-THOR simulation environment. In ColorBlocksworld, a robot is tasked with arranging blocks into a goal configuration while keeping track of which block is in its gripper. Each block is uniquely identified and has a potentially non-unique color. In Kitchenworld, the robot arranges food ingredients and utensils into a goal configuration. These items possess attributes such as temperature, mass, and material. The structural similarities between stacking blocks and stacking ingredients suggest an underlying analogy. In this section, we will formalize this analogy and demonstrate how to transfer plans between these domains in a principled way. An overview of the plan transfer scenario is shown in Figure 7.7.

7.7.1 Domains

We begin by identifying the key ontological features of each domain. Both ColorBlocksworld and Kitchenworld must be able to express predicates such as the existence of blocks, ingredients, and robot grippers. Additionally, they should be capable of representing relationships, such as one entity being on top of another. Finally, both domains need to define attributes like color, temperature, and mass. These concepts can be captured using schema categories, denoted as ColorBlocksworld for ColorBlocksworld and KitchenWorld for Kitchenworld. The data migration

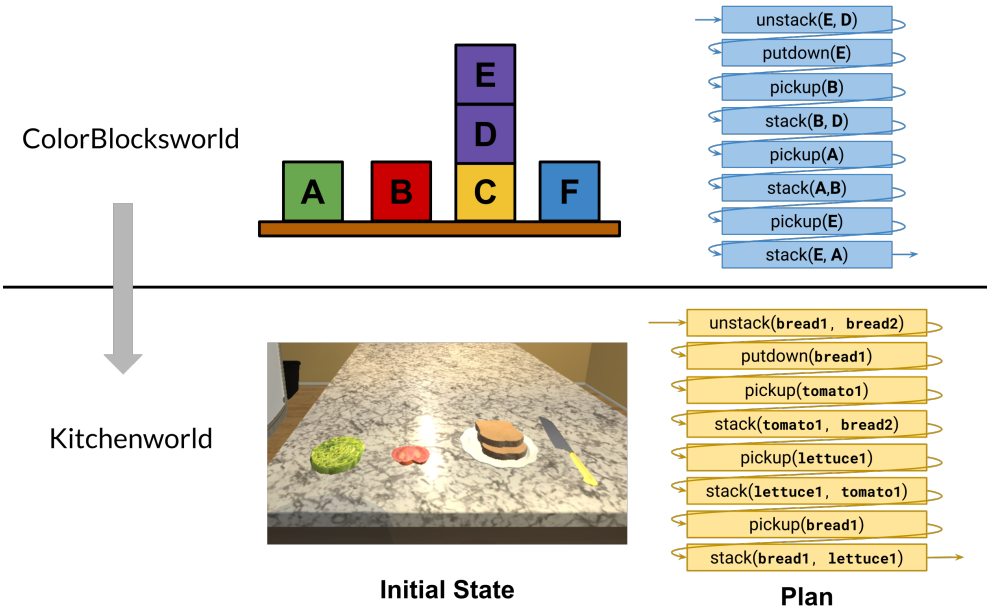


Figure 7.7: Overview of the structural analogy case study.

functor will transfer a plan from the source domain, ColorBlocksworld, to the target domain, KitchenWorld.

7.7.1.1 ColorBlocksworld Domain Ontology

The ColorBlocksworld schema category contains types, such as:

```
Block, Empty, Gripper
```

It also contains types for attributes, such as:

```
Clear, OnTable, Color
```

In this schema category, we include predicates such as:

```
isHolding::Hom(Gripper,Block)
```

which expresses that the `Gripper` is holding a `Block`. We also include the predicate:

```
isEmpty::Hom(Empty,Block)
```

which can be used to indicate that the gripper is holding nothing, making it empty. Additionally, a `Block` can have the following attributes:

```
hasColor::Attr(Block, Color)
```

which describes the color of the block,

```
isClear::Attr(Block, Clear)
```

which expresses whether or not a block is on top of another block, and

```
isOnTable::Attr(Block, OnTable)
```

which indicates whether or not a block is on the table. States in ColorBlocksworld assign these attribute types to sets of primitive data types such as `String` for `Color` and `Bool` for `Clear` and `OnTable`.

Lastly, the ontological type, `On`, is used to express relations between blocks:

```
On::Ob
on_l::Hom(On, Block)
on_r::Hom(On, Block)
```

where `on_l` maps to the block on top and `on_r` maps to the block on the bottom. A full specification of the ColorBlocksworld schema can be found in Appendix A.2.3.

7.7.1.2 Kitchenworld Domain Ontology

The Kitchenworld schema category was constructed by selecting a subset of types from the AI2-THOR world specification [176], focusing on those necessary for the task of building a sandwich. While the full ontology could theoretically be represented, this subset was intentionally chosen for pedagogical clarity and to simplify the case study demonstration. However, the proposed method is not limited to such a small ontology—a larger and more varied ontology, including more utensils or liquids, could be incorporated without fundamental changes to the methodology.

The Kitchenworld domain differs from ColorBlocksworld in its types and predicates, reflecting the distinct nature of a kitchen environment as opposed to a block-stacking one. Instead of blocks, the domain features types such as objects, receptacles, ingredients, and utensils. Un-

like ColorBlocksworld, where color is a primary attribute, Kitchenworld introduces attributes like temperature, mass, and material for its items.

More specifically, Kitchenworld includes types representing various ingredients, such as:

```
BreadSliced, LettuceSliced, TomatoSliced
```

as well as utensils, such as:

```
Plate, Knife
```

These items are all subtypes of a general AI2-THOR `Object`. The hierarchical relationship is expressed using predicates like:

```
plateIsObject::Hom(Plate, Object)
```

which can be read as "Every plate is an object."

Receptacles in Kitchenworld are entities that can hold or support other objects. Some ingredient and utensil types are also considered receptacles, expressed through predicates such as:

```
plateIsReceptacle::Hom(Plate, Receptacle)
```

In this ontology, bread slices, lettuce slices, tomato slices, and plates are classified as receptacles, whereas knives are not.

Each item in Kitchenworld can have attributes like temperature, mass, and material, as described in [176]. These attributes are represented as:

```
plateHasTemperature::Attr(Plate, Temperature)
plateHasMass::Attr(Plate, Mass)
plateHasMaterial::Attr(Plate, Material)
```

Similar predicates apply to other ingredient and utensil types. In Kitchenworld, states assign these attributes to sets of primitive data types, such as `Float64` for `Temperature` and `Mass`, and `String` for `Material`.

Kitchenworld shares some features with ColorBlocksworld, maintaining another sense of analogy between the two domains. It includes the types `Empty` and `Gripper`, along with their corre-

sponding predicates to indicate whether the robot gripper is holding an item or is empty. Additionally, Kitchenworld has a type analogous to the `On` type in ColorBlocksworld, called `InReceptacle`, which represents an object being placed in a receptacle. This type includes the predicates `inReceptacle_l` and `inReceptacle_r` to specify which object is in which receptacle. A full specification of Kitchenworld can be found in Appendix A.2.4.

7.7.2 Ontology Alignment

The domain ontologies have common features, but also differ in a few ways. To formally capture these similarities and differences, we can use a translation functor, F : Kitchenworld $\rightarrow$ ColorBlocksworld. This functor maps types and predicates from the target domain (Kitchenworld) to diagrams of types and predicates in the source domain (ColorBlocksworld) while preserving the composition of relationships. An illustration of these mappings can be seen in Figure 7.8. The dotted gray arrows in Figure 7.8 indicate the alignment of types and predicates from Kitchenworld to patterns of concepts, or namely diagrams (see Section 7.4.1), within ColorBlocksworld (illustrated within solid black rectangles). The solid black arrows represent morphisms between these diagrams.

We begin by mapping types and predicates with similar semantics in both domains:

```
% types
Gripper => Gripper
Empty => Empty
% predicates
isHolding => isHolding
isEmpty => isEmpty
```

Next, we address the central types in each domain, specifically `Block`, `Object`, and `Receptacle`. Here, we can use the intuition that a block in ColorBlocksworld represents a general item in Kitchenworld. Therefore, both objects and receptacles in Kitchenworld can be interpreted as blocks in ColorBlocksworld. Additionally, both domains have a way of expressing that one item is on top

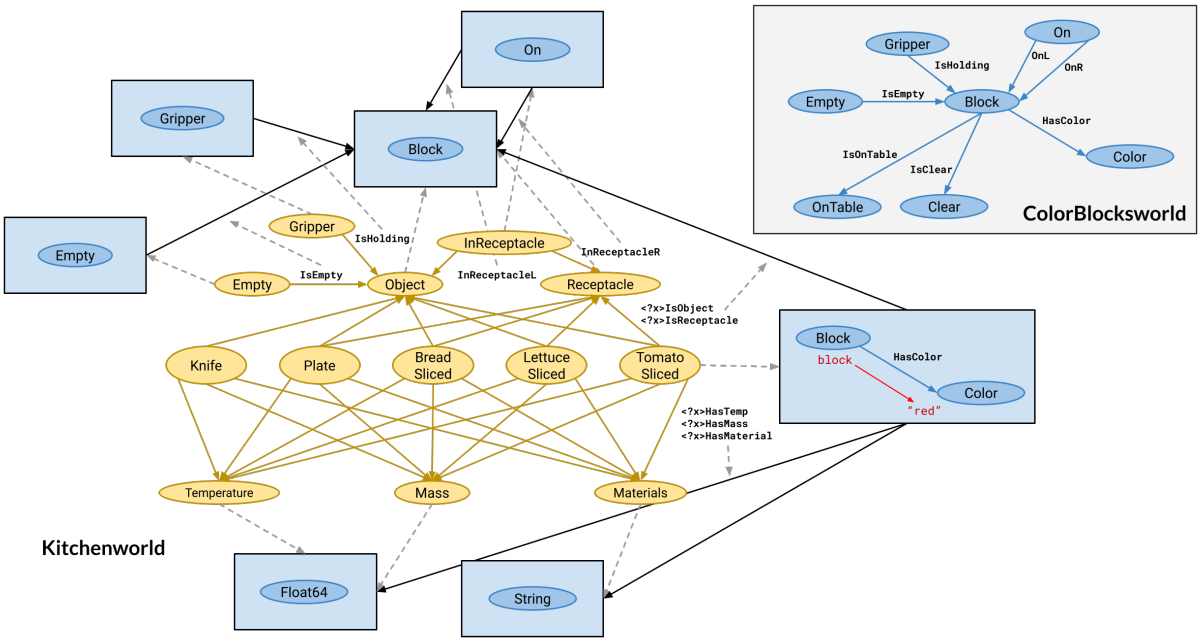


Figure 7.8: Illustration of ontology alignment using a structural analogy.

of another. Thus, we define the following component mappings for F:

```
% types
Object => Block
Receptacle => Block
InReceptacle => On
% predicates
inReceptacle_l => on_l
inReceptacle_r => on_r
```

ColorBlocksworld also includes information about block color for which a correspondence with ingredients and utensils can be established. For example, we can say that all red blocks map to tomato slices using the following assignment:

```
TomatoSliced => @join begin
  block::Block
  C::Color
  (hc:block → C)::hasColor
  (c:block → C)::(block -> "red")
end
```

This process can be repeated for each block color, where purple blocks correspond to bread slices, green blocks to lettuce slices, yellow blocks to plates, and blue blocks to knives.

Regarding attributes, since there are no equivalent notions of temperature, mass, or material in ColorBlocksworld, we map these attributes to primitive data types and assign default values for each type in Kitchenworld. For example:

```
Temperature => Float64
Mass => Float64
Material => String
```

For each predicate associated with `BreadSliced`, we can define functions that assign default values for every instance, such as:

```
breadSlicedHasTemperature => (x -> 0.0)
breadSlicedHasMass => (x -> 0.0)
breadSlicedHasMaterial => (x -> "Unknown")
```

The same approach can be applied to all other ingredient and utensil types.

It is important to note that certain attributes from ColorBlocksworld, such as `Clear` and `OnTable`, do not have direct analogs in Kitchenworld. As a result, the data associated with these attributes are not transferred, resulting in a lossy translation. While this loss is acceptable in this specific scenario, other situations might require adjustments to the domain ontologies to ensure a lossless translation.

7.7.3 Plan Transfer

Having defined an ontology alignment map from Kitchenworld to ColorBlocksworld, we can examine the results of applying a conjunctive query migration functor to a plan within the ColorBlocksworld domain. In this subsection, we will first define the specific problem within ColorBlocksworld, including the initial and goal states. Next, we will describe how the actions in the ColorBlocksworld domain were specified. Finally, we will demonstrate the effects of transferring a plan from ColorBlocksworld to Kitchenworld.

Table 7.2: Table comparing ColorBlocksworld-Set and Kitchenworld-Set.

ColorBlocksworld	Kitchenworld
`# Initial state` `gripper::Gripper` `empty::Empty` `isHolding(gripper) == isEmpty(empty)` `(A, B, C, D, E, F)::Block` `hasColor(A) == "green"` `hasColor(B) == "red"` `hasColor(C) == "yellow"` `hasColor(D) == "purple"` `hasColor(E) == "purple"` `hasColor(F) == "blue"` `(x1, x2)::On` `on_l(x1) == E` `on_r(x1) == D` `on_l(x2) == D` `on_r(x2) == C` `isOnTable(A) == true; isClear(A) == true` `isOnTable(B) == true; isClear(B) == true` `isOnTable(C) == true; isClear(C) == false` `isOnTable(D) == false; isClear(D) == false` `isOnTable(E) == false; isClear(E) == true` `isOnTable(F) == true; isClear(F) == true`	`# Initial state (after transfer)` `gripper::Gripper` `empty::Empty` `isHolding(gripper) == isEmpty(empty)` `A::LettuceSliced` `B::TomatoSliced` `C::Plate` `D::BreadSliced` `E::BreadSliced` `F::Knife` `(x1, x2)::InReceptacle` `inReceptacle_l(x1) == BreadSlicedIsObject(E)` `inReceptacle_r(x1) == BreadSlicedIsReceptacle(D)` `inReceptacle_l(x2) == BreadSlicedIsObject(D)` `inReceptacle_r(x2) == PlateIsReceptacle(C)` `lettuceHasTemperature(A) == 0.0` `lettuceHasMass(A) == 0.0` `lettuceHasMaterial(A) == "Unknown"` `tomatoHasTemperature(B) == 0.0` `tomatoHasMass(B) == 0.0` `tomatoHasMaterial(B) == "Unknown"` `...`
`# Unstack precondition` `gripper::Gripper` `empty::Empty` `isHolding(gripper) == isEmpty(empty)` `(block, underblock)::Block` `o::On` `on_l(o) == block` `on_r(o) == underblock` `isClear(block) == true`	`# Unstack precondition (after transfer)` `gripper::Gripper` `empty::Empty` `isHolding(gripper) == isEmpty(empty)` `block::Object` `underblock::Receptacle` `o::InReceptacle` `inReceptacle_l(o) == block` `inReceptacle_r(o) == underblock`
`# Match morphism` `gripper::Gripper -> gripper::Gripper` `empty::Empty -> empty::Empty` `block::Block -> E::Block` `underblock::Block -> D::Block` `o::On -> x1::On`	`# Match morphism (after transfer)` `gripper::Gripper -> gripper::Gripper` `empty::Empty -> empty::Empty` `block::Object -> E::Object` `underblock::Receptacle` `-> D::Receptacle` `o::InReceptacle -> x1::InReceptacle`

7.7.3.1 ColorBlocksworld problem

Consider a problem in ColorBlocksworld where six blocks, labeled `A, B, C, D, E, F`, are initially arranged on a table, as shown in Figure 7.9. Each block is assigned one of the fol-

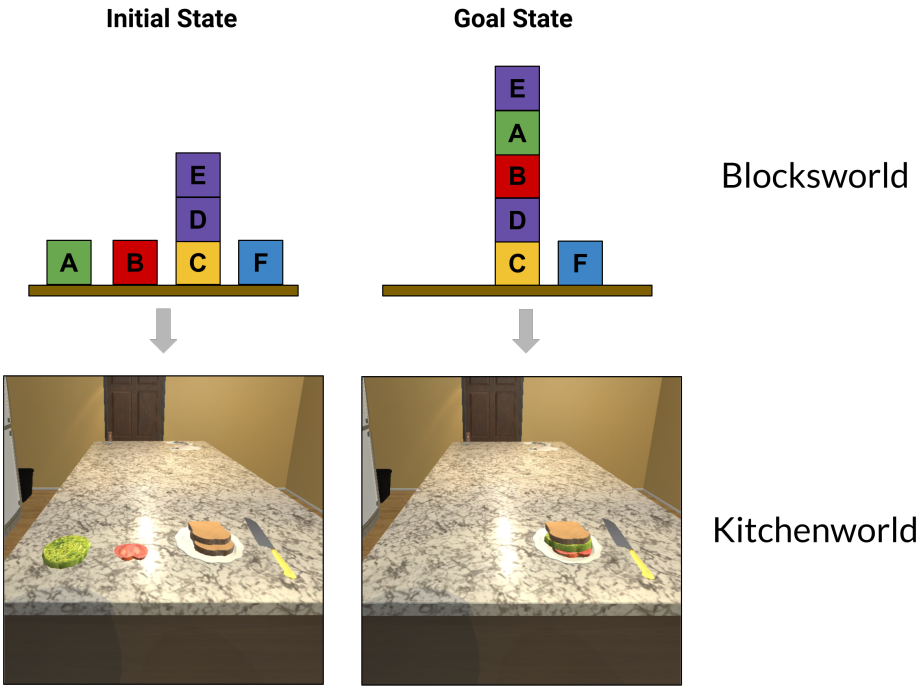


Figure 7.9: The initial and goal states in Blocksworld and Kitchenworld.

lowing colors: `"red"`, `"green"`, `"blue"`, `"purple"`, or `"yellow"`. For instance, blocks `E` and `D` are both purple. The goal is to stack the blocks such that their colors form the sequence `"purple, green, red, purple, yellow"` from top to bottom. The initial and goal states can be defined as functors in ColorBlocksworld-Set by mapping each type in ColorBlocksworld to an element in Set. An example of the initial state specification is provided in Table 7.2. The goal state would be defined similarly.

7.7.3.2 ColorBlocksworld Domain Actions

The actions of the ColorBlocksworld domain are modeled after the PDDL specification of Blocksworld [177]. For instance, two fundamental actions in this domain are `stack` and `unstack`. The `stack` action, `stack(block::Block, underblock::Block)`, places the `block` block on top of the `underblock` block, provided that the gripper is holding the `block` block. Conversely, the `unstack` action, `unstack(block::Block, underblock::Block)`, removes the `block` block from

the `underblock` block and places it in the gripper.

Both actions have preconditions that involve a conjunction of predicates, which specify whether the `block` block is currently on the `underblock` block and whether the gripper is empty or occupied. The effects of these actions involve modifying these predicates to reflect the new state of the world after the action has been executed. Specifically, the effects are the complement of the preconditions—indicating that the blocks and the gripper are now in the opposite Boolean states.

In our representation language, the `unstack` action can be expressed using the following action span:

```
unstack = DRule(@migration(OntColorBlocksworld, begin
            L => @join begin
              gripper::Gripper
              null::Empty
              (block, underblock)::Block
              o::On
              on_l(o) == block
              on_r(o) == underblock
              isClear(block) == true
              isHolding(gripper) == isEmpty(null)
            end
            K => @join begin
              gripper::Gripper
              (block, underblock)::Block
            end
            R => @join begin
              gripper::Gripper
              (block, underblock)::Block
              isClear(underblock) == true
              isHolding(gripper) == block
            end
          end))
```

In this example, the `Pre` condition checks if the gripper is empty and whether the `block` block (`block`) is clear (i.e., has no other block on top of it) and is currently on top of another block (`underblock`). The `Keep` section identifies the entities that remain unchanged, and the `Eff` condition specifies the effects of the action, where the `block` block is now held by the gripper, and the `bottom` block is clear.

The remaining actions in Blocksworld, such as `stack` and others, can be expressed in a

similar manner using this representation.

7.7.3.3 Transfer from ColorBlocksworld to Kitchenworld

To transfer a plan, we must transfer every action and its match. Consider a plan where the first action is to unstack block `E` from `D`. The match for the `unstack` action grounds `block` $\mapsto$ `E` and `underblock` $\mapsto$ `D`. The precondition and match in Blocksworld can be seen in Table 7.2.

Table 7.2 shows the transferred initial state from ColorBlocksworld to Kitchenworld. Although the names of instances (e.g., `red, green, gripper`) remain the same, they are now instances of types in Kitchenworld. The transferred precondition of the `unstack` action is also presented in Table 7.2. The `Keep` and `Eff` components undergo similar transfers. The result is that the types have been translated from ColorBlocksworld to Kitchenworld, with the semantics of the predicates preserved.

A plan, $\pi_{\text{ColorBlocksworld}}$ for this problem in ColorBlocksworld looks like:

```
unstack(E, D); putdown(E); pickup(B); stack(B, D);
pickup(A); stack(A, B); pickup(E); stack(E, A)
```

The same plan, when transferred to Kitchenworld, $\pi_{\text{Kitchenworld}}$, becomes:

```
unstack(bread1, bread2); putdown(bread1);
pickup(tomato); stack(tomato, bread2);
pickup(lettuce); stack(lettuce, tomato);
pickup(bread1); stack(bread1, lettuce);
```

The names of the instances are, again, pointers to elements in a set, therefore, the instances can be relabeled to contextually appropriate terms like `bread1` and `bread2` instead of `E` and `D` for this exposition.

7.7.4 Validation

In the case of conjunctive query data migrations, the functoriality of Δ_F is not guaranteed, particularly when dealing with complex structures or attributes such as numeric values or multi-

Table 7.3: Table comparing the goal and end states to validate $\pi_{\text{Kitchenworld}}$

ColorBlocksworld	Kitchenworld
`# Goal state` `(A, B, C, D, E, F)::Block` `hasColor(A) == "green"` `hasColor(B) == "red"` `hasColor(C) == "yellow"` `hasColor(D) == "purple"` `hasColor(E) == "purple"` `hasColor(F) == "blue"` `(x1, x2, x3, x4)::On` `on_l(x1) == E` `on_r(x1) == A` `on_l(x2) == B` `on_r(x2) == D` `on_l(x3) == B` `on_r(x3) == D` `on_l(x4) == D` `on_r(x4) == C`	`# Goal state (after transfer)` `A::LettuceSliced` `B::TomatoSliced` `C::Plate` `D::BreadSliced` `E::BreadSliced` `F::Knife` `(x1, x2, x3, x4)::InReceptacle` `inReceptacle_l(x1) == BreadSlicedIsObject(E)` `inReceptacle_r(x1) == LettuceSlicedIsReceptacle(A)` `inReceptacle_l(x2) == LettuceSlicedIsObject(A)` `inReceptacle_r(x2) == TomatoSlicedIsReceptacle(B)` `inReceptacle_l(x3) == TomatoSlicedIsObject(B)` `inReceptacle_r(x3) == BreadSlicedIsReceptacle(D)` `inReceptacle_l(x4) == BreadSlicedIsObject(D)` `inReceptacle_r(x4) == PlateIsReceptacle(C)`
`# Final State` `gripper::Gripper` `empty::Empty` `isHolding(gripper) == isEmpty(empty)` `(A, B, C, D, E, F)::Block` `hasColor(A) == "green"` `hasColor(B) == "red"` `hasColor(C) == "yellow"` `hasColor(D) == "purple"` `hasColor(E) == "purple"` `hasColor(F) == "blue"` `(x1, x2, x3, x4)::On` `on_l(x1) == E` `on_r(x1) == A` `on_l(x2) == B` `on_r(x2) == D` `on_l(x3) == B` `on_r(x3) == D` `on_l(x4) == D` `on_r(x4) == C` `isOnTable(A) == false; isClear(A) == false` `isOnTable(B) == false; isClear(B) == false` `isOnTable(C) == true; isClear(C) == false` `isOnTable(D) == false; isClear(D) == false` `isOnTable(E) == false; isClear(E) == true` `isOnTable(F) == true; isClear(F) == true`	`# Final State (after transfer)` `gripper::Gripper` `empty::Empty` `isHolding(gripper) == isEmpty(empty)` `A::LettuceSliced` `B::TomatoSliced` `C::Plate` `D::BreadSliced` `E::BreadSliced` `F::Knife` `(x1, x2, x3, x4)::InReceptacle` `inReceptacle_l(x1) == BreadSlicedIsObject(E)` `inReceptacle_r(x1) == LettuceSlicedIsReceptacle(A)` `inReceptacle_l(x2) == LettuceSlicedIsObject(A)` `inReceptacle_r(x2) == TomatoSlicedIsReceptacle(B)` `inReceptacle_l(x3) == TomatoSlicedIsObject(B)` `inReceptacle_r(x3) == BreadSlicedIsReceptacle(D)` `inReceptacle_l(x4) == BreadSlicedIsObject(D)` `inReceptacle_r(x4) == PlateIsReceptacle(C)` `lettuceHasTemperature(A) == 0.0` `lettuceHasMass(A) == 0.0` `lettuceHasMaterial(A) == "Unknown"` `tomatoHasTemperature(B) == 0.0` `tomatoHasMass(B) == 0.0` `tomatoHasMaterial(B) == "Unknown"` `...`

typed relationships. This lack of guaranteed functoriality introduces challenges when validating that the transferred plan $\pi_{\text{Kitchenworld}}$ satisfies the goal in the target domain, Kitchenworld. Therefore, the validation of the transferred goal requires a different process to ensure that the final state in the target domain accurately reflects the transferred goal.

7.7.4.1 Transferring Actions and States

As with delta data migrations, we begin by transferring the match morphisms and states from the source domain, ColorBlocksworld, to the target domain, Kitchenworld. For the last action in the source plan $\pi_{\text{ColorBlocksworld}}$, the match morphism, $m_n : \text{Pre}_{\text{stack}} \hookrightarrow X_n$, is transferred to $\Delta_F(m_n) : \Delta_F(\text{Pre}_{\text{stack}}) \hookrightarrow \Delta_F(X_n)$. However, unlike delta data migrations, applying Δ_F does not necessarily preserve the DPO construction, which means that the transferred morphism $\Delta_F(m_n)$ may not yield a well-defined pushout in the target domain.

Without the guarantee of preserved pushouts, the final state Y_{n+1} in the target domain must be explicitly recomputed. This involves applying the transferred match morphism, $\Delta_F(m_n)$, and action rule (e.g., the transferred `stack(bread1, lettuce)` action) within the target domain. The final state can be computed by resolving the DPO construction in the target domain for which the result is shown in the second row of Table 7.3.

7.7.4.2 Validating the Transferred Goal

To validate that the transferred plan $\pi_{\text{Kitchenworld}}$ satisfies the goal in the target domain, we must verify that the transferred goal $\Delta_F(G)$ is included in the final state Y_{n+1}. To do this, we can search for a valid monomorphism that matches $\Delta_F(G)$ within Y_{n+1}.

A valid homomorphism in this case, matches elements in the analogous goal state within Kitchenworld to elements of the state in Kitchenworld, is as follows:

```
% Goal -> Final State
A::LettuceSliced -> A::LettuceSliced
B::TomatoSliced -> B::TomatoSliced
C::Plate -> C::Plate
D::BreadSliced -> D::BreadSliced
E::BreadSliced -> E::BreadSliced
F::Knife -> F::Knife
x1::InReceptacle -> x1::InReceptacle
x2::InReceptacle -> x2::InReceptacle
...
```

For this specific transfer from $\pi_{\mathsf{ColorBlocksworld}}$ to $\pi_{\mathsf{Kitchenworld}}$, a homomorphism search algorithm[3] confirms that a valid match from the goal state into the final state has been found which validates that the transferred plan is successful.

7.7.4.3 Addressing Inconsistencies

If the validation process reveals that the transferred goal does not exactly match the final state, it becomes necessary to address any inconsistencies. This may involve refining the conjunctive queries by modifying the mappings between the source and target domains to improve alignment. Further post-transfer adjustments may be required to ensure that the transferred plan properly aligns with the requirements of the target domain and satisfies the intended goals.

7.7.5 Summary

In this case study, we explored the value of using structural analogy to transfer task plans from the ColorBlocksworld domain to the Kitchenworld domain, demonstrating how planning in a simpler, abstract domain can be effectively adapted to a more complex one without replanning. By first solving the problem in the abstracted block-stacking domain, we reduced the number of possible predicates when compared to a rich kitchen domain. This approach not only simplifies the planning process but also has the potential to be used as a valuable planning heuristic: the efficiency of a plan in the abstract domain, such as the fewer number of steps required, can inform the development of an optimal plan in the complex domain. Additionally, the abstract domain serves as a powerful template for replicating the behavior for different problems within home automation, such as stacking linens or books around a home.

[3] `https://github.com/AlgebraicJulia/Catlab.jl/blob/6fd7520c7c8bf732d2b4a07b51930262d3c68298/src/categorical_algebra/HomSearch.jl`

7.8 Practical Considerations

The proposed approach offers a principled method for transferring task plans from one planning context to another by leveraging analogies. We demonstrated its utility by transferring plans from a simple domain, ColorBlocksworld, to more complex domains, namely Assembly, EmergencyComms, and Kitchenworld. While this work focuses on foundational examples, it is possible to envision a variety of design patterns that leverage analogical transfer. In this section, we discuss these general patterns, its integration into a planning architecture, and considerations when defining ontology maps.

7.8.1 Patterns of Use for Analogical Plan Transfer

As demonstrated by the case studies in Section 7.3, planning domains can differ in their level of detail and generalization. An *abstract* domain represents a simplified or generalized environment with fewer and richly-defined concepts, such as Blocksworld. In contrast, a *specific* domain is a detailed and complex environment with many domain-specific attributes, such as Kitchenworld, Assembly, EmergencyComms. Based on this distinction, we have identified four patterns for transferring plans analogically: (i) abstract-to-specific, (ii) specific-to-specific, (iii) specific-to-abstract-to-specific, and (iv) abstract-to-abstract. Each pattern serves a unique purpose and highlights the versatility of the approach. A schematic of each pattern is shown in Figure 7.10.

7.8.1.1 Abstract-to-Specific

This pattern is exemplified by the Blocksworld-to-Kitchenworld example presented in this paper. In this approach, a generic plan developed in an abstract domain is transferred to a specific domain. The advantage of this pattern lies in its ability to reapply generalized behaviors to specific contexts without compromising the integrity of the original behavior.

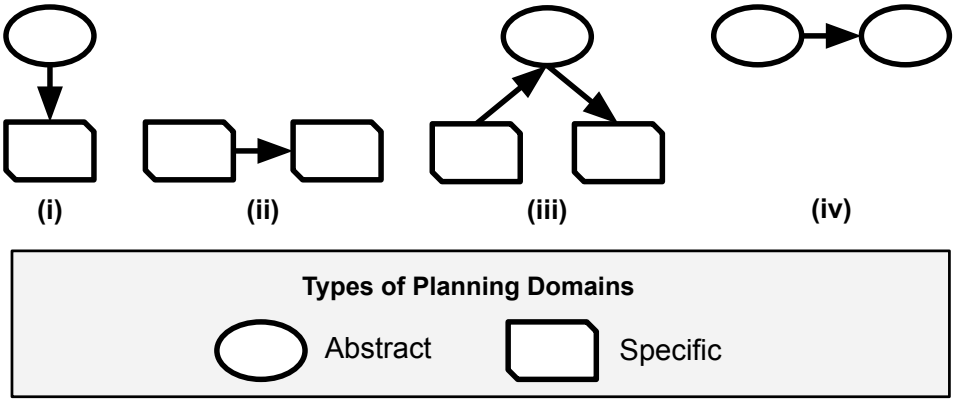


Figure 7.10: Schematic of the four patterns of analogical plan transfer: (i) abstract-to-specific, (ii) specific-to-specific, (iii) specific-to-abstract-to-specific, and (iv) abstract-to-abstract.

An extended application of this pattern could involve migrating a plan from a simulation environment—where domain concepts are fewer and well-defined—to a real-world environment [178], which is richer but less structured. For example, reasoning in a real-world kitchen might require capturing additional relationships, such as the relative spatial alignment of objects on a cluttered countertop. This pattern enables structured actions from a simplified domain to be effectively used in more complex and dynamic settings.

7.8.1.2 Specific-to-Specific

In this pattern, a plan is directly transferred from one specific domain to another. In the Assembly to EmergencyComms case study, we demonstrate the transfer the plans from a specific domain tailored for building furniture to another specific domain tailored for building communications networks. One could envision another example where a plan for tending a shopfloor machine might be adapted to a similar domain-specific task, such as managing an industrial oven for food production—both requiring managing items being processed in a machine. The advantage of this approach lies in its ability to replicate context-specific behaviors that may not be easily generalized or articulated in an abstract domain.

This pattern closely aligns with ongoing research in skill generation [48, 179–181], where plans are designed for highly specialized tasks. By transferring such plans directly between specific domains, it becomes possible to preserve nuanced, context-dependent behaviors that would otherwise be lost during abstraction.

7.8.1.3 Specific-to-Abstract-to-Specific

This pattern involves transferring a plan from one specific domain to an abstract domain, distilling its essential structure, and then applying it to a different specific domain. For example, Factory A's machine-tending plan could be generalized into an abstract template, which is subsequently tailored to fit the operational needs of Factory B.

The primary benefit of this pattern is its ability to generalize and distill the key elements of a complex plan, making it applicable to a wider range of scenarios. This is particularly valuable in privacy-sensitive collaborations [182–185], where manufacturers may wish to adhere to a shared abstract template while safeguarding proprietary details. By relying on the abstract-specific transfer, they can impose generic behavior in different contexts while maintaining confidentiality.

7.8.1.4 Abstract-to-Abstract

This pattern involves transferring plans between two abstract domains. For example, a plan from the Tower of Hanoi domain [186] could be adapted for the Blocksworld domain. This pattern introduces a novel concept in AI planning: reframing a planning problem from one canonical abstract domain to another to potentially gain computational or reasoning advantages. For instance, a planner might exploit constraints related to relative size and position in the Tower of Hanoi to identify an intermediate subgoal, then continue planning in Blocksworld.

This pattern holds significant potential for algorithmic reframing of abstract search problems

in robotics applications. In particular, it could prove useful in scenarios where the structured constraints of a Tower of Hanoi problem offer greater efficiency compared to the less structured nature of a Blocksworld problem. Exploring how such reframing can improve planning efficiency and problem-solving capabilities is a promising direction for future research.

7.8.2 Integration into a Planning Architecture

The aforementioned transfer use cases can be integrated into a planning architecture similar to case-based planning [187]. In this framework, source plans are retrieved from a historical knowledge base, transferred, and adapted to suit specific use cases. The transferred plan acts as a skeleton or backbone for the new plan, ensuring that analogous behavior is preserved as long as the sequence of actions is maintained. The approach may require plan repair, particularly when actions in the transferred plan must be decomposed into more granular actions. For instance, translating a high-level plan or program into a low-level robot programming language might necessitate finer action descriptions [188]. Developing a mathematically rigorous framework to address such decompositions, perhaps through hierarchical transfer methods, is an area of ongoing research.

7.8.3 Designing an Ontology Map

The success of all the aforementioned patterns, in Section 7.8.1, is contingent on the ability to establish a valid ontology map between the source and target domains. The design of the ontology map is critical, because it directly influences the semantics and feasibility of the transferred plan. The structure and mappings of an ontology map dictate whether a meaningful and correct analogy between domains can be established. For instance, consider an inappropriate translation for the Kitchenworld and Blocksworld domains that involves mapping the `Gripper` from Kitchenworld to `Block` in Blocksworld—expressing a nonsensical analogy. Validating this mapping against the

properties of a functor would reveal that no proper ontology map can be defined, thus preventing plan transfer. Alternatively, a semantically flawed but technically valid translation might involve swapping the mappings of `inReceptacle_l` to `on_r` and `inReceptacle_r` to `on_l`. This reversal would invert the stacking order in Kitchenworld, leading to a semantically undesirable plan.

7.8.3.1 Failure to Define an Ontology Map

The inability to define an ontology map indicates the absence of a proper analogy between domains, which prevents reliable plan transfer. Fortunately, such failures are identified during the design stage of the translation process, allowing the issue to be addressed before runtime.

It is, however, important to note that the failure to define an ontology map may sometimes stem from structural mismatches in the domain ontologies rather than an inherent inability to transfer plans. Careful design of both source and target ontologies is therefore critical. Ontology development requires deliberate decisions to represent concepts in ways that facilitate alignment.

For example, consider an alternate choice to make `Receptacle` a subtype of `Object`:

$$\texttt{isSubtypeOf} : \texttt{Receptacle} \to \texttt{Object}$$

in Kitchenworld. While semantically valid, this design choice would prohibit an ontology map that assigns both `Object` and `Receptacle` to `Block` in Blocksworld:

```
Receptacle => Block
  Object => Block
```

as no predicate morphism exists from `Block` to `Block` in Blocksworld (refer to Appendix A.2.3). By contrast, separating `Receptacle` and `Object` in Kitchenworld, as in the current specification, better aligns with the section of the schema pertaining to `Block` in Blocksworld. This example underscores the importance of ontology design in ensuring successful plan transfer.

7.8.3.2 Addressing Non-One-to-One Ontology Mappings

Ontological misalignments between domains are a common challenge in practice, emphasizing the importance of conjunctive query migrations for capturing nuanced relationships. Structural features such as the size of an ontology—defined by the number of types ($|\cdot|$)—or the overlap of aligned content can significantly influence the success of a plan transfer. These features give rise to the following scenarios.

|Target| > |Source|: Stratifying State Data. As described in the case study in Section 7.7, a task plan originating in Blocksworld was successfully transferred to Kitchenworld. In this example, the Kitchenworld domain ontology is larger than the Blocksworld ontology, requiring multiple types in the target domain to be mapped to a single type, `Block`, in the source domain. This mapping stratifies block instances, aligning them with different types of ingredients in Kitchenworld. The conjunctive query migration functor plays the crucial role of identifying and aligning these stratifications, ensuring that instances of the Kitchenworld ontology are populated correctly.

|Target| < |Source|: Merging State Data. In scenarios where a plan is transferred from Kitchenworld to Blocksworld, the target domain ontology is smaller than the source ontology. This results in the loss of distinctions, such as different types of ingredients, as all objects are reduced to a single `Block` type. In such cases, the designer must carefully evaluate whether this simplification affects the validity or applicability of the transferred plan, particularly when domain-specific attributes are critical for task success.

Target ∩ Source: Information Loss. When the source and target ontologies overlap but have non-identical structures, some data loss is inevitable. This loss may be acceptable in certain contexts but detrimental in others. For instance, attributes like `isClear`, `onTable`, `mass`, and `temperature`

might be omitted unless explicitly accommodated in the target ontology. Acceptable loss occurs when the goal is to transfer the sequence of stacking actions without leveraging detailed attributes, which means such omissions may not affect the transfer's success. Conversely, detrimental loss arises when plans rely on specific attribute values, such as ensuring a weight limit of 100 lb for stacking. Defaulting these values to placeholders, such as `weight = 0`, can invalidate the plan. Addressing these issues requires additional validation steps, such as populating missing values or adapting the transferred plan. Developing a mathematically robust framework for handling missing attributes remains a promising area for future research.

7.9 Discussion

In this chapter, we have demonstrated a novel approach to analogical transfer for task plans, using both direct and structural analogies. Unlike classical representations such as PDDL, which are designed primarily for planning within a single domain, our approach facilitates the migration of plans between different domains. Classical representations lack native mechanisms that support analogical transfer and complex domain mappings, making this approach particularly innovative. The case studies presented were carefully selected to effectively illustrate the capabilities of each analogical type. In this section, we will discuss what we see to be the benefits and limitations of this approach as a solution to analogical plan transfer.

7.9.1 Advantages of the Categorical Structure for Analogical Transfer

Generalized Transfer Across Domains. One of the benefits of this approach is its ability to provide a generalized plan transfer mechanism that works uniformly across all plans within a given domain. Unlike methods that require pointwise mappings between pairs of plans, this approach leverages the foundational mappings between the source and target domains to ensure that any valid plan

in the source domain can be systematically and accurately transferred to the target domain. This universality means that once the domain ontologies are aligned, the transfer process is not just applicable to a single plan but can be broadly applied to all plans between the domains.

Structure Preserving Transfer. A key advantage of this approach lies in its ability to preserve the structural relationships inherent in the source domain during the process of plan transfer. By leveraging the mathematical rigor of functors and data migrations, the approach ensures that the compositional relationships between elements in the source domain are faithfully maintained in the target domain. This preservation is crucial for maintaining the integrity and validity of the transferred plans, as it ensures that the semantic relationships informing actions and states remain analogically consistent. In practice, this means that complex relational structures, such as type hierarchies, predicate relationships, and compositional constraints, are not only mapped but also preserved during the transfer process. This significantly reduces the risk of introducing errors or inconsistencies, which is particularly beneficial in domains with intricate interdependencies, such as manufacturing assembly tasks, mission planning, or kitchen service automation.

Planning Through Simplification. Another benefit is its ability to simplify planning in complex domains. This method leverages the inherent similarities between domains to reduce the computational burden associated with planning directly in a complex environment. By first solving the problem in a simpler domain, as in ColorBlocksworld—where the relationships and constraints are easier to manage—and then transferring to the more complex domain, this has the potential to not only accelerate the planning process but also reduce the cognitive load on human operators, who would otherwise need to define and manage actions in the complex domain from the beginning.

Action (Skill) Generation. Another advantage of this approach is its ability to automatically transfer actions from the source domain to the target domain. As plans in the source domain are migrated, the process generates corresponding actions in the target domain—a concept sometimes referred to as *skill generation*. Skill generation involves modeling new skills that were previously unknown to the agent, a challenge that drives a significant amount of research in robotics [48, 179–181]. In the case studies, it was not necessary to manually specify action models in the Kitchenworld or EmergencyComms domain; instead, the actions were automatically generated with a structure similar to those in the ColorBlocksworld and Assembly domain, respectively. This implies that the approach can effectively bootstrap domain actions from existing domains, reducing the human effort required to set up domain actions in new, but structurally similar, environments.

Wide Applicability. One of the most compelling strengths of this approach is ability to be applied to a wide range of domains, owing to its foundation in category theory. For instance, the approach has been successfully demonstrated in domains as varied as block-stacking, assembly tasks, and more sophisticated scenarios like kitchen automation and emergency communication planning. This transfer approach provides an abstraction suitable for a variety of domain ontologies. This generality alleviates the need to tailor the framework to suit different applications, suggesting that the approach is not just a solution for a narrow set of problems but one capable of addressing a wide spectrum of plan transfer scenarios.

7.9.2 Limitations of the Categorical Structure for Analogical Transfer

Dependency on Proper Ontology Alignment. The success of the analogical transfer approach is highly dependent on a proper (functorial) alignment of ontologies between the source and target domains, as discussed in Section 7.8.3. Misalignments or errors in defining these mappings can

lead to incorrect plan transfers, resulting in failures to achieve the desired outcomes in the target domain. The functor, for example, requires that all predicates in the target domain be mapped to corresponding predicates in the source domain in a way that preserves compositional relationships. For instance, if a type in the target domain is involved in multiple predicates—such as `Part` in Assembly—it cannot be effectively mapped to a type in the source domain that lacks structurally similar predicates, such as `Gripper` in ColorBlocksworld. These challenges in expressivity can be significant for certain applications, but they also highlight opportunities for future research aimed at developing mathematical tools to handle more complex mappings.

Potentially Lossy Translations. As noted in Section 7.7, a key challenge in analogical plan transfer is the occurrence of lossy translations. This issue arises when certain attributes or elements in the source domain lack direct counterparts in the target domain, leading to incomplete or approximate transfers of information. For example, in the case study involving the transfer from ColorBlocksworld to Kitchenworld, attributes such as whether the block was clear of obstruction (`IsClear`) in ColorBlocksworld did not have a corresponding concept in Kitchenworld, resulting in the loss of this information. If such information plays a crucial role in the planning process, the transferred plan may need manual interventions to ensure successful execution in the target domain. Addressing these challenges may also require modifying the expressivity of the target ontology or exploring alternative methods to manage lossy translations more effectively.

Handling Structurally Dissimilar Domains. While the analogical transfer approach is highly effective in domains with strong structural similarities, it can encounter significant challenges when applied to domains that are structurally dissimilar or where analogies are not easily identifiable. In these cases, the transfer process may require more complex mappings, or it might struggle to establish a meaningful analogy altogether. As a result, while powerful, it may not generalize well

across all types of plan transfer problems. The reliance on well understood structural similarities limits its applicability to situations where such analogies exist.

7.10 Conclusion

In this chapter, we explored a novel approach to analogical transfer for task plans, identifying two types of analogies—direct and structural—and demonstrating their application through functorial data migration methods. The chapter delved into two case studies, one involving the transfer of a plan from an assembly domain to an emergency communications recovery domain, and another illustrating the transfer from ColorBlocksworld to a domain inspired by AI2-THOR's kitchen simulation environment. These examples highlighted the flexibility and generality of the approach by showing how it enables the transfer of plans between complex domains. The discussion emphasized the benefits, such as general and structure-preserving plan transfer, automatic generation of actions in the target domain, and the ability to simplify planning in complex domains by leveraging simpler ones. We also addressed the limitations, including the dependency on accurate ontology alignment and the challenges posed by lossy translations and structurally dissimilar domains. Overall, the chapter demonstrated the potential of this approach as a robust tool for analogical plan transfer for a variety of plan transfer scenarios.

Chapter 8: Conclusion

In this dissertation, we explored task plan transfer in robotics by developing a framework that defines sequential, hierarchical, and analogical transfer methods contained within the single foundational language of category theory. This chapter concludes the dissertation by summarizing the research, highlighting its interdisciplinary impact, discussing areas for future work and potential applications, and offering closing remarks.

8.1 Summary of Research

Early in the dissertation, we traced the evolution of robotic programming reuse from manual pose recording to modular, skill-based architectures to transfer learning and program translation in robotics. This historical overview laid the foundation for classifying task plan transfer use cases into three types: sequential, hierarchical, and analogical. By identifying these scenarios and defining plan transfer within a unifying conceptual framework in Chapters 1 and 2, we established a model for enabling robots to transfer plans both within and across planning domains. This framing provided a blueprint for deeper exploration of the mathematical tools essential to supporting effective plan transfer. In Chapter 3, we introduced category theory as the foundational formalism. Category theory provides mathematical structures that address the limitations of conventional planning systems by ensuring that plan compositions and migrations are coherent and well-defined.

In Chapter 4, we presented an approach to validating sequential plan transfer. This method

ensures that the preconditions and effects of plans are consistently propagated, using symmetric monoidal categories (SMCs) and string diagrams to validate task plan composition across the entire sequence. In Chapter 5, we introduced the RoboCat framework which leverages functors to connect high-level plans with low-level ones. This framework ensures that hierarchies of plans and programs within the categorical architecture maintain both compositional integrity and intended behavior.

In later chapters, we introduced a novel planning representation language based on C-sets and double-pushout (DPO) rewriting to manage structured and interconnected knowledge (see Chapter 6). This representation language served as a prerequisite for analogical transfer (see Chapter 7). We introduced an algorithm that automates analogical plan transfer using ontology alignment maps and functorial data migrations. Each type of transfer was demonstrated through case studies involving diverse planning domains, including Blocksworld, an AI2-THOR kitchen environment, a manufacturing assembly environment, a machine-tending scenario, and a communication recovery scenario.

By redefining task plan transfer through a unified framework grounded in category theory, this dissertation not only addresses the challenges of adaptability in robot planning but also opens up new pathways for formally robust intelligence.

8.2 Interdisciplinary Impact: Expanding Theoretical Tools for AI and Robotics

Category theory, while often regarded as a highly abstract mathematical framework, has shown promise in direct applications to robotics and AI through this dissertation. Specifically, this work shows that category-theoretic constructs—such as SMCs and C-sets—can effectively address key challenges in task composition, modularity, and reuse within robot planning systems. Traditional task planning frameworks often overlook plan transfer or treat reuse scenarios in isola-

tion, failing to recognize the potential for unified approaches that can address diverse plan transfer scenarios. By introducing formal methods for modular programming, task plan composition, and ontology-driven reasoning, this work establishes a new paradigm in which category theory becomes an essential toolkit for robotics and AI.

Ultimately, this dissertation, alongside other works [35, 41, 102, 189–191], contributes to the shifting perspective that category theory is not merely an abstract discipline but a powerful, actionable tool for bridging theory and real-world challenges. As robotics and AI evolve, the need for systems that can learn, adapt, and transfer knowledge across tasks becomes paramount. By showing that category-theoretic principles can meet these needs, this research paves the way for robotic systems that not only execute interdependent tasks but also adapt safely to new environments with formal guarantees.

8.3 Areas of Future Work

The most exciting aspect of this work is the outlook for the future. By offering advancements in robotic task plan transfer, this dissertation opens several new directions for further exploration and development. In this section, we break down these opportunities according to each type of task plan transfer and then discuss a broader, integrated perspective on future work.

8.3.1 Sequential Plan Transfer

Although sequential transfer provides a robust framework for task composition, several key directions for future research remain.

Expanding beyond STRIPS representations. Modern planning often requires more expressive systems that exceed the limitations of STRIPS planning frameworks, such as reactive control [192–194], conditional branching [195, 196], and temporal reasoning [13, 197, 198]. Understanding

how sequential composition can be adapted to behave in these settings is left for future work.

Analysis of plan order. The parallel structure expressible by the SMC representation offers the flexibility required to encode plans that are executed concurrently or in multiple orders, as in partially order plans [23]. Developing mathematical and computational methods for counting equivalent plan arrangements generated by a given SMC could deepen our understanding of the combinatorial complexity of free arrangements for a particular plan and sequence of plans [199].

8.3.2 Hierarchical Plan Transfer

Hierarchical plan transfer has shown promise, but further research is needed to expand its scope and applicability.

Applications to Hierarchical Task Networks (HTNs). HTNs provides added flexible to the symbolic planning representations like PDDL, as discussed in Section 2.3.3. One promising direction is the use of the proposed hierarchical transfer methods to validate HTNs, where decomposing complex tasks into subtasks may benefit from the formal guarantees, such as plan equivalence and consistency checks, provided by SMCs and monoidal functors.

Diverse case studies for hierarchical transfer. In Section 5.3, we demonstrate hierarchical transfer on a case study related to machine-tending using single arm robot manipulators. Identifying and conducting additional case studies across varied domains will be critical to assessing the scalability and practical applicability of hierarchical transfer. These studies will help refine the methodology for broader and more complex use cases.

8.3.3 Structured Knowledge in Planning

Structured knowledge plays a vital role in analogical transfer, but additional research is needed to maximize its potential especially in general use cases.

Exploring the utility of hierarchical actions. Positive applicaton conditions (PACs) and negative application conditions (NACs) were introduced to allow the factoring of preconditions (refer to Section 6.4.3.1). An area of future work includes investigating utility of PACs and NACs in constructing hierarchical actions. The ability to progressively layer preconditions presents an opportunity to support the structured relaxation of planning actions which may be useful in planning in light of uncertainty [200, 201].

Compositional domain ontologies. The long tradition of logic-based representations of knowledge [202] have led to development of unifying base ontologies *and* their extensions [203, 204]. Exploring the use of categorical constructs like operads [205] to support compositional domain ontologies, namely schemas within schemas, is a useful direction for future work. Such work would significantly enhance the practicality of this approach by supporting a nested structure and organization of complex planning domain ontologies.

Grounding Large Language Model (LLM)-based task planning. The rise of LLMs [206] has sparked significant interest in LLM-driven task planning and reasoning, yet researchers are increasingly grappling with fundamental challenges such as long-horizon inconsistencies, error propagation, and hallucinations [207–209]. Chapter 6 presents a structured planning approach that integrates formal ontologies and symbolic representations, potentially offering a robust foundation for grounding LLM-based planning in well-defined world models. A promising future direction is to investigate whether this approach can serve as a semantic bridge between natural language

instructions and formal task representations.

8.3.4 Analogical Plan Transfer

The work on analogical transfer introduces new possibilities, but several areas merit further investigation.

Deeper analysis of conjunctive queries. Conducting a deeper mathematical analysis of conjunctive queries, as discussed in Section 7.2.3, could clarify when these data migrations are functorial and guarantee valid plan transfers. By understanding the conditions under which queries yield valid plans, we can improve the expressiveness of translation functors and expand the range of applications of this analogical transfer approach.

Automatic ontology alignment for translation functors. Analogical transfer currently depends on manually identifying a match for each concept and relationship from one domain ontology to another to form translation functors. Future research could explore how existing ontology alignment techniques [99, 165, 210–212] can be leveraged to automatically infer domain-to-domain translation functors to ensure the scalability of analogical transfers.

Formal Validation of Attributes in Real-World Plan Transfer. While this dissertation provides theoretical guarantees for transferred goal satisfaction using delta migration (refer to Section 7.6), an open question remains regarding the validity of transferred plans when non-combinatorial elements, such as boolean, numeric, and continuous state variables are involved. A formal theorem and proof are needed to establish whether a plan that is valid in the source domain remains valid in the target domain under real-world planning constraints.

8.3.5 Integrated Sequential-Hierarchical-Analogical (SHA) Transfer Framework

A highly promising direction for future research is the development of a unified framework that integrates sequential, hierarchical, and analogical transfer methods into a cohesive system. While this dissertation explores each method individually, their shared foundation in category theory suggests that they are theoretically interoperable. This integrated approach could address fundamental limitations in current task planning systems, such as the inability to easily transfer plans across domains, scale to new tasks, or handle scenarios with highly structured knowledge. Future work in this area involves an interdisciplinary blend of mathematical, AI, and software advancements.

Understanding mathematical interaction between SMCs and categorical rewriting. SMCs provide a well-structured algebraic foundation for modeling sequential and hierarchical compositions in planning, while categorical rewriting—particularly double-pushout (DPO) rewriting—offers a principled way to modify structured knowledge representations in planning domains. Despite their individual strengths, a formal mathematical theory that unifies these two approaches remains underexplored. One research direction involves formalizing the relationship between SMCs and rewriting systems to capture the compositional structure of planning operators while preserving semantic constraints imposed by rewriting rules. This may be addressed by investigating higher-categorical formulations of planning, potentially relating to bicategories of relations [213], where 1-morphisms represent sequential plans and 2-morphisms capture state transformations via rewriting rules. Alternatively, double categories [214] may offer a stricter framework for managing the interplay between sequential composition and structured knowledge.

Exploring analogical chaining in AI planning. In complex planning problems, it is common for analogies to break down at certain stages, requiring a shift in perspective to continue problem-solving effectively. Analogical chaining extends the concept of analogical transfer by allowing a planning system to sequentially apply multiple analogies, using one analogy to reach an intermediate subgoal before transitioning to another analogy that better suits the remaining problem structure. For example, a planner might initially leverage the structured constraints of the Tower of Hanoi domain to guide early-stage planning before switching to a Blocksworld-based reasoning framework to refine object manipulations, as discussed in Section 7.8.1. This approach has the potential to improve planning efficiency by dynamically adapting the problem representation to different phases of the task. Future work should explore formal mechanisms for detecting when an analogy is no longer beneficial and systematically selecting the next best analogy to maintain planning coherence.

A software system for integrated SHA transfer and categorical planning. A significant next step is the development of a knowledge-based planning system that operationalizes this integrated framework. A software system that supports planning and plan transfer using these category theory-based framework would make these approaches accessible to practitioners in AI and robotics. Such a system would would automate plan transfer along sequential, hierarchical, and analogical dimensions, while incorporating structured world knowledge and automated ontology alignment. This system could potentially be used in real-world scenarios—from industrial automation and robotic manufacturing to disaster response and autonomous exploration—where robots need to adapt to complex and changing environments.



(a) Privacy in Manufacturing [215]



(b) Interoperability in Mission-Planning [216]

Figure 8.1: Example potential application areas for robot plan transfer

8.3.5.1 Potential Applications

The category-theoretic framework developed in this dissertation offers a fresh perspective on longstanding challenges in robotic programming and task planning. This framework has the potential to address critical issues like privacy, explainability, and interoperability which can lead to the safe and ethical deployment of AI systems in both embodied and non-embodied settings. In this section, we envision two potential applications of the integrated SHA framework for planning.

Privacy and Explainability in Manufacturing Settings. As AI systems tackle increasingly complex tasks, they often rely on detailed, sensitive information about their environments. This introduces potential risks when task plans are shared or transferred across domains or organizations, as proprietary processes or strategic layouts may be inadvertently exposed. In industrial manufacturing, for instance, companies may wish to leverage external planning systems without revealing sensitive aspects of their workflows [185, 217]. Analogical task plan transfer, as proposed here, offers a solution by enabling organizations to adapt task plans from other domains without compromising confidential knowledge.

Additionally, the inherent opaqueness of many data-driven planning systems—particularly those based on reinforcement learning—poses challenges around transparency and explainability.

These issues are especially relevant for stakeholders such as regulators, operators, and the public, who need to understand how and why certain decisions are made. By incorporating formal semantics and structured representations into task planning, this framework builds trust by making AI decision-making processes more transparent and explainable. Ultimately, this approach brings AI systems closer to being trusted and reliable partners in high-stakes applications.

Interoperability within Mission-Planning. Effective task plan transfer is crucial in disaster response scenarios, where human teams and robots must work seamlessly in unstructured environments [218]. By enhancing interoperability, task plan transfer enables robots and humans to execute tasks efficiently and adaptively, increasing the overall effectiveness of disaster response. Sequential transfer, for example, ensures that critical tasks like search and rescue operations follow the correct sequence—assessing damage, locating survivors, and providing medical aid [219]. Hierarchical transfer enables integration of local teams' task plans with higher-level coordination efforts, aligning ground-level actions with overarching mission goals. Analogical transfer allows for the adaptation of task plans across various disaster scenarios, from urban search and rescue missions to natural disasters like wildfires or floods, while preserving essential elements such as resource allocation and personnel coordination [219].

Leveraging these three forms of task transfer together makes it possible to reuse, adapt, and share task plans more effectively across mission scenarios. This framework's potential to improve disaster response efficiency could optimize resource allocations and allow for more efficient planning in critical, life-saving scenarios.

8.4 Concluding Remarks

Robots are employed in countless settings around the world, each tailored to their specific platforms, work cells, materials, and tasks. Every one of these finely-tuned solutions introduces unique variations that rarely scale beyond a single robot or, at best, robots from the same vendor. One can blur their eyes and see shared behaviors across these solutions, but most robotic plans are re-computed because of the lack of consistent methods for transferring common behaviors.

The potential for robots to become ubiquitous in daily life hinges on their ability to transcend rigid, one-off solutions. Achieving this will require a unified, flexible framework that facilitates the understanding and verification of action dependencies, manages extensive networks of world knowledge, and systematically transfers plans across various planning contexts. Such a framework demands a mathematical language capable of describing interdependent processes and highly-connected knowledge. In this dissertation, we demonstrated the viability of category theory as a promising candidate uniquely suited to address these challenges.

By pioneering a category-theoretic plan transfer framework, this research lays foundational groundwork for the next generation of modular robotic programming, secure plan transfer, and knowledge-based planning. It sets the stage for new advancements in automation science and intelligent systems, bringing us closer to a future where robots are intelligent, adaptive, and indispensable partners in the quest for greater human innovation and discovery.

Appendix A: Supplementary Material

A.1 Forward Planning using C-sets and DPO Rewriting

Algorithm 4 provides an example of a depth-first search algorithm can be implemented using the categorical planning representation language introduced in Chapter 6.

A.2 Schema Categories

In this section, we provide full code implementations for the schema categories and translation functors introduced in Chapter 7. We have verified compatibility of this code with the following Julia packages and versions: ACSets.jl (0.2.20), AlgebraicRewriting.jl (0.3.5), Catlab.jl (0.16.13), and DataMigrations.jl (0.0.3).

A.2.1 Assembly

Here, we provide a specification of the schema category for the **Assembly** domain using AlgebraicJulia syntax.

```
@present OntAssembly(FreeSchema) begin
  # Types
  Part::Ob
  Assembly::Ob
  Location::Ob
  PartType::Ob

  # Predicates
  IsPartOf::Hom(Part, Assembly)
  PIsLocatedAt::Hom(Part, Location)
  HasType::Hom(Part, PartType)
  AIsLocatedAt::Hom(Assembly, Location)
```

Algorithm 4: ForwardPlan

Input: C-set `world`, C-set `goal`, List of action spans `actions`, Action Limits `action_limits`, List of action and match pairs `plan`

Output: List of action and match pairs `plan`

```
Exit criteria: goal is reached if there is a monic morphism from goal to world;
if homomorphism(goal, world, monic=True) ≠ nothing then
    print "Goal reached.";
    return plan;
Find applicable actions for the current world state;
applicable_actions ← find_applicable_actions(world, actions);
Backtrack criteria: no applicable actions
if applicable_actions is empty then
    throw "No applicable actions found! Terminate path...";
Iterate over all applicable actions
for action ← applicable_actions do
    Backtrack criteria: action applied too many times
    if action_usage[action.key] ≥ action_limits[action.key] then
        "Abort path. Action used too many times.";
        continue;
    else
        action_usage[action.key] ← 1
    Save current world state for backtracking
    prev_world ← world;
    selected_action ← actions⌊action.key⌋;
    Apply the selected action to the current world state
    try
        world ← DPORewrite(selected_action, world);
    catch
        throw "Could not apply action!" ;
    Record the action and corresponding match morphism in the current plan
    Append (selected_action, action.match) to plan;
    Recursively call ForwardPlan() with updated world state
    try
        plan ← ForwardPlan(world, goal, actions, action_limits, plan);
        return plan;
    catch
        Backtrack to previous world state if path fails
        world ← prev_world;
        Remove last action from plan;
        throw "Path failed. Try a different path.";
If no valid plan is found, report failure
"No plan found."
```

```
  # Relations
  IsAttachedTo::Ob
  IsAttachedToL::Hom(IsAttachedTo, Part)
  IsAttachedToR::Hom(IsAttachedTo, Part)

  # Constraints
  compose(IsPartOf, AIsLocatedAt) == PIsLocatedAt
end
```

A.2.2 EmergencyComms

Here we provide a specification of the schema category for the **EmergencyComms** domain using AlgebraicJulia syntax.

```
@present OntEmergencyComms(FreeSchema) begin
  # Types
  NetworkComponent::Ob
  CommunicationNetwork::Ob
  Area::Ob
  ComponentType::Ob

  # Predicates
  IsPartOf::Hom(NetworkComponent, CommunicationNetwork)
  ComponentIsServingArea::Hom(NetworkComponent, Area)
  HasType::Hom(NetworkComponent, ComponentType)
  NetworkIsServingArea::Hom(CommunicationNetwork, Area)

  # Relations
  IsConnectedTo::Ob
  IsConnectedToL::Hom(IsConnectedTo, NetworkComponent)
  IsConnectedToR::Hom(IsConnectedTo, NetworkComponent)

  # Constraint
  compose(IsPartOf, NetworkIsServingArea) == ComponentIsServingArea
end
```

A.2.3 ColorBlocksworld

Here we provide a specification of the schema category for the **ColorBlocksworld** domain using AlgebraicJulia syntax. In this example, the attributes are formed by spans. For example:

```
hasColor::Attr(HasColor, Color)
hasColorDom::Hom(HasColor, Block)
```

These spans are built such that an object representing the attribute, i.e. `HasColor::Ob`,

maps to an attribute type, i.e. `Color::AttrType`, and the object for which the attribute is for, i.e. `Block::Ob`. This is to allow for attribute values to be updated without discarding the object, namely the block, itself.

```
@present OntColorBlocksworld(FreeSchema) begin
  # Types
  Block::Ob
  Empty::Ob
  Gripper::Ob
  HasColor::Ob
  IsClear::Ob
  IsOnTable::Ob
  HasLabel::Ob
  Color::AttrType
  Clear::AttrType
  OnTable::AttrType
  Label::AttrType

  # Predicates
  isEmpty::Hom(Empty, Block)
  isHolding::Hom(Gripper, Block)

  hasColor::Attr(HasColor, Color)
  hasColorDom::Hom(HasColor, Block)

  isClear::Attr(IsClear, Clear)
  isClearDom::Hom(IsClear, Block)

  isOnTable::Attr(IsOnTable, OnTable)
  isOnTableDom::Hom(IsOnTable, Block)

  hasLabel::Attr(HasLabel, Label)
  hasLabelDom::Hom(HasLabel, Block)

  # Relations
  On::Ob
  on_l::Hom(On, Block)
  on_r::Hom(On, Block)
  MyBool::AttrType
  MyString::AttrType
  MyFloat::AttrType
end
```

A.2.4 Kitchenworld

Here we provide a specification of the schema category for the **Kitchenworld** domain using AlgebraicJulia syntax.

```
@present OntKitchenworld(FreeSchema) begin
  # Types
  Object::Ob
  Receptacle::Ob
  Empty::Ob
  Gripper::Ob
  BreadSliced::Ob
  LettuceSliced::Ob
  TomatoSliced::Ob
  Plate::Ob
  Knife::Ob

  # Predicates
  isHolding::Hom(Gripper, Object)
  isEmpty::Hom(Empty, Object)
  Temperature::AttrType
  Mass::AttrType
  SalientMaterials::AttrType

  breadSlicedIsObject::Hom(BreadSliced, Object)
  breadSlicedIsReceptacle::Hom(BreadSliced, Receptacle)
  breadSlicedHasTemperature::Attr(BreadSliced, Temperature)
  breadSlicedHasMass::Attr(BreadSliced, Mass)
  breadSlicedHasSalientMaterials::Attr(BreadSliced, SalientMaterials)

  lettuceSlicedIsObject::Hom(LettuceSliced, Object)
  lettuceSlicedIsReceptacle::Hom(LettuceSliced, Receptacle)
  lettuceSlicedHasTemperature::Attr(LettuceSliced, Temperature)
  lettuceSlicedHasMass::Attr(LettuceSliced, Mass)
  lettuceSlicedHasSalientMaterials::Attr(LettuceSliced, SalientMaterials)

  tomatoSlicedIsObject::Hom(TomatoSliced, Object)
  tomatoSlicedIsReceptacle::Hom(TomatoSliced, Receptacle)
  tomatoSlicedHasTemperature::Attr(TomatoSliced, Temperature)
  tomatoSlicedHasMass::Attr(TomatoSliced, Mass)
  tomatoSlicedHasSalientMaterials::Attr(TomatoSliced, SalientMaterials)

  plateIsObject::Hom(Plate, Object)
  plateIsReceptacle::Hom(Plate, Receptacle)
  plateHasTemperature::Attr(Plate, Temperature)
  plateHasMass::Attr(Plate, Mass)
  plateHasSalientMaterials::Attr(Plate, SalientMaterials)

  knifeIsObject::Hom(Knife, Object)
  knifeHasTemperature::Attr(Knife, Temperature)
  knifeHasMass::Attr(Knife, Mass)
  knifeHasSalientMaterials::Attr(Knife, SalientMaterials)

  # Relations
  InReceptacle::Ob
  inReceptacle_l::Hom(InReceptacle, Object)
  inReceptacle_r::Hom(InReceptacle, Receptacle)

  MyBool::AttrType
```

```
  MyString::AttrType
  MyFloat::AttrType
end
```

A.3 Translation Functors

In this section, we provide the implementations for the translation functors from EmergencyComms to Assembly and from Kitchenworld to ColorBlocksworld.

A.3.1 Translation Functor from EmergencyComms to Assembly

Here we provide the implementation for the translation functor EmergencyComms to Assembly. Because this is leveraging a direct analogy (see Chapter 7), all the types and predicates of EmergencyComms are mapped to the types and predicates of Assembly one-to-one.

```
F = @migration OntEmergencyComms OntAssembly begin
  # Types
  NetworkComponent => Part
  CommunicationnNetwork => Assembly
  Area => Location
  ComponentType => PartType

  # Relations
  IsConnectedTo => IsAttachedTo
  IsConnectedToL => IsAttachedToL
  IsConnectedToR => IsAttachedToR

  # Part
  IsPartOf => IsPartOf
  ComponentIsServingArea => PIsLocatedAt
  HasType => HasType

  # Assembly
  NetworkIsServingArea => AIsLocatedAt
end
```

A.3.2 Translation Functor from Kitchenworld to ColorBlocksworld

Here we provide the implementation for the translation functor from Kitchenworld to ColorBlocksworld. Because this is leveraging a structural analogy (see Chapter 7), all the types and predicates of

Kitchenworld are mapped to diagrams of types and predicates of ColorBlocksworld. For instance, notice that `BreadSliced` is mapped to a conjunctive query, denoted by `@join`, that queries for purple blocks.

```
G = @migration OntKitchenworld OntColorBlocksworld begin
  # Parent Types
  Object => Block
  Receptacle => Block
  Empty => Empty
  Gripper => Gripper

  # Robot
  isHolding => isHolding
  isEmpty => isEmpty

  # Properties
  Temperature => MyFloat
  Mass => MyFloat
  SalientMaterials => MyString

  # Relations
  InReceptacle => On
  inReceptacle_l => on_l
  inReceptacle_r => on_r

  # Ingredients
  # BreadSlice
  BreadSliced => @join begin
      block::Block
      color::HasColor
      C::Color
      (hc:color → C)::hasColor
      (hcd:color → block)::hasColorDom
      (c:color → C)::(color -> "purple")
  end
  breadSlicedIsObject => block
  breadSlicedIsReceptacle => block
  breadSlicedHasTemperature => (block > (x -> 0.0))
  breadSlicedHasMass => (block > (x -> 0.0))
  breadSlicedHasSalientMaterials => (block |> (x -> "Unknown"))

  # LettuceSlice
  LettuceSliced => @join begin
      block::Block
      color::HasColor
      C::Color
      (hc:color → C)::hasColor
      (hcd:color → block)::hasColorDom
      (c:color → C)::(color -> "green")
  end
  lettuceSlicedIsObject => block
```

```
  lettuceSlicedIsReceptacle => block
  lettuceSlicedHasTemperature => (block > (x -> 0.0))
  lettuceSlicedHasMass => (block > (x -> 0.0))
  lettuceSlicedHasSalientMaterials => (block |> (x -> "Unknown"))

  # TomatoSlice
  TomatoSliced => @join begin
      block::Block
      color::HasColor
      C::Color
      (hc:color → C)::hasColor
      (hcd:color → block)::hasColorDom
      (c:color → C)::(color -> "red")
  end
  tomatoSlicedIsObject => block
  tomatoSlicedIsReceptacle => block
  tomatoSlicedHasTemperature => (block > (x -> 0.0))
  tomatoSlicedHasMass => (block > (x -> 0.0))
  tomatoSlicedHasSalientMaterials => (block |> (x -> "Unknown"))

  # Plate
  Plate => @join begin
      block::Block
      color::HasColor
      C::Color
      (hc:color → C)::hasColor
      (hcd:color → block)::hasColorDom
      (c:color → C)::(color -> "yellow")
  end
  plateIsObject => block
  plateIsReceptacle => block
  plateHasTemperature => (block > (x -> 0.0))
  plateHasMass => (block > (x -> 0.0))
  plateHasSalientMaterials => (block |> (x -> "Unknown"))

  # Knife
  Knife => @join begin
      block::Block
      color::HasColor
      C::Color
      (hc:color → C)::hasColor
      (hcd:color → block)::hasColorDom
      (c:color → C)::(color -> "blue")
  end
  knifeIsObject => block
  knifeHasTemperature => (block > (x -> 0.0))
  knifeHasMass => (block > (x -> 0.0))
  knifeHasSalientMaterials => (block |> (x -> "Unknown"))

  MyFloat => MyFloat
  MyString => MyString
  MyBool => MyBool
end
```

Appendix B: Scholarly Contributions

B.1 Papers

1. **Aguinaldo A.**, Patterson E., Regli W. Automating Transfer of Robot Task Plans using Functorial Data Migrations. IEEE Transactions on Automation Science and Engineering (Under Revision).
2. **Aguinaldo A.**, Patterson E., Fairbanks, J., Regli W., Ruiz J. A Categorical Representation Language and Computational System for Knowledge-Based Planning. 2023 AAAI Fall Symposium on Unifying Representations for Robot Application Development (Best Paper). 2023.
3. **Aguinaldo A.**, Regli W. Modeling traceability, change information, and synthesis in autonomous system design using symmetric delta lenses. ICRA Compositional Robotics Workshop 2022.
4. **Aguinaldo A**., Regli W. Encoding Compositionality in Classical Planning Solutions. IJCAI Workshop on Generalization in Planning 2021.
5. **Aguinaldo A.**, Regli W. A Graphical Model-Based Representation for Classical AI Plans using Category Theory. ICAPS 2021 Workshop on Explainable AI Planning.
6. **Aguinaldo A.**, Bunker J., Pollard B., Shukla A., Canedo A., Quiros G., Regli W. RoboCat: A Category Theoretic Framework for Robotic Interoperability Using Goal-Oriented Programming. IEEE Transactions on Automation Science and Engineering, 2021. doi: 10.1109/TASE.2021.3094055.

B.2 Talks

1. *"Analogical Plan Transfer in Robotics using Functorial Data Migrations"* at AMS Special Session on Applied Category Theory, II in 2025
2. *"'What's the plan?', asked my robot"* at JuliaCon in 2024
3. *"A Category Theoretic Approach to Planning in a Complex World"* at Microsoft Future Leaders in Robotics and AI Seminar Series in 2023
4. *"Contextual affordances in context-aware autonomous systems"* at AMS JMM 2023 Special Session on Applied Category Theory (a Mathematics Research Communities session)
5. *"Contextual affordances in context-aware autonomous systems"* at NIST Compositional Structures for Systems Engineering and Design Workshop 2022
6. *"Category theory for automated planning and program compilation in robotics"* at Topos Berkeley Seminar 2022
7. *"Applications of category theory to automated planning and program compilation in robotics"* at Xerox PARC Design Seminar in 2022
8. *"Modeling traceability, change information, and synthesis in autonomous system design using symmetric delta lenses"* at International Conference for Robotics and Automation (ICRA) 2022 Compositional Robotics Workshop in 2022

9. *"Encoding Compositionality in Classical Planning Solutions"* at International Joint Conference for Artifical Intelligence (IJCAI): Workshop on Generalization in Planning in 2021
10. *"A Graphical Model-Based Representation for Classical AI Plans using Category Theory"* at International Conference for Automated Planning and Scheduling (ICAPS): Explainable AI Planning Workshop in 2021
11. *"Polynomial Functors in Catlab"* at the Applied Category Theory Conference in 2021
12. *"Diary of a software engineering using categories"* at Topos Institute: Berkeley Seminar and Lógicos em Quarentena (LQ) Seminar in 2021
13. *"Category Theory for Software Modeling and Design"* at Hunter College Applied Mathematics Seminar in 2020

B.3 Community Service

- ICAPS 2025 (International Conference on Automated Planning and Scheduling) Paper Reviewer
- Co-author of *Relational Thinking: From Abstractions to Applications* (2024) digital textbook
- Open-source contributor to AlgebraicJulia
- Co-organizer of Compositional Robotics: Mathematics and Tools (ICRA), 2023
- Co-organizer of The Adjoint School, 2022-2023
- Student of The Adjoint School, 2021

Bibliography

[1] Bob Malone. George devol: A life devoted to invention, and robots. *IEEE Spectrum*, September 2011. Accessed: 2024-08-16.

[2] Tudor B. Ionescu. Leveraging Graphical User Interface Automation for Generic Robot Programming. *Robotics*, 2021, 2020.

[3] Anthony Cowley, Luiz Chaimowicz, and Camillo J. Taylor. Design minimalism in robotics programming. *International Journal of Advanced Robotic Systems*, 3(1):031–036, 2006.

[4] Dana Nau, Okhtay Ilghami, Ugur Kuter, J William Murdock, Dan Wu, and Fusun Yaman. SHOP2: An HTN Planning System. *Journal of Artificial Intelligence Research*, 20:379–404, 2003.

[5] Dedre Gentner and L. Smith. *Analogical Reasoning*, volume 1. Elsevier Inc., 2 edition, 2012.

[6] Luca Spalzzi. A survey on case-based planning. *Artificial Intelligence Review*, 16(1):3–36, 2001.

[7] Lisa Heuss, Daniel Gebauer, and Gunther Reinhart. Concept for the automated adaption of abstract planning domains for specific application cases in skills-based industrial robotics. *Journal of Intelligent Manufacturing*, 2023.

[8] David I Spivak and Robert E Kent. Ologs: a categorical framework for knowledge representation. *PLOS ONE*, 2011.

[9] D. E. Wilkins and M. DesJardins. A call for knowledge-based planning. *AI Magazine*, 22(1):99–115, 2001.

[10] Michael Beetz, Ulrich Klank, Ingo Kresse, Alexis Maldonado, Lorenz Mösenlechner, Dejan Pangercic, Thomas Ruhr, and Moritz Tenorth. Robotic roommates making pancakes. *IEEE-RAS International Conference on Humanoid Robots*, pages 529–536, 2011.

[11] Josip Tomo Licardo, Mihael Domjan, and Tihomir Orehovački. Intelligent Robotics—A Systematic Review of Emerging Technologies and Trends. *Electronics (Switzerland)*, 13(3), 2024.

[12] Caelan Reed Garrett, Tomás Lozano-Pérez, and Leslie Pack Kaelbling. Pddlstream: Integrating symbolic planners and blackbox samplers via optimistic adaptive planning. In J. Christopher Beck, Olivier Buffet, Jörg Hoffmann, Erez Karpas, and Shirin Sohrabi, editors, *Proceedings of the Thirtieth International Conference on Automated Planning and Scheduling, Nancy, France, October 26-30, 2020*, pages 440–448. AAAI Press, 2020.

[13] Josh Zapf, Marco Roveri, Francisco Martin, and Juan Carlos Manzanares. Constructing Behavior Trees from Temporal Plans for Robotic Applications. *arXiv*, 2024.

[14] Tara Estlin, Daniel Gaines, Caroline Chouinard, Rebecca Castano, Benjamin Borastein, Michele Judd, Issa Nesnas, and Robert Anderson. Increased mars rover autonomy using AI planning, scheduling and execution. *Proceedings - IEEE International Conference on Robotics and Automation*, pages 4911–4918, 2007.

[15] Masoumeh Mansouri, Federico Pecora, and Peter Schüller. Combining Task and Motion Planning: Challenges and Guidelines. *Frontiers in Robotics and AI*, 8(May):1–12, 2021.

[16] Fabien Lagriffoul, Neil T. Dantam, Caelan Garrett, Aliakbar Akbari, Siddharth Srivastava, and Lydia E. Kavraki. Platform-Independent Benchmarks for Task and Motion Planning. *IEEE Robotics and Automation Letters*, 3(4):3765–3772, 2018.

[17] Huihui Guo, Fan Wu, Yunchuan Qin, Ruihui Li, Keqin Li, and Kenli Li. Recent trends in task and motion planning for robotics: A Survey. *ACM Computing Surveys*, 55(13), 2023.

[18] Caelan Reed Garrett, Rohan Chitnis, Rachel Holladay, Beomjoon Kim, Tom Silver, Leslie Pack Kaelbling, and Tomas Lozano-Perez. Integrated Task and Motion Planning. *Annual Review of Control, Robotics, and Autonomous Systems*, 4:265–293, 2021.

[19] Leslie Pack Kaelbling and Tomás Lozano-PÉrez. Hierarchical Task and Motion Planning in the Now. *2011 IEEE International Conference on Robotics and Automation*, 2011.

[20] Noemie Jaquier, Michael C. Welle, Andrej Gams, Kunpeng Yao, Bernardo Fichera, Aude Billard, Ales Ude, Tamim Asfour, and Danica Kragic. Transfer Learning in Robotics: An Upcoming Breakthrough? A Review of Promises and Challenges. *SAGE*, ...(...), 2023.

[21] Christian Bitter, Jannik Peters, Hasan Tercan, and Tobias Meisen. Industrial Cross-Robot Transfer Learning. *Procedia CIRP*, 120:1297–1302, 2023.

[22] Alejandro Barredo Arrieta, Natalia Díaz-Rodríguez, Javier Del Ser, Adrien Bennetot, Siham Tabik, Alberto Barbado, Salvador Garcia, Sergio Gil-Lopez, Daniel Molina, Richard Benjamins, Raja Chatila, and Francisco Herrera. Explainable Artificial Intelligence (XAI): Concepts, taxonomies, opportunities and challenges toward responsible AI. *Information Fusion*, 58(Dl):82–115, 2020.

[23] Malik Ghallab, Dana Nau, and Paolo Traverso. *Automated Planning: Theory and Practice*. Morgan Kaufmann Publishers, 2004.

[24] Tomás Lozano-PÉrez. Robot Programming. *Proceedings of the IEEE*, 71(7):821–841, 1983.

[25] Simon Bøgh, Oluf Skov Nielsen, Mikkel Rath Pedersen, Volker Krüger, and Ole Madsen. Does your robot have skills. In *Proceedings of the 43rd International Symposium on Robotics*, 2012.

[26] D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Commun. ACM*, 15(12):1053–1058, December 1972.

[27] Kutluhan Erol, James Hendler, and Dana S. Nau. UMCP: A Sound and Complete Procedure for Hierarchical Task-network Planning. *Aips*, 94:249–254, 1994.

[28] Eckhard Freund, Bernd Lüdemann-Ravit, Oliver Stern, and Thorsten Koch. Creating the architecture of a translator framework for robot programming languages. *Proceedings - IEEE International Conference on Robotics and Automation*, 1:187–192, 2001.

[29] Janet L. Kolodner. Reconstructive memory: A computer model. *Cognitive Science*, 7(4):281–328, 1983.

[30] Samuel Eilenberg and Saunders MacLane. General Theory of Natural Equivalences. *Transactions of the American Mathematical Society*, 58(2):231, 1945.

[31] Elias Zafiris. Complex Systems From the Perspective of Category Theory: I. Functioning of the Adjunction Concept. *Axiomathes*, 15(1):147–158, 2005.

[32] Mieczyslaw M. Kokar, Kenneth Baclawski, and Hongge Gao. Category theory-based synthesis of a higher-level fusion algorithm: An example. *2006 9th International Conference on Information Fusion, FUSION*, 2006.

[33] Samson Abramsky and Bob Coecke. Categorical Quantum Mechanics. *Handbook of Quantum Logic and Quantum Structures vol II*, pages 1–63, 2008.

[34] Bob Coecke and Aleks Kissinger. 286categorical quantum mechanics i: Causal quantum processes. In *Categories for the Working Philosopher*. Oxford University Press, 11 2017.

[35] Zinovy Diskin. Algebra of bidirectional model synchronization. *Technical Report CSRG573 Department of Compuitng Science University of Toronto*, 573pdf([7), 2008.

[36] Zinovy Diskin, Yingfei Xiong, and Krzysztof Czarnecki. From State- to Delta-Based Bidirectional Model Transformations: the Asymmetric Case. *The Journal of Object Technology*, 10:6:1, 2011.

[37] Zinovy Diskin and Tom Maibaum. Category theory and model-driven engineering: From formal semantics to design patterns and beyond. *Electronic Proceedings in Theoretical Computer Science, EPTCS*, 93:1–21, 2012.

[38] Eric Kolve, Roozbeh Mottaghi, Winson Han, Eli VanderBilt, Luca Weihs, Alvaro Herrasti, Matt Deitke, Kiana Ehsani, Daniel Gordon, Yuke Zhu, Aniruddha Kembhavi, Abhinav Gupta, and Ali Farhadi. AI2-THOR: An Interactive 3D Environment for Visual AI. *arXiv*, pages 1–12, 2017.

[39] André Joyal and Ross Street. The geometry of tensor calculus, i. *Advances in Mathematics*, 88(1):55 – 112, 1991.

[40] Peter Selinger. A survey of graphical languages for monoidal categories. *Lecture Notes in Physics*, 813:1–63, 2010.

[41] Evan Patterson, Owen Lynch, and James Fairbanks. Categorical Data Structures for Technical Computing. *Compositionality*, 4(5):1–27, 2021.

[42] Kristopher Brown, Evan Patterson, Tyler Hanks, and James Fairbanks. Computational category-theoretic rewriting. *Journal of Logical and Algebraic Methods in Programming*, 134:100888, 2023.

[43] David I. Spivak and Robert E. Kent. Ologs: A categorical framework for knowledge representation. *PLoS ONE*, 7(1), 2012.

[44] George C. Devol. Programmed article transfer, 1954.

[45] Richard M Murray, D Curtis Deno, Kristofer S J Pister, and S Shankar Sastry. Control Primitives for Robot Systems. *IEEE Transactions on Systems, Man, and Cybernetics*, 22(1), 1992.

[46] Paul J Koch, Marike K Van Amstel, D Patrycja, Moritz A Thormann, Adrian J Tetzlaff, Simon Bøgh, and Dimitrios Chrysostomou. A Skill-based Robot Co-worker for Industrial Maintenance Tasks. *Procedia Manufacturing*, 11(June):83–90, 2017.

[47] Casper Schou, Rasmus Skovgaard Andersen, Dimitrios Chrysostomou, Simon Bøgh, and Ole Madsen. Skill-based instruction of collaborative robots in industrial settings. *Robotics and Computer Integrated Manufacturing*, 53(March):72–80, 2018.

[48] Luis Miguel Vieira Da Silva, Aljosha Köcher, Philip Topalis, and Alexander Fay. A Python Framework for Robot Skill Development and Automated Generation of Semantic Descriptions. *IEEE International Conference on Emerging Technologies and Factory Automation, ETFA*, 2023-Septe, 2023.

[49] Franz Steinmetz and Roman Weitschat. Skill Parametrization Approaches and Skill Architecture for Human-Robot Interaction. *2016 IEEE International Conference on Automation Science and Engineering (CASE)*, pages 280–285, 2016.

[50] Rasmus S Andersen, Casper Schou, Jens S Damgaard, and Ole Madsen. Using a Flexible Skill-Based Approach to Recognize Objects in Industrial Scenarios. In *Proceedings of ISR 2016: 47st International Symposium on Robotics*, volume 2016, pages 399–406, 2016.

[51] Maj Stenmark and Elin A Topp. From Demonstrations to Skills for High-Level Programming of Industrial Robots. In *2016 AAAI Fall Symposium Series: Artificial Intelligence for Human-Robot Interaction*, pages 75–78, 2016.

[52] Pei-chi Huang, Yi-hsuan Hsieh, and Aloysius K Mok. A Skill-Based Programming System for Robotic Furniture Assembly. *2018 IEEE 16th International Conference on Industrial Informatics (INDIN)*, pages 355–361, 2018.

[53] Francesco Rovida, Matthew Crosby, Dirk Holz, Athanasios S Polydoros, and Bjarne Großmann. SkiROS — A skill-based robot control platform on top of ROS SkiROS — A Skill-Based Robot Control Platform on Top of ROS. In *Robot Operating System (ROS)*, volume 707 of *Studies in Computational Intelligence*, pages 121–160. Springer, 2017.

[54] Mikkel Rath Pedersen, Lazaros Nalpantidis, Rasmus Skovgaard Andersen, Casper Schou, Simon Bøgh, Volker Krüger, and Ole Madsen. Robot skills for manufacturing: From concept to industrial deployment. *Robotics and Computer-Integrated Manufacturing*, 37:282–291, 2016.

[55] Dirk Holz, Angeliki Topalidou-kyniazopoulou, Francesco Rovida, Mikkel Rath Pedersen, Volker Kr, and Sven Behnke. A Skill-Based System for Object Perception and Manipulation for Automating Kitting Tasks. In *2015 IEEE 20th Conference on Emerging Technologies & Factory Automation (ETFA)*, 2015.

[56] Mikkel Rath Pedersen, Lazaros Nalpantidis, Aaron Bobick, and Volker Kr. On the Integration of Hardware-Abstracted Robot Skills for use in Industrial Scenarios. In *Conference:*

Second international workshop on Cognitive Robotics Systems: Replicating Human Actions and Activities, 2013.

[57] Timo Bänziger, Andreas Kunz, and Konrad Wegener. A library of skills and behaviors for smart mobile assistant robots in automotive assembly lines. In *Proceedings of the Companion of the 2017 ACM/IEEE International Conference on Human-Robot Interaction*, HRI '17, page 77–78, New York, NY, USA, 2017. Association for Computing Machinery.

[58] International Organization for Standardization. Automation systems and integration — numerical control of machines — program format and definitions of address words — part 1: Data format for positioning, line motion, and contouring control systems. Technical Report ISO 6983-1:2009, International Organization for Standardization, Geneva, Switzerland, 2009.

[59] Frederick Proctor, William Shackleford, Stephen Balakirsky, Zeid Kootbally, Thomas Kramer, and Craig Schlenoff. The Canonical Robot Command Language (CRCL). *Industrial Robot*, 43(5):495–502, 2016.

[60] Richard E. Fikes and Nils J. Nilsson. Strips: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, 1971.

[61] Malik Ghallab, Craig Knoblock, Drew McDermott, Ashwin Ram, Manuela Veloso, Daniel Weld, and David Wilkins. PDDL - The Planning Domain Definition Language (Version 1.2). Technical report, Yale Center for Computational Vision and Control, 1998.

[62] Susan Bonner and Kang G. Shin. A Comparative Study of Robot Languages. *Computer*, 1982.

[63] David Kortenkamp and Reid Simmons. Robotic Systems Architectures and Programming. *Springer Handbook of Robotics*, pages 187–206, 2008.

[64] Alwin Hoffmann, Andreas Angerer, Andreas Schierl, Michael Vistein, and Wolfgang Reif. Managing extensibility and maintainability of industrial robotics software. *2013 16th International Conference on Advanced Robotics, ICAR 2013*, 2013.

[65] Yang Liang. A Middleware Based Control Architecture for Modular Robot Systems Iunfurthemore system as a standardized service unit which provides services. In *2008 IEEE/ASME International Conference on Mechtronic and Embedded Systems and Applications*, pages 327–332, 2008.

[66] B. Kast, S. Albrecht, W. Feiten, and J. Zhang. Bridging the gap between semantics and control for industry 4.0 and autonomous production. In *2019 IEEE 15th International Conference on Automation Science and Engineering (CASE)*, pages 780–787, 2019.

[67] Brian C Williams and Pandurang Nayak. A Model-based Approach to Reactive Self-Configuring Systems. *AAAI-96*, pages 971–978, 1996.

[68] Janarbek Matai, Young-ho Suh, Hyuongsun Kim, Kang-woo Lee, and Hyun Kim. Integration Framework for Interoperability of Distributed and Heterogeneous Robot Middlewares. *2008 10th International Conference on Control, Automation, Robotics and Vision*, pages 2337–2343, 2008.

[69] Manas Pradhan, Alexander Tiderko, and Daniel Ota. Approach Towards Achieving Interoperability between Military Land Vehicle and Robotic Systems. *2017 International Conference on Military Communications and Information Systems (ICMCIS)*, pages 1–7, 2017.

[70] Zoran Salcic, Senior Member, Udayanto Dwi Atmojo, Heejong Park, Andrew Tzer-yeu Chen, Student Member, and Kevin I-kai Wang. A Framework for Designing Dynamic and Interoperable Automation and Robotics Systems. In *2017 IEEE 15th International Conference on Industrial Informatics (INDIN)*, volume 2, 2017.

[71] Jeffrey Wesley Wallace and Sara Jane Kambouris. A language, framework, and sdk for robotic communications, integration, and interoperability. In *2016 International Conference on Computational Science and Computational Intelligence (CSCI)*, pages 472–477, 2016.

[72] Yifeng Zhu, Jonathan Tremblay, Stan Birchfield, and Yuke Zhu. Hierarchical Planning for Long-Horizon Manipulation with Geometric and Symbolic Scene Graphs. *Proceedings - IEEE International Conference on Robotics and Automation*, 2021-May:6541–6548, 2021.

[73] Philippe Morere, Lionel Ott, and Fabio Ramos. Learning to Plan Hierarchically from Curriculum. *IEEE Robotics and Automation Letters*, 4(3):2815–2822, 2019.

[74] Dana Nau, Yue Cao, Amnon Lotem, and Hector Munoz-Avila. SHOP: Simple Hierarchical Ordered Planner. *IJCAI International Joint Conference on Artificial Intelligence*, pages 1–6, 1999.

[75] Dongyeop Kang and Ho-jin Choi. Hierarchical Planning through Operator and World Abstraction using ontology for home service robots. *2009 11th International Conference on Advanced Communication Technology*, pages 2209–2214, 2009.

[76] Ilche Georgievski and Marco Aiello. HTN planning: Overview, comparison, and beyond. *Artificial Intelligence*, 222:124–156, 2015.

[77] Dedre Gentner. Structure Mapping: A Theoretical Framework for Analogy. *Cognitive Science*, 7:155–170, 1983.

[78] Eckhard Freund and Bernd Luedemann-Ravit. A system to automate the generation of program variants for industrial robot applications. *IEEE International Conference on Intelligent Robots and Systems*, 2(October):1856–1861, 2002.

[79] João Pedro Souza, André Castro, Luís Rocha, Pedro Relvas, and Manuel F. Silva. Converting Robot Offline Programs to Native Code Using the AdaptPack Studio Translators. *19th IEEE International Conference on Autonomous Robot Systems and Competitions, ICARSC 2019*, pages 1–7, 2019.

[80] João Pedro Carvalho de Souza, André Luiz Castro, Luís F. Rocha, and Manuel F. Silva. AdaptPack studio translator: translating offline programming to real palletizing robots. *Industrial Robot*, 47(5):713–721, 2020.

[81] Jan Oliver Ringert, Alexander Roth, Bernhard Rumpe, and Andreas Wortmann. Code generator composition for model-driven engineering of robotics component & connector systems. *CEUR Workshop Proceedings*, 1319:66–77, 2014.

[82] Kai Adam, Katrin Olldobler, Bernhard Rumpe, and Andreas Wortmann. Modeling Robotics Software Architectures with Modular Model Transformations. *Journal of Software Engineering for Robotics*, 8(1):3–16, 2017.

[83] Qiliang Chen, Babak Heydari, and Mohsen Moghaddam. Leveraging task modularity in reinforcement learning for adaptable industry 4.0 automation. *Journal of Mechanical Design, Transactions of the ASME*, 143(7), 2021.

[84] Yuke Zhu, Roozbeh Mottaghi, Eric Kolve, Joseph J. Lim, Abhinav Gupta, Li Fei-Fei, and Ali Farhadi. Target-driven visual navigation in indoor scenes using deep reinforcement learning. *Proceedings - IEEE International Conference on Robotics and Automation*, pages 3357–3364, 2017.

[85] Junhyuk Oh, Satinder Singh, Honglak Lee, and Pushmeet Kohli. Zero-shot task generalization with multi-task deep reinforcement learning. In *Proceedings of the 34th International Conference on Machine Learning - Volume 70*, ICML'17, page 2661–2670. JMLR.org, 2017.

[86] Lingfeng Sun, Haichao Zhang, Wei Xu, and Masayoshi Tomizuka. Efficient Multi-Task and Transfer Reinforcement Learning With Parameter-Compositional Framework. *IEEE Robotics and Automation Letters*, 8(8):4569–4576, 2023.

[87] Wei Meng, Hao Ju, Tongxu Ai, Randy Gomez, Eric Nichols, and Guangliang Li. Transferring Meta-Policy From Simulation to Reality via Progressive Neural Network. *IEEE Robotics and Automation Letters*, 9(4):3696–3703, 2024.

[88] Subbarao Kambhampati and James A. Hendler. A validation-structure-based theory of plan modification and reuse. *Artificial Intelligence*, 55(2-3):193–258, 1992.

[89] Steve Hanks and Daniel S. Weld. A Domain-Independent Algorithm for Plan Adaptation. *Journal of Artificial Intelligence Research*, pages 5–6, 1995.

[90] Ralph Bergmann, Hector Muñoz-Avila, and Manuela Veloso. Case-based planning: Selected methods and systems. *AI Communications*, 9(3):128–137, 1996.

[91] Flavio Tonidandel and Márcio Rillo. The FAR-OFF System: A Heuristic Search Case-Based Planning. *Proceedings of the 6th International Conference on Artificial Intelligence Planning Systems, AIPS 2002*, pages 302–311, 2002.

[92] Ivan Serina. Kernel functions for case-based planning. *Artificial Intelligence*, 174(16-17):1369–1406, 2010.

[93] Hankz Hankui Zhuo, Tuan Nguyen, and Subbarao Kambhampati. Model-lite case-based planning. *Proceedings of the 27th AAAI Conference on Artificial Intelligence, AAAI 2013*, pages 1077–1083, 2013.

[94] Tomas de la Rosa, Angel Garciz-Olaya, and Daniel Borrajo. A case-based approach to heuristic planning. *Applied Intelligence*, 39(1):184–201, 2013.

[95] Daniel Borrajo, Anna Roubíčková, and Ivan Serina. Progress in Case-based planning. *ACM Computing Surveys*, 47(2), 2015.

[96] Khen Elimelech, Lydia E. Kavraki, and Moshe Y. Vardi. Automatic Cross-domain Task Plan Transfer by Caching Abstract Skills. *Springer Proceedings in Advanced Robotics*, 25:470–487, 2023.

[97] Khen Elimelech, Lydia E. Kavraki, and Moshe Y. Vardi. Efficient Task Planning Using Abstract Skills and Dynamic Road Map Matching. *Springer Proceedings in Advanced Robotics*, 27:487–503, 2023.

[98] Javier Segovia-Aguas, Sergio Jiménez Celorrio, Laura Sebastiá, and Anders Jonsson. Scaling-Up Generalized Planning as Heuristic Search with Landmarks. *Proceedings of the International Symposium on Combinatorial Search*, 15(1):171–179, 2022.

[99] Fatima Ardjani, Djelloul Bouchiha, and Mimoun Malki. Ontology-Alignment Techniques: Survey and Analysis. *International Journal of Modern Education and Computer Science*, 7(11):67–78, 2015.

[100] Marco Kainzner, Christoph Klösch, Dominik Filipiak, Tek Raj Chhetri, Anna Fensel, and Jorge Martinez-Gil. Towards reusable ontology alignment for manufacturing maintenance. *CEUR Workshop Proceedings*, 2941:1–5, 2021.

[101] Tom Leinster. *Basic Category Theory*. Cambridge University Press, 2016.

[102] Brendan Fong and David I. Spivak. *Seven sketches in compositionality: An invitation to applied category theory*. Cambridge University Press, 2018.

[103] Saunders MacLane. *Categories for the Working Mathematician*. Springer New York, NY, 1971.

[104] Spencer Breiner, Al Jones, and Eswaran Subrahmanian. Categorical models for process planning. *Computers in Industry*, pages 1–35, 2016.

[105] Anthony Cowley and Camillo J. Taylor. Stream-oriented robotics programming: The design of roshask. *IEEE International Conference on Intelligent Robots and Systems*, pages 1048–1054, 2011.

[106] John Baez and Mike Stay. Physics, topology, logic and computation: A Rosetta Stone. *Lecture Notes in Physics*, 813:95–172, 2011.

[107] Justin Curry. Sheaves, Cosheaves and Applications. *arXiv*, 2013.

[108] Michael Robinson. Sheaves are the canonical data structure for sensor integration. *Information Fusion*, 36:208–224, 2017.

[109] Paul Vickers, Joe Faith, and Nick Rossiter. Understanding Visualization: A Formal Approach Using Category Theory and Semiotics . *IEEE Transactions on Visualization & Computer Graphics*, 19(06):1048–1061, June 2013.

[110] Ming Zhu, Peter Grogono, and Olga Ormandjieva. Using category theory to verify implementation against design in concurrent systems. *Procedia Computer Science*, 52(1):530–537, 2015.

[111] Brendan Fong. Decorated cospans. *Theory and Applications of Categories*, 30:1096–1120, 2015.

[112] Brendan Fong, P. Rapisarda, and P. Sobocinski. A categorical approach to open and interconnected dynamical systems. *2016 31st Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*, pages 1–10, 2016.

[113] Brendan Fong, Pawe Sobociski, and Paolo Rapisarda. A categorical approach to open and interconnected dynamical systems. *Proceedings - Symposium on Logic in Computer Science*, 05-08-July(1):495–504, 2016.

[114] Daniel Marsden. Category Theory Using String Diagrams. *arXiv*, pages 1–60, 2014.

[115] W3C OWL Working Group. Owl 2 web ontology language document overview (second edition). `https://www.w3.org/TR/owl2-overview/`, December 2012.

[116] Richard Cyganiak, David Wood, and Markus Lanthaler. Rdf 1.1 concepts and abstract syntax. `https://www.w3.org/TR/rdf11-concepts/`, February 2014.

[117] Stefan Edelkamp and Jörg Hoffmann. PDDL2.2: The Language for the Classical Part of the 4th International Planning Competition 1 Introduction 2 Derived Predicates. Technical report, International Planning Competition, 2004.

[118] Peter Selinger. A survey of graphical languages for monoidal categories. *arXiv*, pages 289–355, 2010.

[119] Blai Bonet and Hector Geffner. Heuristic search planner 2.0. *AI Magazine*, 22(3):77, Sep. 2001.

[120] Jörg Hoffmann and Bernhard Nebel. The FF planning system: Fast plan generation. *Journal of Artificial Intelligence Research*, 14:263–312, 2001.

[121] Damien Pellier and Humbert Fiorino. Pddl4j: a planning domain description library for java. *Journal of Experimental & Theoretical Artificial Intelligence*, 30(1):143–176, 2018.

[122] John McCarthy and Patrick Hayes. Some Philosophical Problems From Standpoint of Artificial Intelligence. Technical report, Stanford University, 1969.

[123] Z. Kootbally, C. Schlenoff, C. Lawler, T. Kramer, and S.K. Gupta. Towards robust assembly with knowledge representation for the planning domain definition language (pddl). *Robotics and Computer-Integrated Manufacturing*, 33:42–55, 2015. Special Issue on Knowledge Driven Robotics and Manufacturing.

[124] Frederick Proctor, Stephen Balakirsky, Zeid Kootbally, Thomas Kramer, Craig Schlenoff, and William Shackleford. The canonical robot command language (crcl). *Industrial Robot: An International Journal*, 43:495–502, 08 2016.

[125] Morgan Quigley, Ken Conley, Brian Gerkey, Josh Faust, Tully Foote, Jeremy Leibs, Rob Wheeler, and Andrew Ng. Ros: an open-source robot operating system. *ICRA Workshop on Open Source Software*, 3, 01 2009.

[126] Sdformat, 2020. `http://sdformat.org/spec`.

[127] Kasunic Mark and Anderson William. Measuring systems interoperability: Challenges and opportunities. Technical report, Carnegie Mellon University, 2001.

[128] Saikou Y. Diallo, Andreas Tolk, Jason Graff, and Anthony Barraco. Using the levels of conceptual interoperability model and model-based data engineering to develop a modular interoperability framework. *Proceedings of the 2011 Winter Simulation Conference (WSC)*, pages 2571–2581, 2011.

[129] W. Wang, A. Tolk, and Weiping Wang. The levels of conceptual interoperability model: applying systems engineering principles to m&s. *arXiv*, abs/0908.0191, 2009.

[130] Giovanni Lugaresi, Gianluca Aglio, Federico Folgheraiter, and Andrea Matta. Real-time validation of digital models for manufacturing systems: a novel signal-processing-based approach. In *2019 IEEE 15th International Conference on Automation Science and Engineering (CASE)*, pages 450–455, 2019.

[131] A. Lüder, M. Schleipen, N. Schmidt, J. Pfrommer, and R. Henßen. One step towards an industry 4.0 component. In *2017 13th IEEE Conference on Automation Science and Engineering (CASE)*, pages 1268–1273, 2017.

[132] Takayoshi Takayanagi, Yusuke Kurose, and Tatsuya Harada. Hierarchical task planning from object goal state for human-assist robot. In *2019 IEEE 15th International Conference on Automation Science and Engineering (CASE)*, pages 1359–1366, 2019.

[133] Stuart Russell, Peter Norvig, Ming-Wei Chang, Jacob Devlin, Anca Dragan, David Forsyth, Ian Goodfellow, Jitendra Malik, and Vikash Mansinghka. *Artificial Intelligence: a modern approach*. Pearson, 4th editio edition, 2021.

[134] Michael Beetz, Daniel Bessler, Andrei Haidu, Mihai Pomarlan, Asil Kaan Bozcuoglu, and Georg Bartels. Know Rob 2.0 - A 2nd Generation Knowledge Processing Framework for Cognition-Enabled Robotic Agents. *Proceedings - IEEE International Conference on Robotics and Automation*, pages 512–519, 2018.

[135] Moritz Tenorth and Michael Beetz. Representations for robot knowledge in the KNOWROB framework. *Artificial Intelligence*, 247:151–169, 2017.

[136] Saeid Amiri, Kishan Chandan, and Shiqi Zhang. Reasoning with Scene Graphs for Robot Planning under Partial Observability. *IEEE Robotics and Automation Letters*, 7(2):5560–5567, 2022.

[137] Cipriano Galindo, Juan Antonio Fernández-Madrigal, Javier González, and Alessandro Saffiotti. Robot task planning using semantic maps. *Robotics and Autonomous Systems*, 56(11):955–966, 2008.

[138] Runqing Miao, Qingxuan Jia, and Fuchun Sun. Long-term robot manipulation task planning with scene graph and semantic knowledge. *Robotic Intelligence and Automation*, 2023.

[139] Christopher Agia, Krishna Murthy Jatavallabhula, Mohamed Khodeir, Ondrej Miksik, Vibhav Vineet, Mustafa Mukadam, Liam Paull, and Florian Shkurti. TASKOGRAPHY: Evaluating robot task planning over large 3D scene graphs. *5th Conference on Robot Learning*, pages 1–18, 2021.

[140] Zhen Zeng, Zheming Zhou, Zhiqiang Sui, and Odest Chadwicke Jenkins. Semantic Robot Programming for Goal-Directed Manipulation in Cluttered Scenes. *Proceedings - IEEE International Conference on Robotics and Automation*, pages 7462–7469, 2018.

[141] Sylvie Thiebaux, Jorg Hoffmann, and Bernhard Nebel. In Defense of PDDL Axioms. *Syntax And Semantics*, pages 1–6, 2000.

[142] Franc Ivankovic and Patrik Haslum. Optimal planning with axioms. *IJCAI International Joint Conference on Artificial Intelligence*, 2015-Janua(Ijcai):1580–1586, 2015.

[143] Thomas R. Gruber. A translation approach to portable ontology specifications. *Knowledge Acquisition*, 5(2):199–220, 1993.

[144] Lavdim Halilaj, Juergen Luettin, Cory Henson, and Sebastian Monka. Knowledge Graphs for Automated Driving. *Proceedings - 2022 IEEE 5th International Conference on Artificial Intelligence and Knowledge Engineering, AIKE 2022*, pages 98–105, 2022.

[145] David Paulius and Yu Sun. A Survey of Knowledge Representation in Service Robotics. *Robotics and Autonomous Systems*, 118:13–30, 2019.

[146] Moritz Tenorth, Lars Kunze, Dominik Jain, and Michael Beetz. KNOWROB-MAP – Knowledge-Linked Semantic Object Maps. *IEEE-RAS International Conference on Humanoid Robots*, pages 430–435, 2010.

[147] Séverin Lemaignan, Raquel Ros, Lorenz Mösenlechner, Rachid Alami, and Michael Beetz. ORO, a knowledge management platform for cognitive architectures in robotics. *IEEE/RSJ 2010 International Conference on Intelligent Robots and Systems, IROS 2010 - Conference Proceedings*, pages 3548–3553, 2010.

[148] Philipp Matthias Schäfer, Franz Steinmetz, Stefan Schneyer, Timo Bachmann, Thomas Eiband, Florian Samuel Lay, Abhishek Padalkar, Christoph Sürig, Freek Stulp, and Korbinian Nottensteiner. Flexible Robotic Assembly Based on Ontological Representation of Tasks, Skills, and Resources. *Proceedings of the 18th International Conference on Principles of Knowledge Representation and Reasoning, KR 2021*, pages 702–706, 2021.

[149] Iro Armeni, Jerry Zhi-yang He, Junyoung Gwak, Amir R Zamir, Martin Fischer, Jitendra Malik, and Silvio Savarese. 3D Scene Graph. *Iccv*, 2019.

[150] Sylvie Thiébaux, Jörg Hoffmann, and Bernhard Nebel. In defense of PDDL axioms. *Artificial Intelligence*, 168(1-2):38–69, 2005.

[151] Bart Jacobs. *Categorical Logic and Type Theory*, volume 141. Elsevier, 1999.

[152] Marie La, Palme Reyes, and Gonzalo E Reyes. *Generic figures and their glueings*. Polimetrica, 2008.

[153] Daniel Rosiak. *Sheaf Theory through Examples*. The MIT Press, 10 2022.

[154] AlgebraicJulia. AlgebraicJulia: Bringing compositionality to technical computing. `https://www.algebraicjulia.org`, 2024. Accessed: 2024-04-15.

[155] Emily Riehl. *Category theory in context*. Aurora: Dover Modern Math Originals, 2016.

[156] Evan Patterson. Knowledge Representation in Bicategories of Relations. *CoRR*, pages 1–73, 2017.

[157] David I. Spivak and Ryan Wisnesky. Relational foundations for functorial data migration. *DBPL 2015 - Proceedings of the 15th Symposium on Database Programming Languages*, pages 21–28, 2015.

[158] Angeline Aguinaldo, Evan Patterson, James Fairbanks, William Regli, and Jaime Ruiz. A Categorical Representation Language and Computational System for Knowledge-Based Planning. In *2023 AAAI Fall Symposium on Unifying Representations for Robot Application Development*, 2023.

[159] Annegret Habel, Reiko Heckel, and Gabriele Taentzer. Graph grammars with negative application conditions. *Fundamenta Informaticae*, 26(3-4):287–313, 1996.

[160] Hartmut Ehrig, M. Pfender, and H. J. Schneider. Graph-grammars: An algebraic approach. *14th Annual Symposium on Switching and Automata Theory (swat 1973)*, pages 167–180, 1973.

[161] Saunders Mac Lane and Ieke Moerdijk. *Sheaves in Geometry and Logic: A First Introduction to Topos Theory*. Universitext. Springer-Verlag New York, Inc., 1 edition, 1992.

[162] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. MIT Press, 3rd edition, 2009.

[163] Yu Men, Ligang Jin, Tao Cui, Yunfeng Bai, Fengming Li, and Rui Song. Policy Fusion Transfer: The Knowledge Transfer for Different Robot Peg-in-Hole Insertion Assemblies. *IEEE Transactions on Instrumentation and Measurement*, 72:1–10, 2023.

[164] Lorena Otero-Cerdeira, Francisco J. Rodríguez-Martínez, and Alma Gómez-Rodríguez. Ontology matching: A literature review. *Expert Systems with Applications*, 42(2):949–971, 2015.

[165] Pavel Shvaiko and Jérôme Euzenat. Ontology matching: State of the art and future challenges. *IEEE Transactions on Knowledge and Data Engineering*, 25(1):158–176, 2013.

[166] Patrick Schultz, David Spivak, Christina Vasilakopoulou, and Ryan Wisnesky. Algebraic Databases. *Theory and Applications of Categories*, 32(16):547–619, 2017.

[167] Spencer Breiner, Blake Pollard, and Eswaran Subrahmanian. Functorial Model Management. *Proceedings of the Design Society: International Conference on Engineering Design*, 1(1):1963–1972, 2019.

[168] James R. Hester. A robust, format-agnostic scientific data transfer framework. *Data Science Journal*, 15:1–17, 2016.

[169] Evan Patterson. Categories of diagrams in data migration and computational physics. `https://www.algebraicjulia.org/assets/slides/topos-colloquium-2021.pdf`, 2021. Accessed: 2024-04-15.

[170] David I. Spivak. Functorial aggregation. *arXiv*, 2021.

[171] Kyunghee Koh and Keith J Holyoak. Surface and structural similarity in analogical transfer. *Memory & Cognition*, 15(4):332–40, 1987.

[172] Ali Keshvarparast, Daria Battini, Olga Battaia, and Amir Pirayesh. *Collaborative robots in manufacturing and assembly systems: literature review and future research agenda*, volume 35. Springer US, 2024.

[173] Yunsheng Tian, Karl D. D. Willis, Bassel Al Omari, Jieliang Luo, Pingchuan Ma, Yichen Li, Farhad Javid, Edward Gu, Joshua Jacob, Shinjiro Sueda, Hui Li, Sachin Chitta, and Wojciech Matusik. ASAP: Automated Sequence Planning for Complex Robotic Assembly with Physical Feasibility. *2024 IEEE International Conference on Robotics and Automation (ICRA)*, pages 4380–4386, 2023.

[174] Kuangrong Hao, Chenwei Zhao, and Xiaoyan Liu. A robot-assisted adaptive communication recovery method in disaster scenarios. *Complex and Intelligent Systems*, 10(1):1531–1549, 2024.

[175] Dhruv Batra, Angel X. Chang, Sonia Chernova, Andrew J. Davison, Jia Deng, Vladlen Koltun, Sergey Levine, Jitendra Malik, Igor Mordatch, Roozbeh Mottaghi, Manolis Savva, and Hao Su. Rearrangement: A Challenge for Embodied AI. *arXiv*, 2020.

[176] Allen Institute for Artificial Intelligence. AiTHOR Object Types Documentation. `https://ai2thor.allenai.org/ithor/documentation/objects/object-types`, 2024. Accessed: 2024-04-15.

[177] PDDL4J. Blocksworld domain definition in pddl. `https://github.com/gerryai/PDDL4J/blob/master/pddl/blockworld/blocksworld.pddl`, 2008. Accessed: 2024-05-10.

[178] Andrei Pitkevich and Ilya Makarov. A Survey on Sim-to-Real Transfer Methods for Robotic Manipulation. *SISY 2024 - IEEE 22nd International Symposium on Intelligent Systems and Informatics, Proceedings*, pages 259–265, 2024.

[179] Yufei Wang, Zhou Xian, Feng Chen, Tsun-Hsuan Wang, Yian Wang, Zackory Erickson, David Held, and Chuang Gan. RoboGen: Towards Unleashing Infinite Data for Automated Robot Learning via Generative Simulation. *arXiv*, pages 1–39, 2023.

[180] Daniel Hernández García, Concepción A. Monje, and Carlos Balaguer. Generation and adaptation of robot skills models. *IEEE-RAS International Conference on Humanoid Robots*, 2015-February:173–178, 2015.

[181] Haizhen Li and Xilun Ding. Adaptive and intelligent robot task planning for home service: A review. *Engineering Applications of Artificial Intelligence*, 117(July 2022):105618, 2023.

[182] Yulin Zhang, Dylan A. Shell, and Jason M. O'Kane. *Finding Plans Subject to Stipulations on What Information They Divulge*, volume 14. Springer International Publishing, 2020.

[183] Murtadha Alsayegh, Jose Fuentes, Leonardo Bobadilla, and Dylan A. Shell. Oblivious Markov Decision Processes: Planning and Policy Execution. *Proceedings of the IEEE Conference on Decision and Control*, 2(Cdc):3850–3857, 2023.

[184] Nicholas Demarinis, Stefanie Tellex, Vasileios P. Kemerlis, George Konidaris, and Rodrigo Fonseca. Scanning the internet for ROS: A view of security in robotics research. *Proceedings - IEEE International Conference on Robotics and Automation*, 2019-May:8514–8521, 2019.

[185] Jaehyoung Park, Dong Seong Kim, and Hyuk Lim. Privacy-preserving reinforcement learning using homomorphic encryption in cloud computing infrastructures. *IEEE Access*, 8, 2020.

[186] Giray Havur, Kadir Haspalamutgil, Can Palaz, Esra Erdem, and Volkan Patoglu. A case study on the Tower of Hanoi challenge: Representation, reasoning and execution. In *2013 IEEE International Conference on Robotics and Automation*, pages 4552–4559, 2013.

[187] Yuchul Jung, Hogun Park, Yoonjung Choi, and Sung-Hyon Myaeng. Designing a cognitive case-based planning framework for home service robots. In *RO-MAN 2007 - The 16th IEEE International Symposium on Robot and Human Interactive Communication*, pages 827–832, 2007.

[188] Angeline Aguinaldo, Jacob Bunker, Blake Pollard, Ankit Shukla, Arquimedes Canedo, Gustavo Quiros, and William Regli. RoboCat: A Category Theoretic Framework for Robotic Interoperability Using Goal-Oriented Programming. *IEEE Transactions on Automation Science and Engineering*, pages 1–9, 2021.

[189] Spencer Breiner, Ram D. Sriram, and Eswaran Subrahmanian. *Compositional Models for Complex Systems*. Academic Press, 2019.

[190] Gioele Zardini, David I Spivak, Andrea Censi, and Emilio Frazzoli. A compositional sheaf-theoretic framework for event-based systems. In *Proceedings of the 3rd Annual International Applied Category Theory Conference, Cambridge, USA*, volume 333 of *Electronic Proceedings in Theoretical Computer Science*, pages 139–153. Open Publishing Association, July 2021.

[191] Gioele Zardini, David I. Spivak, Andrea Censi, and Emilio Frazzoli. A compositional sheaf-theoretic framework for event-based systems. *Electronic Proceedings in Theoretical Computer Science, EPTCS*, 333:139–153, 2021.

[192] Zejun Yang, Li Ning, Haitao Wang, Tianyu Jiang, Shaolin Zhang, Shaowei Cui, Hao Jiang, Chunpeng Li, Shuo Wang, and Zhaoqi Wang. Text2Reaction : Enabling Reactive Task Planning Using Large Language Models. *IEEE Robotics and Automation Letters*, 9(5):4003–4010, 2024.

[193] Vasileios Vasilopoulos, Sebastian Castro, William Vega-Brown, Daniel E. Koditschck, and Nicholas Roy. A Hierarchical Deliberative-Reactive System Architecture for Task and Motion Planning in Partially Known Environments. *Proceedings - IEEE International Conference on Robotics and Automation*, pages 7342–7348, 2022.

[194] Jonathan Styrud, Matteo Iovino, Mikael Norrlof, Marten Bjorkman, and Christian Smith. Combining Planning and Learning of Behavior Trees for Robotic Assembly. *Proceedings - IEEE International Conference on Robotics and Automation*, pages 11511–11517, 2022.

[195] Santiago Franco, Levi H S Lelis, Mike Barley, Stefan Edelkamp, Moises Martinez, and Ionut Moraru. The Complementary2 Planner in the IPC 2018. *International Planning Competition 2018 Classical Tracks*, pages 3–6, 2018.

[196] Sergej Scheck, Alexandre Niveau, and Bruno Zanuttini. Knowledge Compilation for Nondeterministic Action Languages. *Proceedings International Conference on Automated Planning and Scheduling, ICAPS*, 2021-Augus:308–316, 2021.

[197] Maria Fox and Derek Long. PDDL2.1: An extension to PDDL for expressing temporal planning domains. *Journal of Artificial Intelligence Research*, 20:61–124, 2003.

[198] A. K. Pani and G. P. Bhattacharjee. Temporal representation and reasoning in artificial intelligence: A review. *Mathematical and Computer Modelling*, 34(1-2):55–80, 2001.

[199] Michael Katz, Shirin Sohrabi, and Octavian Udrea. Top-Quality Planning: Finding Practically Useful Sets of Best Plans. *AAAI 2020 - 34th AAAI Conference on Artificial Intelligence*, pages 9900–9907, 2020.

[200] Aliakbar Akbari, Mohammed Diab, and Jan Rosell. Contingent task and motion planning under uncertainty for human-robot interactions. *Applied Sciences (Switzerland)*, 10(5), 2020.

[201] Bradley Hayes and Brian Scassellati. Effective robot teammate behaviors for supporting sequential manipulation tasks. *IEEE International Conference on Intelligent Robots and Systems*, 2015-December:6374–6380, 2015.

[202] Vladimir Lifschitz and Leora Morgenstern. Knowledge Representation and Classical Logic. *Elsevier*, 2007.

[203] Robert Arp, Barry Smith, and Andrew D. Spear. *Building Ontologies with Basic Formal Ontology*. MIT Press, Cambridge, MA, 2015.

[204] Ontology matching through alignment and extension: a best practice. `https://docs.crow.nl/ontology-alignment/whitepaper/`. Accessed: 2025-02-14.

[205] Martin Markl, Steve Schnider, and Jim Stasheff. *Operads in Algebra, Topology, and Physics*. Mathematical Surveys and Monographs, volume 96 edition, 2002.

[206] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language models are few-shot learners. *Advances in Neural Information Processing Systems*, 2020-December, 2020.

[207] Karthik Valmeekam, Matthew Marquez, Sarath Sreedharan, and Subbarao Kambhampati. On the Planning Abilities of Large Language Models: A Critical Investigation. *Advances in Neural Information Processing Systems*, 36(NeurIPS), 2023.

[208] Jiatao Zhang, Lanling Tang, Yufan Song, Qiwei Meng, Haofu Qian, Jun Shao, Wei Song, Shiqiang Zhu, and Jason Gu. FLTRNN: Faithful Long-Horizon Task Planning for Robotics with Large Language Models. *Proceedings - IEEE International Conference on Robotics and Automation*, pages 6680–6686, 2024.

[209] Kaya Stechly, Karthik Valmeekam, and Subbarao Kambhampati. On the Self-Verification Limitations of Large Language Models on Reasoning and Planning Tasks. *arXiv*, 2024.

[210] Natalya Fridman Noy and Mark A. Musen. Algorithm and Tool for Automated Ontology Merging and Alignment. *Proceedings of the 17th National Conference on Artificial Intelligence and 12th Conference on Innovative Applications of Artificial Intelligence, AAAI 2000*, pages 450–455, 2000.

[211] Fabian M. Suchanek, Serge Abiteboul, and Pierre Senellart. PARIS: Probabilistic alignment of relations, instances, and schema. *Proceedings of the VLDB Endowment*, 5(3):157–168, 2011.

[212] Saleha Javed, Muhammad Usman, Fredrik Sandin, Marcus Liwicki, and Hamam Mokayed. Deep Ontology Alignment Using a Natural Language Processing Approach for Automatic M2M Translation in IIoT. *Sensors (Basel, Switzerland)*, 23(20):1–24, 2023.

[213] A. Carboni and R.F.C. Walters. Cartesian bicategories i. *Journal of Pure and Applied Algebra*, 49(1):11–32, 1987.

[214] David Jaz Myers. String diagrams for double categories and equipments. *arXiv*, pages 1–33, 2016.

[215] Siemens. Image from "ai-enhanced robotics and the future of manufacturing". Siemens Website, 2024. Accessed: 2025-02-14.

[216] CNN. Image accompanying "disaster fatigue: Responders exhausted, americans at risk". `https://www.cnn.com/2024/07/04/opinions/disaster-fatigue-responders-exhausted-americans-at-risk-gaynor/index.html`, 2024. Accessed: 2025-02-14.

[217] Yulin Zhang, Dylan A. Shell, and Jason M. O'Kane. *Finding Plans Subject to Stipulations on What Information They Divulge*, volume 14. Springer International Publishing, 2020.

[218] Xiaolei Sun, Yu Zhang, and Jing Chen. High-level smart decision making of a Robot based on ontology in a search and Rescue Scenario. *Future Internet*, 11(11), 2019.

[219] Michael Chertoff. National Preparedness Guidelines. Technical Report September, U.S. Department of Homeland Security, 2007.