Modeling Multi-Agent Systems with Category Theory

Jinzi Huang

A Thesis
in
The Department
of
Computer Science and Software Engineering

Presented in Partial Fulfillment of the Requirements
for the Degree of Master of Applied Science (Software Engineering) at
Concordia University
Montreal, Quebec, Canada

August 2011

© Jinzi Huang, 2011

CONCORDIA UNIVERSITY

School of Graduate Studies

This is to certify that the thesis prepared

By: Jinzi Huang

Entitled: Modeling Multi-Agent Systems with Category Theory

and submitted in partial fulfillment of the requirements for the degree of

Master of Applied Science (Software Engineering)

complies with the regulations of the University and meets the accepted standards with respect to originality and quality.

Signed by the final examining committee:

____ Chair
Dr. Volker Haarslev

____ Examiner
Dr. Mourad Debbabi

____ Examiner
Dr. Yuhong Yan

____ Supervisor
Dr. Olga Ormandjieva

____ Supervisor
Dr. Jamal Bentahar

Approved by ____
Chair of Department or Graduate Program Director

____
Dr. Robin A. L. Drew, Dean
Faculty of Engineering and Computer Science

Date ____

// Abstract

Modeling Multi-Agent Systems with Category Theory

Jinzi Huang

The rapidly growing complexity of integrating and monitoring computing systems is beyond the capabilities of even the most expert systems and software developers. The solution is systems must learn to monitor their own behaviors and conform to the requirements – a vision referred to as Autonomic Computing. Reactive Autonomic Systems Framework (RASF) is introduced for real-time reactive systems, which contain autonomic self-managing properties and are adaptive to their environments.

The goal of this thesis is about modeling Multi-Agent Systems (MAS) with Category Theory (CAT). MAS is introduced as the realization of Reactive Autonomic Systems, and Jadex is used as a representation of MAS approach. This thesis respects Belief-Desire-Intension (BDI) agent architecture, models the entire Multi-Agent Systems (MAS), zooms into individual intelligent agent, analyzes the relationships among agent plans, goals and beliefs, and provides a fully formal CAT representation on MAS structure. Furthermore, this thesis proposes a formalization of fault-tolerance property of MAS using CAT.

Acknowledgments

I would first like to express my sincere thanks and appreciation to my supervisors: Dr. Olga Ormandjieva and Dr. Jamal Bentahar, for their insight, thoughtful guidance and constant encouragement throughout my study. This thesis would not have been possible without their support and help.

I also would like to express my deep gratitude and respect to Dr. Stan Klasa, whose technical advices and insight were invaluable to me. Thanks to the members of my examination committee: Dr. Yuhong Yan and Dr. Mourad Debbabi, for their valuable discussions and comments. Thanks to all my other professors for their help in these years.

My greatest appreciation and friendship goes to my closest friend, Cui Zhu, who is always a great support in all my struggles and frustrations in my life and studies.

Last but not least, I am forever indebted to my great parents and David for their understanding, endless patience and encouragement when it was most required. Without them I could not have made it here.

Table of Contents

Abstract ... iii

List of Figures ... viii

List of Tables ... x

List of Abbreviations ... xi

Chapter 1: Introduction ... 1

1.1. Problems Statement ... 1

1.2. Context of Research ... 2

1.3. Motivation ... 4

1.4. Research Questions ... 6

1.5. Proposed Approach and Contribution ... 6

1.6. Outline ... 8

Chapter 2: Background ... 9

2.1. Autonomic Computing ... 9

2.1.1. Autonomic Computing Definition ... 10

2.1.2. Autonomic Computing Characteristics ... 11

2.2. Reactive Autonomic Systems (RAS) ... 12

2.3. Multi Agent Systems ... 14

2.3.1. Autonomous Agent ... 14

2.3.2. Multi-Agent Computing ... 15

2.3.3. Agent Architecture ... 16

2.3.4. Jadex BDI Agent System ... 17

2.4. Category Theory ... 19

2.4.1. Definition of Category ... 20

2.4.2. Type Category ... 21

2.4.3. Null Object in Category 22

2.4.4. PATH Category 23

2.5. From Autonomic Systems to Category Theory and Multi-Agent Systems 24

2.6. Case Study 31

Chapter 3: Modeling Multi-Agents System by Category Theory 34

3.1. Introduction 34

3.2. Representing Plans 34

3.2.1. Categorical Representation 34

3.2.2. Illustration 40

3.2.3. Properties 44

3.3. Representing Goals 45

3.3.1. Categorical Representation 46

3.3.2. Illustration 48

3.3.3. Properties 50

3.4. Representing Beliefs 51

3.4.1. Categorical Representation 51

3.4.2. Illustration 53

3.4.3. Properties 56

3.5. Representing Agents 57

3.5.1. Introduction 57

3.5.2. Categorical Representation of Plan and Goal 57

3.5.3. Illustration of Plan and Goal 58

3.5.4. Categorical Representation of Plan and Belief 59

3.5.5. Illustration of Plan and Belief 59

3.5.6. Categorical Representation of Goal and Belief 60

3.5.7. Illustration of Goal and Belief 61

3.5.8. Plan, Goal and Belief Together ... 62

3.6. Representing Multi-Agent Systems ... 64

3.6.1. Categorical Representation of MAS ... 65

3.6.2. Repository Agent ... 66

3.6.3. Repository Type ... 67

3.6.4. MAS and Repository Type ... 67

Chapter 4: Fault-Tolerance Properties in Multi-Agents System Categorical Model ... 70

4.1. A Categorical Model for Robotic Case Study ... 70

4.2. Fault-Tolerance ... 81

4.2.1. Fault-Tolerance Property- Restart The Same Agent ... 81

4.2.2. Robotic Case Study: Restart the Same Carry Agent ... 83

4.2.3. Fault-Tolerance Property- Takeover by Inclusion Agent ... 85

4.2.4. Robotic Case Study: Takeover Damaged Carry Agent by Inclusion Agent ... 86

Chapter 5: Related Work and Conclusions ... 93

5.1. Related Work and Significance of the Proposed Research ... 93

5.2. Conclusions ... 96

5.3. Contributions ... 97

5.4. Future Work ... 98

References ... 100

List of Figures

Figure 1.1: Reactive autonomic systems framework: components and stages 4

Figure 1.2: The schema of the proposed approach 7

Figure 2.1: Reactive autonomic system package diagram 13

Figure 2.2: Jadex abstract architecture [PB07] 18

Figure 2.3: Type category example 22

Figure 2.4: Null Object in Category 23

Figure 2.5: Example of PATH Category 24

Figure 2.6: Reactive Autonomic System Project 25

Figure 2.7: An idea of mapping RAS to MAS 25

Figure 2.8: A MAS representation 26

Figure 2.9: Hierarchical agents 27

Figure 2.10: Agents local communication 28

Figure 2.11: Agents global communication 30

Figure 3.1: Representation of the Action category 35

Figure 3.2: Representation of the Plan category 36

Figure 3.3: Representation of the PLAN category 37

Figure 3.4: Representation of the sequence _action and refined _by _plan functors 39

Figure 3.5: Illustration for Plan, PLAN and Discrete-Time example 41

Figure 3.6: Self-update of PLAN 43

Figure 3.7: Representation of the GOAL category 46

Figure 3.8: Representation of the Dependency category 47

Figure 3.9: Representation of the assigned _dependency functor 47

Figure 3.10: Self-update of GOAL 49

Figure 3.11: Representation of the FactSet category 52

Figure 3.12: Representation of the BELIEF category 52

Figure 3.13: Self-update of FactSet 54

Figure 3.14: Self-update of BELIEF 55

Figure 3.15: Functor plan _goal from PLAN to GOAL 58

Figure 3.16: Functor plan _belief from PLAN to BELIEF 60

Figure 3.17: Functor goal _belief from GOAL to BELIEF 62

Figure 3.18: Representation of the Agent category 63

Figure 3.19: MAS category example 66

Figure 3.20: MAS to Repository Type 68

Figure 4.1: Repository Type categories in case study 70

Figure 4.2: Type carry agent 71

Figure 4.3: $Carry_1$ agent 73

Figure 4.4: $Carry_2$ Agent 77

Figure 4.5: Fault-tolerance property- restart in agent A 83

Figure 4.6: Fault-tolerance property takeover by inclusion agent 86

Figure 4.7: Include in action in the case study 87

Figure 4.8: Include in plan in the case study 88

Figure 4.9: Include in PLAN in the case study 89

Figure 4.10: Include in GOAL in the case study 90

Figure 4.11: Include in BELIEF in case study 91

Figure 5.1: Reactive autonomic systems framework project with research coverage 97

List of Tables

Table 2.1: Agents local communication 28

Table 2.2: Agents global communication 30

Table 2.3: Detect and analyze ore mines use case 31

Table 2.4: Produce ore use case 32

Table 2.5: Delivery ore use case 33

Table 2.6: Recover damaged carry agent use case 33

Table 3.1: Additional Properties of F 69

Table 4.1: F Additional properties map $Action_1$ to Action Type 74

Table 4.2: F Additional properties map $Plan_{1_A}$ to Plan Type 74

Table 4.3: F Additional properties map $Plan_{1_B}$ to Plan Type 75

Table 4.4: F Additional properties map $PLAN_1$ to PLAN Type 75

Table 4.5: F Additional properties map $GOAL_1$ to GOAL Type 76

Table 4.6: F Additional properties map $BELIEF_1$ to BELIEF Type 76

Table 4.7: F Additional properties map $Action_2$ to Action Type 78

Table 4.8: F Additional properties map $Plan_{2_A}$ to Plan Type 78

Table 4.9: F Additional properties map $Plan_{2_B}$ to Plan Type 79

Table 4.10: F Additional properties map $Plan_{2_Move}$ to Plan Type 79

Table 4.11: F Additional properties map $PLAN_2$ to PLAN Type 80

Table 4.12: F Additional properties map $BELIEF_2$ to BELIEF Type 81

List of Abbreviations

RAS .... Reactive Autonomic System

RASF .... Reactive Autonomic System Framework

MAS .... Multi-Agent System

CAT .... Category Theory

RAO .... Reactive Autonomic Object

RAOL .... Reactive Autonomic Object Leader

RAC .... Reactive Autonomic Component

RACG .... Reactive Autonomic Component Group

RACS .... Reactive Autonomic Component Supervisor

RACGM .... Reactive Autonomic Component Group Manager

SS .... System Supervisor

GM .... Group Manager

BDI .... Belief-Desire-Intention

UML .... Unified Modeling Language

AML .... Agent Modeling Language

Chapter 1: Introduction

This thesis is about modeling Multi-Agent Systems (MAS) with Category Theory (CAT). The work presented here is a part of a wide project about modeling and implementing Reactive Autonomic Systems using MAS and CAT. In this chapter, we will discuss the problems of complex software systems, context of research, motivations of using CAT as a formal method and tool to model MAS, research questions, and proposed approach.

1.1. Problems Statement

The rapidly growing complexity of integrating and monitoring computing systems, which are more and more large is beyond the capabilities of even the most expert systems and software developers. System and software complexity crisis is the main obstacle to further progress in IT industry, as the difficulty of managing complex and massive computing systems goes well beyond IT administrators' capabilities. Although current software engineering methodologies and programming language innovations have extended the size as well as complexity of computing systems, only relying on those two solutions will not get IT industry through the present software complexity crisis. The remaining option is: systems must learn to monitor their own behavior and conform to the requirements in conjunction with high-level guidance from humans – a vision referred to as *autonomic computing* [Hp01].

1.2. Context of Research

Autonomic Computing. The term *autonomic* is derived from human autonomic nervous system that monitors heartbeat, blood pressure and body temperature without any conscious thought. This self-regulation and separation provides the ability for human beings to concentrate on high level objectives without managing specific details [HP01]. Similarly, *autonomic computing* is described as [Mur04]: The ability to manage computing enterprise through hardware and software that automatically and dynamically respond to the business requirements. This means self-healing, self-configuring, self-optimizing, and self-protecting hardware and software that behave in accordance to defined service levels and policies. Just like the nervous system responds to the needs of the body, the autonomic computing system responds to the needs of the business. Therefore, IT professionals can focus on business oriented objectives instead of computing level tasks with implementation, configuration and maintenance details.

The absence of a formal framework for autonomic systems based on a strong theoretical backbone has encouraged the authors of [KO08] to propose Reactive Autonomic Systems Framework.

Multi-Agent System. A Multi-Agent System (MAS) is a software system possessing a number of autonomous agents that interact with one another and exchange messages through certain agent communication languages [Woo09]. Therefore, those agents require reactive, proactive, and social abilities, so that they can cooperate, coordinate, and negotiate with others through successful interactions. Agents are equipped with different beliefs, goals as well as motivations, and the MAS can achieve its goals, which are difficult to be reached by each individual agent.

Category Theory. Category theory (CAT) is an area of study in mathematics that examines in an abstract way the properties of particular mathematical concepts, by formalizing them as collections of objects and arrows (morphisms). A ***category*** consists of the following components:

- *Objects*: *A, B, C*, etc.
- *Morphisms*: f, g, h, etc.
- *Domain* and *Codomain:* For each arrow (morphism) f there are given objects: $dom(f)$, $cod(f)$ called the *domain* and *codomain* of f. We write: $f: A \rightarrow B$ to indicate that $A = dom(f)$ and $B = cod(f)$.b
- *Composition:* Given arrows $f: A \rightarrow B$ and $g: B \rightarrow C$, i.e. with: $cod(f) = dom(g)$, there is a given arrow: $g \circ f: A \rightarrow C$, called the *composite* of f and g.
- *Identity:* For each object A there is a given arrow $1_A: A \rightarrow A$, called the *identity* arrow of A.

These components are required to satisfy the following laws:

- *Associativity*: $h \circ (g \circ f) = (h \circ g) \circ f$, for all $f : A \rightarrow B$, $g : B \rightarrow C$, and $h : C \rightarrow D$.
- *Unit*: $f \circ 1_A = f = 1_B \circ f$, for all $f: A \rightarrow B$.

Reactive Autonomic Systems Framework. *Reactive Autonomic Systems Framework* (*RASF*) [KO08] was introduced to realize the vision of large-scale self-managing autonomic systems built from potentially very large numbers of highly autonomic and reactive, yet socially interactive, elements. To model, validate and implement the properties of *RAS*, new techniques have to be developed to add to existing formal methods and tools. *RASF* includes four basic components: *RAS*, *MAS*, *CAT* and *Jadex*, and consists of five stages as follows (Figure 1.1):

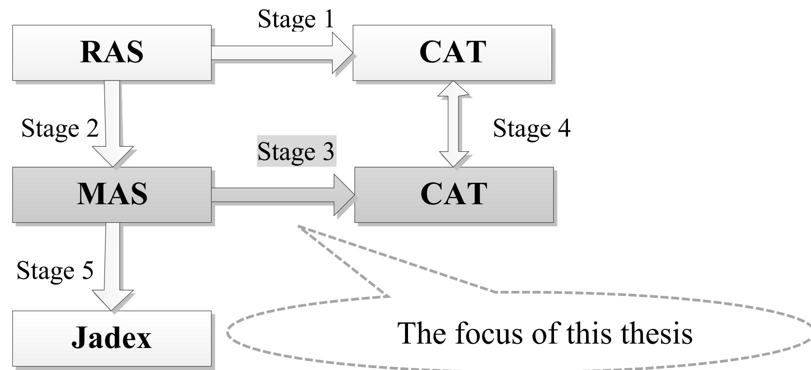


Figure 0.1: Reactive autonomic systems framework: components and stages

Stage 1: Using *Category Theory* (*CAT)* approach as a formal language to specify RAS' autonomic behavior.

Stage 2: Using *Multi-Agent Systems (MAS)* to design and implement *RAS*. A Mapping from *RAS* to *MAS* aims at reducing the gap between the formal specification of *RAS* and its implementation.

Stage 3: Applying *CAT* to formalize *MAS'* autonomic behavior.

Stage 4: Proving the isomorphism between the two *categorical models* mapped respectively from *RAS* and *MAS*. This step will guarantee that the autonomic behaviors of *MAS* translated from *RAS* are correct.

Stage 5: Implementing the created *MAS* with *Jadex*.

This thesis focuses on stage 3 "Modeling *MAS* to *CAT*", whereas stages 2 and 5 have been implemented in [Sha11], stage 1 is described in [SERA 2009, SoMeT 2010, ASAP 2010], and stage 4 is the future work.

1.3. Motivation

Category Theory (CAT) has been introduced and used as a framework in many areas of computer science and software engineering fields [Fia98]. This framework offers a

structure for formalizing large specifications and provides composition primitives in both algebraic [Wir90] and temporal logic specification languages [FM92]. Category theory has a rich body of theory to reason about objects along with their relations (specifications as well as their interactions), and is abstract enough to capture a wide range of different specification languages. Moreover, with category theory and its own properties, automation can be achieved, for example, the composition of two specifications can be derived automatically.

The motivation of using Category Theory (CAT) to model Multi-Agent Systems (MAS) and Reactive Autonomic Systems (RAS) in Reactive Autonomic Systems Framework (RASF) is that CAT is considered as a formal modeling method and powerful tool for abstracting from individual components to specifications and capturing the interactions and compositions among those components in a natural way, which cannot be done using some other semi-formal languages (i.e UML). By comparing the two CATs obtained respectively from RAS (stage 1) and MAS (stage 3), we can check the correctness of RAS transformation to MAS. Another important motivation of using CAT is category theory from mathematical point of view is the study of (abstract) algebras of functions, so using this theory allows us to focus on the morphisms or relationships among objects, instead of concentrating on objects' representations, which is suitable for agent-based systems, since communication among agents is a first-class concept [PB07].

The motivation behind using MAS in RASF is mainly due to the fact that the MAS approach is well suited for autonomic computing systems because the ability of an autonomous agent can be easily mapped to self-managing behaviors in autonomic systems, where agents provide natural solutions to model autonomic components. In

addition, the ability of MAS to manage interactions among components explicitly and control them in a flexible way provides a solution for the distributed complexity [TC04]. Autonomic systems can adapt many features and properties from MAS, such as emergent behavior, automatic group formation, agent coordination, agent adaptation, virtual localization, knowledge mining, interfacing, and evolution [WH03].

1.4. Research Questions

We are aiming to address the following research questions in this work:

1. How can each agent be modeled with CAT?
    a. What are the components of each agent?
    b. How do we model each component with CAT?
    c. How do we model the relations among the components with CAT?
2. How can MAS be modeled with CAT?
    a. What are objects and morphisms to be used to capture the transformation from MAS to CAT?
    b. Since agents and their communication can be classified into different types, how do we model these types with CAT?
3. How can CAT represent MAS properties?

1.5. Proposed Approach and Contribution

Our goal of this thesis is to provide modeling assistance as a foundation for the graphical formalization of the MAS requirements (both functional and non-functional), their interrelations and change management within the MAS life cycle, in terms of Category theory. This goal can be distributed into the following objectives:

1. Modeling Agent with CAT [Chapter 3]

    a. Modeling agent's plans, goals, and beliefs with CAT

    b. Modeling relations between plans and goals, plans and beliefs, and goals and plans with CAT

2. Modeling MAS with CAT [Chapter 3]

    a. Modeling relations between agents

    b. Applying Type Category in MAS

3. Modeling robotic fault-tolerance with CAT [Chapter 4]

Figure 1.2 illustrates the proposed approach of this thesis.

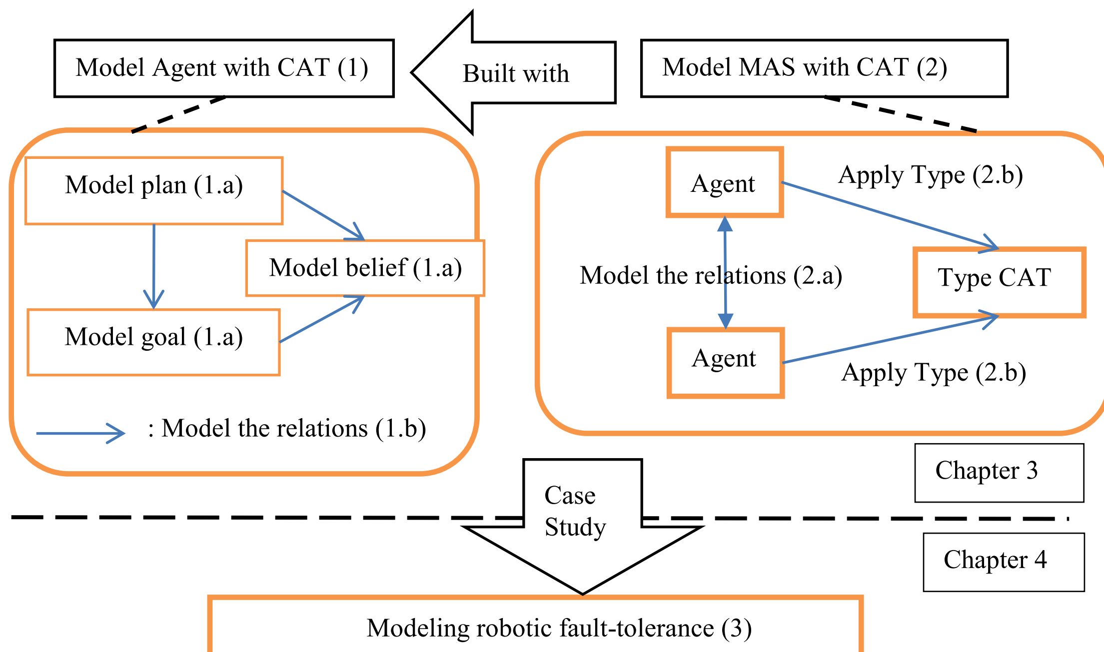


Figure 0.2: The schema of the proposed approach

1.6. Outline

The thesis is organized as follows. In Chapter 2, we introduce the basic concepts of *autonomic computing*, *multi-agent system*, *reactive autonomic system*, *category theory* and description of a case study. Chapter 3 is the core of this thesis that presents the modeling of MAS by using CAT concepts, which includes mapping agent's plans, goals, and beliefs to our defined categories. We also prove some properties of the category representation. In Chapter 4, we introduce robotic case study, more specifically its fault-tolerance property, and the corresponding modeling with the CAT concepts introduced in Chapter 3. In Chapter 5, we briefly review the related work on using category theory to formalizing multi-agent systems and conclude this thesis with a short summary of the presented work and an outline of future work directions.

Chapter 2: Background

In this chapter, we will introduce the backgrounds concerning *autonomic computing*, *reactive autonomic systems*, *category theory* and *multi-agent systems* required to understand the remaining chapters of this thesis. In particular, we will use the definitions of *autonomic computing* and *reactive autonomic systems framework* [Mur04 KO08], introduce *multi-agent systems* and *Jadex implementation environment* [Woo09 PB07] and use *category theory* [Awo06] as formal modeling language. Interested readers can refer to [Mur04 KO08 Woo09 PB07 Awo06] for a more detailed discussion.

2.1. Autonomic Computing

Within the past three decades, the developments of computer hardware and software have grown at exponential rates as software requirements are getting more intricate. As a result, these phenomenal growths along with the advent of the Internet have led to a new age of accessibility - to people, systems, and most importantly, to information. These growths have also led to unprecedented levels of complexity. This complexity is derived from the following aspects:

- The need to integrate several heterogeneous software environments into one cooperated computing system, and to extend trillions computing devices connected to the Internet.

- The rapid stream of changing and conflicting demands at runtime requires timely along with decisive responses.
- As the growing uncertainty of software environments due to unpredictable, diverse and interconnected computing systems, it is very difficult to anticipate and design interactions among the elements of those systems.

The simultaneous explosion of information and integration of technology into everyday life has brought on new demands for how people manage and maintain computer systems. This brings difficulties to design, develop, and maintain software systems. Currently this volume of complexity is managed by highly skilled humans; but the demand for skilled IT personnel is already outstripping supply. From both economic and software development points of view, a solution for software system with self-managing characteristics is urgently necessary.

2.1.1. Autonomic Computing Definition

IBM has introduced a new paradigm for the future of computing-- "*autonomic computing*" [Mur04]. The main idea behind autonomic computing is to shift the fundamental definition of the IT technology from one of purely computing to one defined by data. Access to data from both distributed and centralized sources will allow users to transparently access information when and where they need it. Furthermore, this new computing vision and paradigm will require changing the industry's focus on processing speed and storage to one of developing distributed systems that are largely self-managing, self-diagnostic, and transparent to the user. *Autonomic computing* is not a totally new technology, but a goal-oriented and holistic computing paradigm that aims at developing computer systems having a high degree of autonomy. Thus, autonomic computing is not

a conventional computer systems project, but a visionary approach that groups existing technologies together to achieve a common goal [SB02].

The term *autonomic* is derived from human autonomic nervous system that monitors heartbeat, blood pressure and body temperature without any conscious thought. This self-regulation and separation provides the ability for human beings to concentrate on high level objectives without managing specific details

Similarly, *autonomic computing* is described as [Mur04]: "The ability to manage computing enterprise through hardware and software devices that automatically and dynamically responds to the business requirements. This means developing and managing self-healing, self-configuring, self-configuring, self-optimising, and self-protecting hardware and software systems so that they behave in accordance to defined service levels and policies. "Just like the nervous system responds to the needs of the body, the autonomic computing system responds to the needs of the business".

2.1.2. Autonomic Computing Characteristics

The essence of autonomic computing systems is self-management that can be achieved by realizing self-configuration, self-healing, self-optimization and self-protection.

- *Self-Configuration* [KD03]: *Autonomic computing* systems are able to configure themselves automatically according to high level policies representing business level objectives, which specify what is required instead of how they are implemented. For instance, after a new element joins, it automatically learns composition as well as configuration of the system and registers itself in terms of being used by other elements.

- *Self-Healing* [KD03]: *Autonomic computing* systems can detect, manage and repair bugs or failures in software as well as hardware systems. For example, a problem diagnosis component analyzes information from log files or monitors by using system knowledge, and then compares the diagnosis against system patches or alerts IT professionals. Finally, the system installs the appropriate patches followed by a suitable test.
- *Self-Optimization* [KD03]: *Autonomic computing* systems are able to improve their operations and make themselves more efficient in performance or cost. For example, they can monitor, test and tune their parameters; they also can proactively upgrade their functions through finding, verifying, applying and validating the latest updates.
- *Self-Protection* [KD03]: *Autonomic computing* systems can protect the whole system against malicious attacks and failures uncorrected by self-healing; they are also able to predict and anticipate problems according to early reports from sensors and react to avoid or mitigate them.

IBM has addressed some benefits of *Autonomic computing* [IBM01]. In short-term, it will reduce dependence on human intervention to maintain complex systems accompanied by a substantial decrease in costs. In long-term it will allow individuals, organizations and businesses to collaborate on solving complex problems.

2.2. Reactive Autonomic Systems (RAS)

Reactive Autonomic System (*RAS*) was introduced by the authors of [KO08], which includes four tiered components: Reactive Autonomic Object (*RAO*), Reactive Autonomic Component (*RAC*), Reactive Autonomic Component Group (*RACG*) and Reactive Autonomic System (*RAS*), which are shown by a package diagram in Figure 2.1.

Since RASF is a layered framework, each tier only can communicate with the same tier or the tier immediately above or below. With this design methodology, the system obtains modularity, encapsulation, hierarchical decomposition and reusability. Additionally, autonomic behavior is implemented by the *RAO* Leaders (*RAOL*), *RAC* Supervisors (*RACS*), and *RACG* Managers (*RACGM*) at the *RAC*, *RACG*, as well as *RAS* tiers.

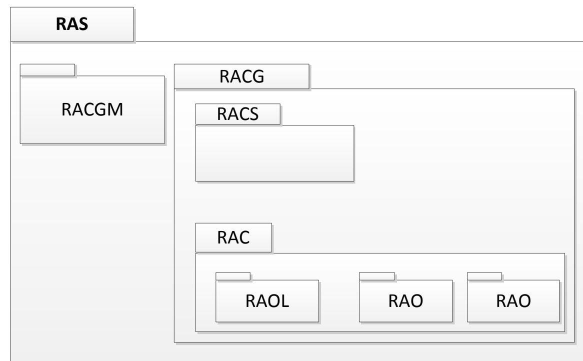


Figure 0.1: Reactive autonomic system package diagram

RAO is Reactive Autonomic Object, which is modeled as a label transition system augmented with ports, time constraints, attributes, and logical assertion on those attributes [OQ08]. *RAC* is Reactive Autonomic Component, which includes synchronously communicated *RAOs*, and where one of the *RAO* is assigned as a leader (*RAOL*) for the remaining *RAOs*. *RAOs* are mainly responsible for reactive tasks, while *RAOL* works on autonomic tasks [KO08]. *RACG* is Reactive Autonomic Component Group, which is constructed by centralized or distributed *RACs*, and the communication between *RACs* has to be synchronous. *RAC* is the minimal reactive autonomic element, which can independently accomplish complete reactive tasks in the *RAS* meta-model. Each *RACG* has a special *RAC* acting as the group supervisor (*RACS*) and all other *RAC* within the same group are under its supervision. *RAS* is the entire system, and it includes

all the centralized or distributed *RACSs* with asynchronous communication. Within each *RAS*, a special *RAC* will be assigned as the system manager (*RACGM*). *RAOL, RACS*, and *RACGM* ensure autonomic tasks are done by intelligent control loops [KC03] modeled as labeled transition systems, where a set of states specifies their task status; a set of events introduces triggers from a state to another and a set of transitions representing states sequence under certain time constraints [KO08].

2.3. Multi Agent Systems

The Multi-Agent System (MAS) approach is well suited for autonomic computing systems since agent-based computing is a natural way to model autonomic systems. In fact, the ability of an autonomous agent can be easily mapped to self-managing behaviors in autonomic systems. In addition, the ability of MAS to engineer interactions among components explicitly and control them in a flexible way supports a more distributed complexity [TC04]. Autonomic systems can adapt many features and properties from MAS, such as emergent behavior, automatic group formation, agent coordination, agent adaptation, virtual localization, knowledge mining, interfacing, and evolution [WH03].

In this section we will discuss agent and multi-agent systems, agent architecture, and Jadex (the agent-oriented programming applied to Beliefs, Desires, and Intentions (BDI model).

2.3.1. Autonomous Agent

An agent is defined as a computer system functioning within an environment, and is capable of performing independent autonomous actions in order to achieve its design objectives [Woo09]. Agents embody a stronger notion of autonomy than objects do in object-oriented paradigm, and in particular, they make decision for themselves whether

or not they need to perform an action requested by another agent. Moreover, agents are able to control their internal states and own behaviour; they experience environment through their sensors and act by effectors.

An autonomous agent is an agent with the following properties [JS98]:

- Reactive: the agent should perceive its environment and respond in a timely way to the environment changes;
- Proactive: the agent should not simply respond to its environment but take initiatives and be capable to show opportunistic and goal-directed;
- Social: the agent should be able to interact with other agents or users when appropriate to complete it and help others with their activities.

2.3.2. Multi-Agent Computing

A Multi-Agent System (MAS) is a software system possessing a number of autonomous agents that interact with one another and exchange messages through certain agent communication languages [Woo09]. Therefore, those agents are required to be reactive, proactive, and social, so that they are able to cooperate, coordinate, and negotiate with others. The agents act on behalf of users having different and maybe conflicting goals as well as motivations, and the MAS can achieve its goals, which are difficult to be reached by each individual agent. The characteristics of the MAS are [JS98]:

- Each agent has incomplete information or capabilities for solving problems.
- There is no global system control.
- Data is decentralized.
- Computation is asynchronous.

The increasing interest in the MAS research is mainly justified by [JS98]:

- Solving problems that are too large for a centralized agent to solve because of resource limitations, performance bottlenecks, or single-point of failures.
- Allowing for interconnection and interoperation of multiple existing legacy systems.
- Solving problems in which data, expertise, or control is distributed.
- Solving problems that can be naturally regarded as a society of autonomous interacting components or agents.

2.3.3. Agent Architecture

How the agent can be decomposed into a set of component modules and how these modules communicate with each other are specified by the agent architecture. According to [Woo09] three categories should be distinguished:

- *Deliberative* agent architecture: an agent develops plans and makes decisions through logical reasoning and uses logical and mathematical representations of the environment. Belief-Desire-Intention (BDI) architecture is one of the main deliberative agent architectures.
- *Reactive* agent architecture [WJ94]: an agent acts based on stimulus-response rules and it does not need to represent its environment logically. In this architecture, agents are able to take parts in interactions with their environment and respond to its changes.
- *Hybrid* agent architecture: an agent is able to act both deliberatively and reactively. In this architecture, agent designers merge deliberative techniques through symbolic representations and reactive techniques through stimulus-response techniques, so agents can reacting to events without performing complex reasoning.

The BDI architecture is a philosophical model for describing rational agents [104], and it contains specific denotation of *Beliefs*, *Desires* and *Intentions*. The architecture

addresses how *Beliefs*, *Desires* and *Intentions* are represented, updated, processed, and interact with one another. In the BDI architecture, agents with particular mental attitudes are able to choose appropriate actions based on their capabilities and internal states.

Beliefs indicate the agent beliefs about its surroundings, which include the environment and other agents. The Beliefs also include inference rules, which allow acquiring new beliefs. However, unlike knowledge, beliefs may be not true.

Desires are goals that agents would like to achieve, and they are the motivational state of those agents.

Intentions are the targets of agents, and they indicate what the agents have chosen to do, which represent the deliberative state of those agents. In an implemented system (such as Jadex), the Intentions are described as executable plans, which include sequences of actions performed by an agent in order to achieve one or more *desires*.

When new information arrives, agents can update their *beliefs* or *desires*. The new *beliefs* or *desires* are able to trigger certain actions, but only one intended action is selected as well as activated. After executing that action, the intentions of those agents are updated, and the new beliefs or desires are stored.

2.3.4. Jadex BDI Agent System

Jadex, a Java-based and FIPA-compliant agent environment, allows modeling goal-oriented agents according to the BDI architecture. In the abstract Jadex architecture [PB07], an agent is able to communicate by sending and receiving messages. The received messages or goal events can trigger the internal reaction as well as deliberation mechanism of the agent, which dispatches those events to the plans selected from a plan

base. Running plans may access and modify a belief base, exchange messages with other agents, create new goals, and trigger internal events [PB07].

Jadex provides infrastructure allowing the use and exploitation of the BDI model in the context mainstream programming, by introducing beliefs, goals and plans as first class objects that can be created and manipulated inside the agent definition. In Jadex, agents have beliefs, which can be any kind of Java objects and are stored in a belief base. Goals represent the concrete motivations (e.g. states to be achieved) that influence an agent's behavior. To achieve its goals, the agent executes plans, which are java programs. The abstract architecture of a Jadex agent is depicted in Figure 2.2 [PB07].

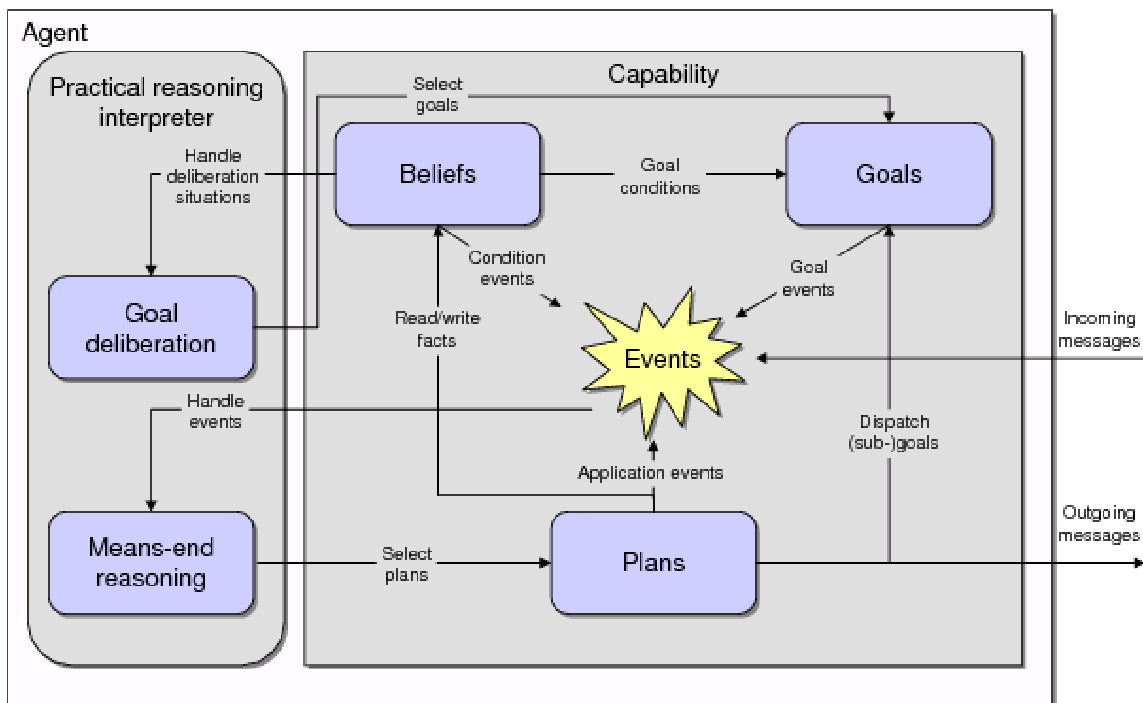


Figure 0.2: Jadex abstract architecture [PB07]

Belief: beliefs in Jadex are a set of facts that make up the knowledge of an agent. Unlike other BDI-based multi-agent systems, where beliefs are represented by certain kind of first-order predicate logic (e.g. Jason) or relational models (e.g. JACK), the

beliefs in Jadex is a storage of knowledge as a database for an agent. Those beliefs cannot support any inference mechanism.

Goal: goals in Jadex are central concepts and not just a special type of event as in pure BDI-based multi-agent systems. Agents are goal-oriented, so they are able to engage into some actions for their goals until they are achieved, unreachable, or undesired. A goal lifecycle consists of the following states [PB05]: option, active, and suspended, which can distinguish between just adopted and actively pursued goals. When a goal is adopted, it becomes an option added to the desire structure, and application specific goal deliberation mechanisms are responsible for managing the state transitions of all adopted goals.

Plan: plans are java procedures used to specify agents' actions towards achieving their goals. Jadex uses a plan-library approach to represent the agents' plans, which are predefined by developers. Those plans are specified in terms of handling events, achieving goals, and building action libraries for the agents.

2.4. Category Theory

In this thesis, category theory is used to specify and formalize MAS for autonomic systems. In this section, we provide an overview of this theory, which is needed to understand the rest of the thesis. Category theory has been introduced and used as a framework in many areas of computer science and software engineering fields [Fia98]. This framework offers a structure for formalizing large specifications and provides composition primitives in both algebraic [Wir90] and temporal logic specification languages [FM92]. Category theory has a rich body of theory to reason about structures (that is objects along with their relations) and is abstract enough to represent a wide range

of different specification languages. Moreover, automation may be achieved in category theory, for example, the composition of two specifications can be derived automatically. Category theory for software specification has adopted a correct by construction approach by which components are specified, proved, and composed in the way of preserving their properties [WE98]. From mathematical point of view, category theory is a study of (abstract) algebras of functions. So using category theory helps us to focuses on the morphisms or relationships between objects, instead of concentrating on objects' representations.

2.4.1. Definition of Category

Definition 2.1 [Awo06]: A ***category*** consists of the following components:

- *Objects*: A, B, C, etc.
- *Morphisms*: $f, g, h,$ etc.
- *Domain* and *Codomain:* For each arrow f there are given objects: $dom(f), cod(f)$ called the *domain* and *codomain* of f. We write: $f: A \to B$ to indicate that $A = dom(f)$ and $B = cod(f)$.
- *Composition:* Given arrows $f: A \to B$ and $g: B \to C$, i.e. with: $cod(f) = dom(g)$, there is a given arrow: $g \circ f: A \to C,$ called the *composite* of f and g.
- *Identity:* For each object A there is a given arrow $1_A: A \to A$, called the *identity* arrow of A.

These components are required to satisfy the following laws:

- *Associativity*: $h \circ (g \circ f) = (h \circ g) \circ f$, for all $f : A \to B, g : B \to C, h : C \to D$.
- *Unit*: $f \circ 1_A = f = 1_B \circ f$, for all $f: A \to B$.

Definition 2.2 [Awo06]: A ***functor*** $F: C \rightarrow D$ between categories C and D is a mapping of objects to objects along with morphisms to morphisms in the way of:

$1)\ F(f: A \rightarrow B) = F(f) : F(A) \rightarrow F(B)$; 2) $F(g \circ f) = F(g) \circ F(f);\ 3)\ F(1A) = 1_{F(A)}$.

Definition 2.3 [Awo06]: in any category C, an arrow $f: A \rightarrow B$ is called an isomorphism if there is an arrow $g: B \rightarrow A$ in C such that $g \circ f = 1_A$ and $f \circ g = 1_B$. Since inverses are unique, $g = f^{-1}$. A is ***isomorphic*** to B: $A \cong B$ if there exists an isomorphism between them.

Definition 2.4 [Awo06]: in any category C, an object is called ***initial object*** "I" if for any object "X" in C, there is a unique morphism $I \rightarrow X$.

Definition 2.5 [Awo06]: in any category C, an object is called ***terminal object*** "T" if for any object "X" in C, there is a unique morphism $X \rightarrow T$.

Definition 2.6 [Awo06]: the ***Category of sets*** is the category whose objects are sets. The arrows or morphisms between sets A and B are all functions from A to B.

Definition 2.7 [Eas99]: ***discrete category*** is a category where the morphisms are only identity morphisms. For example, suppose X and Y are different objects in category C, morphism from X to X only can be X's identity morphism, and morphism from X to Y will not exists, which means:

$$mor\ (X, X) = \{id_X\} \text{ for all objects } X, \text{ and}$$

$$mor\ (X, Y) = \emptyset \text{ for all objects } X \neq Y.$$

2.4.2. Type Category

Definition 2.8 ***Type*** is a category whose objects represent the object types denoted by *ObjType(Type)*, and whose morphisms represent the morphism types denoted by *MorType(Type)*. *MyCategory* is a category whose objects are denoted by *Obj(MyCategory)* and morphisms denoted by *Mor(MyCategory)*. There is a functor F

from *MyCategory* to *Type* which maps each object of *MyCategory* to a type (an object of *Type*): *F(Obj(MyCategory)) = ObjType(Type)*, and maps each morphism of *MyCategory* to a type (a morphism of *Type*): *F(Mor(MyCategory)) = MorType(Type)*.

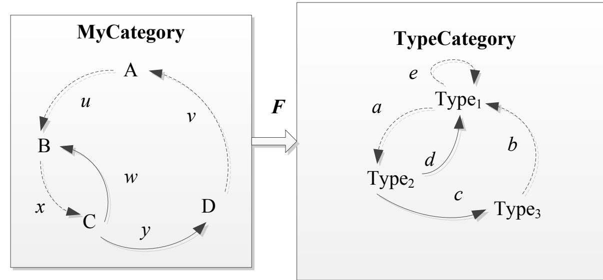


Figure 0.3: Type category example

For example, in Figure 2.3, a *type category* called ***TypeCategory*** contains objects: *typeA*, *typeB* and *typeC*; type *m* morphisms: *c* and *d*, and type *n* morphisms: *a, b* and *e*. ***MyCategory*** contains objects: *A, B, C* and *D*, and morphisms: *u, v, w, x* and *y*. Functor ***F*** maps *MyCategory* objects and morphisms to types in *TypeCategory*: $F(A) = Type_1$, $F(B) = Type_1$, $F(C) = Type_2$, $F(D) = Type_3$, *F(u) = e (type n), F(v) = b (type m), F(w) = d (type m), F(x) = a (type n) and F(y) = c (type m)*.

2.4.3. Null Object in Category

In Chapter 3, we use a special object, called $\boldsymbol{Object_{Null}}$ to help category to catch exceptions. There is no difference between $\boldsymbol{Object_{Null}}$ and other categories' objects, except that $\boldsymbol{Object_{Null}}$ doesn't have any real meaning or content, and it doesn't have any relationship with other object. $\boldsymbol{Object_{Null}}$ and its identity morphism are useful for catching "non-useful" or "non-related" objects and morphisms from other categories through defined functor (relation).

Figure 2.4 is an example of using $\boldsymbol{Object_{Null}}$. ***MyCategory A*** contains objects: *A, B, C*, *D* and $\boldsymbol{Object_{Null}}$, and morphisms: *a, b, c, d* and *e*. ***MyCategory B*** has objects: *A, B*, *C* and

$Object_{Null}$, and morphisms: *a, b* and *c*. Functor ***H*** maps ***MyCategory A*** objects and morphisms to ***MyCategory B***: *H (A) = A, H (B) = B, H (C) = C, H (D) =* ***$Object_{Null}$****, H (****$Object_{Null}$****) =* ***$Object_{Null}$****, H (a) = a, H (b) = b, H (c) = c, H (d) = id* $_{ObjectNull}$ and *H (e) = id* $_{ObjectNull}$. From this example, we can see ***MyCategory B*** contains all the objects and morphisms of ***MyCategory A***, except object *D* and its related morphisms *d* and *e.*

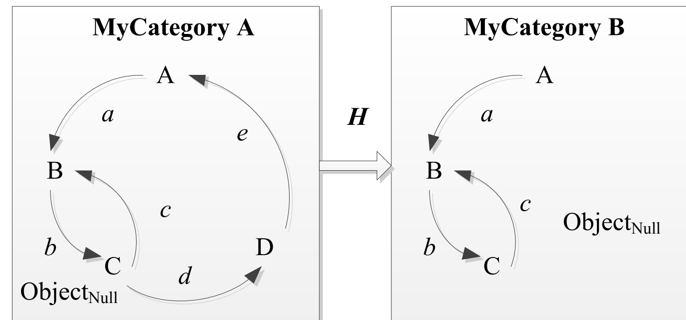


Figure 0.4: Null Object in Category

2.4.4. PATH Category

Before we introduce the PATH category needed in the next chapter, we need to have some background knowledge about directed graphs. A directed graph *G* is a set *O* of objects called vertices or nodes, and a set *A* of ordered pairs of vertices are called arrows or directed edges [Mac71]. Every arrow diagram or directed graph can be interpreted as a category named **PATH**, whose morphisms are sequences (paths) of arrows. One can create a directed graph by drawing an arrow from *x* to *y* where *x, y* ∈ a same set *X*, which can be associated with the category denoted by **PATH (X)** or **PATH [**PS07**]**. The objects are elements in *X* and the morphisms are all sequences (paths) of adjacent arrows. This naturally defines a *composition of arrows*. This viewpoint leads to a general categorical semantics for relational structures. Vice versa, every category is a graphical structure (with nodes and arrows).

Figure 2.5 is an example of PATH. For morphisms (arrows) $f: x \rightarrow y$, $g: y \rightarrow z$ and morphism $k: x \rightarrow z$, if f, g and k are of the same type, then k is not considered as a direct arrow since k equals to the sequence (path) of consecutive arrows (f and g). By the definition of PATH, the lengths of the sequences f and g are one, and the length of k is two. The existence of the identity arrow for each object will always be assumed by definition, and it can be interpreted as sequences of length zero.

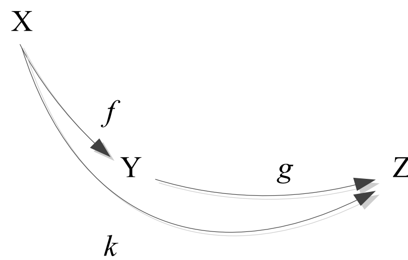


Figure 0.5: Example of PATH Category

2.5. From Autonomic Systems to Category Theory and Multi-Agent Systems

Implementing *Reactive Autonomic System Framework* (*RASF*) has led to propose a methodology including four basic components: *RAS*, *MAS*, *CAT* and *Jadex*, and consists of five stages (Figure 2.6):

- *CAT* (*category theory*) approach will be used as a formal language to specify RAS' autonomic behaviour.
- *MAS* (*multi-agent systems*) will be introduced to design and implement *RAS*. A Mapping from *RAS* to *MAS* will reduce the gap between the formal specification of *RAS* and its implementation.
- *CAT* will be applied for formalizing *MAS'* autonomic behaviour.

- Proving that the two *categorical representations* mapped from *RAS* and *MAS* are isomorphic. This step will guarantee the autonomic behaviours of *RAS* and *MAS* are the same.
- At the end, an implementation with *Jadex* code will be created.

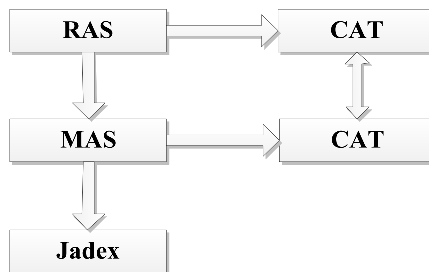


Figure 0.6: Reactive Autonomic System Project

Figure 2.7 shows the general mapping from *RAS* to *MAS*: the elements within *MAS* are layered too, reactive autonomic system (*RAS*) is mapped to multi-agent system (*MAS*); reactive autonomic component group (*RACG*) is mapped to sub-multi-agent system (*sub-MAS*, which is a sub group of agents); reactive autonomic components (*RAC*) are mapped to agents; and reactive autonomic objects (*RAO*) are mapped to agents' plans, goals and beliefs. A *MAS* comprises centralized or distributed *sub-MAS,* which are differentiated by their responsibilities/goals/tasks. A *sub-MAS* contains agent(s), and the agents are grouped by common tasks/goals and differentiated by their individual roles. An agent includes various plans based on agents believes, goals, events and environments.

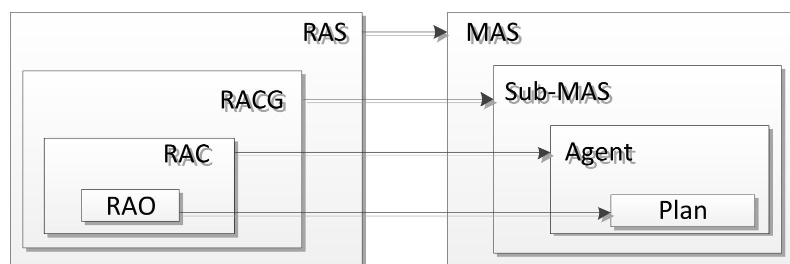


Figure 0.7: An idea of mapping RAS to MAS

Figure 2.8 is a package diagram of *MAS* which reflects the *RAS hierarchy*. It exhibits a static global view of the overall system. The basic components for the system are: *system manager agent*, *supervisor agent*, and *regular agent*. Interested readers can refer to [Sha11] for detailed discussion.

System manager agent is the most essential part that acts as a brain for the overall system. It governs and manages the entire system, and has the most global view which allows it to control and monitor any other agent within the system. It guarantees that the whole system is running correctly.

Supervisor agent exists within each multi-agent group – sub-MAS. It is the group leader that manages the group. It plays a similar role as the system manager agent, but with limited power and localized view of the entire system.

Regular agent is the worker within multi-agent society. Unlike *supervisor agent* or *system manager agent*, worker agents perform actual jobs, obey orders and report events.

Each agent in this package has goals, beliefs and plans components. We chose Jadex BDI architecture to model and specify agents as discussed earlier in this chapter.

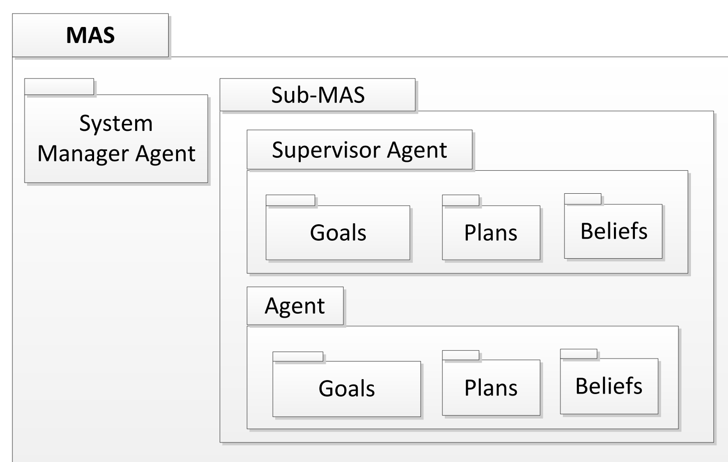


Figure 0.8: A MAS representation

Agents communicate with each other in order to work together to perform different tasks. Agents are hierarchical (Figure 2.9): *regular agents* are in the bottom level, and *system manager agent* is in the up level. Agents can only communicate with the agents in the same level or the level directly below or above. In this case, *system manager agent* can only converse with *supervisor agents*, *regular agents* are only able to communicate with *supervisor agents*, and *supervisor agents* have the ability to send messages to both *system manager agent* and *regular agents*. This design strategy reduces the coupling between agents' communications, and assigns system with modularity, encapsulation, hierarchical decomposition and reusability.

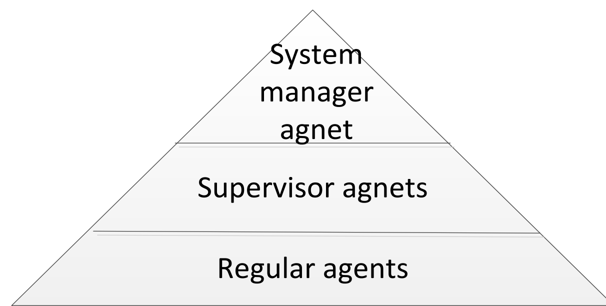


Figure 0.9: Hierarchical agents

In autonomic computing multi-agents system (ACMAS), there are two different communication types: local communication and global communication. Local communication happens only within a group (sub-multi agent system). Inside a group, *regular agents* communicate with each other to cooperate. If communication issues happen between regular agents, error report messages will be sent to *supervisor agent* by concerned *regular agents*. Based on its beliefs, the *supervisor agent* will make a decision and send messages back to the *regular agents*. For example, Table 2.1 and Figure 2.10 represent a local communication use case.

Use Case	*Agents A* requests *Agents B* to work together in a same group	
Scenario	Step	Action
	1	*Agent A* sends a request to ask for *Agent B*'s help on performing a task together
	2	*Agent B* refuses to make an agreement with *Agent A* because it is busy on working on its own task
	3	Both *Agent A* and *Agent B* send reports with explanations to their *Supervisor Agent*
	4	*Supervisor Agent* sends back its decision to *Agent A* and *Agent B*, which is *Agent B* has to abandon its current job and work together with *Agent A*
Post condition	After *Agent A* and *Agent B* receive the decision from *Supervisor Agent*, they will start working together	

Table 0.1: Agents local communication

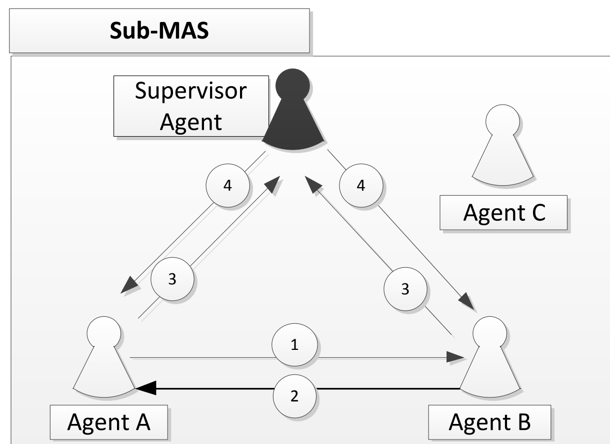


Figure 0.10: Agents local communication

The second case is global communication, which happens between different sub-multi agent systems (subMAS). *Regular agents* are forbidden to communication with

other groups' agents, unless there are some well-defined pre-conditions that clearly address situations in which regular agents can have global communication. For example *Agent A* and *Agent B* are regular agents, and they are from two different sub-multi agent groups. In general, they should not able to communicate with each other, but in order to avoid collision, they have ability to contact each other when they are too close. *Supervisor agents* have ability to communicate with other *supervisor agents*, and *system manager agent*, but they are not allowed to have contacts with regular agents, which are in different groups, except there are some well-defined exceptional situations. System manager agent has the ability to get in touch with supervisor agents. For example, Table 2.2 and Figure 2.11 represent a global communication use case.

Use Case	*Agents A* requests to work with *Agents B* who is from different groups	
Pre-condition	Regular *Agent A* needs to work with *Agent B*, but they belong to two different groups and their communications are limited.	
Scenario	Step	Action
	1	*Agent A* reports "working with *Agent B*" request to its own *Supervisor Agent SA*
	2	*Supervisor Agent SA* accepts *Agent A*'s and negotiates with *Agent B*'s *Supervisor Agent SB*
	3	*Supervisor Agent SB* accepts *Supervisor Agent SA*'s request
	4	*Supervisor Agent SA* sends decided message to *Agent A*

	5	*Supervisor Agent SB* sends decided message to *Agent B*
Exception	Step 3 *Supervisor Agent SA* and *SB* cannot make a decision	
	Exception Steps	
	3.1	*Supervisor Agent SA* and *SB* inform the system manager agent the situation
	3.2	*System Manager Agent* proposes a solution and delegates it back to *Supervisor Agents SA* and *SB*
Post-condition	*Agent A* and *Agent B* will either have communication ability and start to work together or perform other actions based on the decision message	

Table 0.2: Agents global communication

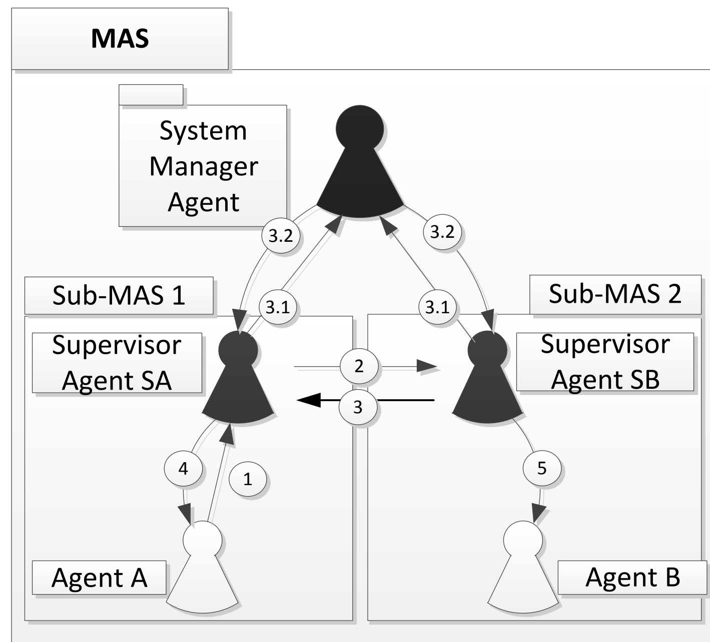


Figure 0.11: Agents global communication

2.6. Case Study

In Chapter 4, we will model a fault-tolerance property with CAT, and this property is based on a robotic case study "Marsworld" [Fer99]. This case study describes a group of agents (which are robot in this case) cooperation together to accomplish ore exploitation goal on the Mars planet. Based on different rolls, agents are classified into three major types: *sentry agent*, *production agent*, and *carry agent*. In order to match the design of *RASF* and better illustrate our approach, we added two more types: *system manager agent* and *group supervisor agent*. *System manager agent* directly receives commands from earth and assigns the orders to different *group supervisor agents*. *Group supervisor agents* will ask *sentry agents* start searching ore mine by the given location. The detailed scenarios are described as following use cases (Table 2.3, 2.4, 2.5, and 2.6):

Use Case	Detect and analyze ore mines
Description	Sentry agents have a sensor to detect and analyze ore mines, and they will inform production agents the valid ore location.
Goal	Sentry agents detect and analyze ore mines successfully.
Actors	Sentry agents
Pre-conditions	Sentry Agents are functional.
Main Scenario	1. Group supervisor agent calls Sentry Agents for searching and analyzing ore mines. 2. Sentry agents move around and looking for ores. 3. Sentry agents analyze found ores. 4. Sentry agents call Production Agent to produce ores which are exploitable.
Post-condition	Amount of ores is delivered to base station.

Table 0.3: Detect and analyze ore mines use case

Use Case	Produce ore
Description	Production agents start performing produce ores task after receive messages from Sentry agents, and they will call carry agents to transport the produced ore to home base.
Goal	Production agents produce ore successfully.
Actors	Production agents
Preconditions	Sentry agents have detected ore mines and analyzed the mines are exploitable.
Main Scenario	1. Production agents receive calls from sentry agents. 2. Production agents move to the mine location and start producing ores.
Post-condition	Amount of ores is produced by Production Agents.

Table 0.4: Produce ore use case

Use Case	Delivery ore
Description	Carry agents start performing delivery ores task after receive calls from production agents, which contain the location and amount of the ore mines.
Goal	Carry agents delivery Ore successfully
Actors	Carry agent
Pre-conditions	Amount of ores is produced by production agents.
Main Scenario	1. Carry Agents receive calls from production agents. 2. Carry Agents move to the mine location and start delivery

	ores to base.
Post-condition	Amount of ores is delivered to base station.

Table 0.5: Delivery ore use case

Use Case	Recover damaged carry agent
Description	Carry agent "*Carry*$_1$" is demanded, but the fault-tolerant property [Fau11] enables the system won't be affected by the failure.
Goal	System continue operating properly
Actors	Carry agents from the same group: "*Carry*$_1$" and "*Carry*$_2$" The group supervisor agent: *Supervisor*
Pre-conditions	*Carry*$_1$ does not perform its tasks correctly
Main Scenario	1. *Supervisor* sends messages to *Carry*$_1$ and asks it to restart from its default stage 2. *Carry*$_1$ reboots itself.
Exceptions	2. *Carry*$_1$ doesn't have any reactions to *Supervisor*'s message 2.1 *Supervisor* communicates with carry agent *Carry*$_2$ in the same group and asks *Carry*$_2$ to take-over the duties of *Carry*$_1$.
Post-condition	*Carry*$_1$ is restarted, or take over by *Carry*$_2$

Table 0.6: Recover damaged carry agent use case

Chapter 3: Modeling Multi-Agents System by Category Theory

3.1. Introduction

In this chapter, we will introduce categorical modeling of multi-agent systems. We will zoom into agent's structure, and represent its main concepts: *plans*, *goals* and *beliefs*, and their relationships via category theory. At the end we will zoom out to the level of entire multi-agent system, and represent it by using category theory constructs. The multi-agent systems definition is taken from [Woo09] [Syc98] and [WJ95] adapted to the context of Agent programming language: Jadex.

3.2. Representing Plans

Plans represent the agent's means to act on the requests initiated by other agents or from its environment, and one single plan is abstracted as a sequence of actions. Therefore, plans of an agent are collections of sequences of actions, which are performed in a discrete time [Woo09]. This section provides category definitions of *Action*, *Plan*, *PLAN*, *Discrete-Time* and their relations. Using these definitions, we will formalize agent's plans by category theory, and capture the behavior and properties of agent's plans and actions.

3.2.1. Categorical Representation

We will define a category, which includes all the needed actions for an agent to perform its plans as follow:

Definition 3.2.1 ***Action*** *is a discrete category whose objects are "actions" denoted by* Act_1*,* Act_2*…, and the only morphisms are identity morphisms.*

In this thesis, "actions" are defined as an abstraction of agents' reaction to the environment events. Figure 3.1 is an example of ***Action category***, where the identity morphisms are not displayed

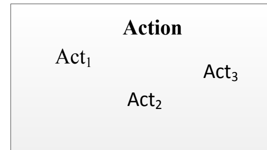


Figure 0.1: Representation of the Action category

Within an agent, a plan represents agent's behaviour, and we abstract a plan as a category *Plan* defined as follows.

Definition 3.2.2 ***Plan*** *is a category that represents one plan whose objects are "actions" denoted by* Act_1*,* Act_2*... and morphisms are named "before" [OMG]. Morphism "before" models the partial order between the actions. A sequence of actions can be understood as a path in category theory [Mac71] [Pfa05] [PS07](see Chapter 2), and only paths of length equal or less than one are considered as morphisms. Inside* ***Plan****, we define a special object, denoted as* Act_{Null} *(chapter 2). An* Act_{Null} *means a null action, and it doesn't have any morphism from or to other actions. In this definition,* Act_{Null} *is used for catching exceptions (detailed example will be given latter).*

Figure 3.2 shows a simple example (The identity morphisms are not displayed) with actions: Act_1 Act_2 and Act_3, and morphisms $f\colon Act_1 \rightarrow Act_2$ and $g\colon Act_2 \rightarrow Act_3$, which models the timing dominance hierarchy: Act_1 occurs earlier than Act_2 and Act_2 occurs earlier than Act_3. In the figure, *<morphisms name> :: <type>* indicates the *type* of the morphism. For instance, morphisms *f::before* and *g::before* means *f* and *g* are of type

before. From the meaning of "*before*" (definition 3.2.2) there should exist a morphism *k::before* such that $k: Act_1 \rightarrow Act_3$ meaning Act_1 occurs earlier than Act_3. In PATH category (Chapter2), the morphisms *f* and *g* are "direct arrows" with sequences (paths) of length one. The morphism *k* from Act_1 to Act_3 is not a "direct arrow" but a path (or sequence) $Act_1 \rightarrow Act_2 \rightarrow Act_3$ with length two. Based on definition 3.2.2, *k* will not be shown within *Plan* category.

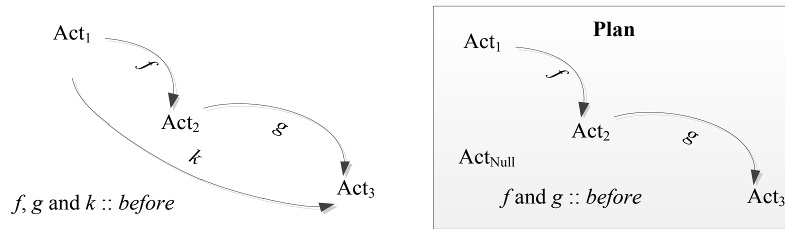


Figure 0.2: Representation of the Plan category

More formally, suppose Act_1 starts at time t_1, Act_2 starts at time t_2, and Act_3 starts at time t_3, where t_1, t_2 and t_3 are integers. Morphism "*before*" indicates: t_1 is less than t_2, and t_2 is less than t_3. There is a composition operation on morphisms, *f::before* $Act_1 \rightarrow Act_2$ and *g::before* $Act_2 \rightarrow Act_3$ are morphisms, then *g o f::before* $Act_1 \rightarrow Act_3$, the composition of *f* and *g* of type *before* is meaningful: Act_1 is performed earlier than Act_3. ***Plan*** satisfies *Associativity* and *Unit* laws (see Chapter 2). Therefore, the validity of the category Plan is proved.

Each ***Plan*** is built by a sequence of actions (actions can be repeated by having morphism of type "*before*" to itself), and the sequence represents a plan. So we say a ***Plan*** stands for one plan of an agent. The first action of the sequence, named *trigger action*, represents the action of receiving "trigger event messages". The received messages can be sent from internal or external source. Internal messages are those sent

from the owner of the plan, and external messages are those sent from other agents or the environment. So when we say a plan is started, we mean the *trigger action* of this plan has been performed.

Within an agent, we need a category to abstract all the plans and their partial orders. We call this category ***PLAN*** and we define it as follows.

Definition 3.2.3 ***PLAN*** *is a category whose objects are plans denoted by* $Plan_1$, $Plan_2$*... and morphisms are "before" [OMG], which model the partial order between plans. This partial order can be understood as a path in category theory [Mac71] [Pfa05] [PS07] (see Chapter2), and only paths of length equal or less than one are considered as validated morphisms. Inside* ***PLAN****, we define a special object, called* $Plan_{Null}$*. A* $Plan_{Null}$ *means a null object, and it doesn't have any morphism from or to other plans.*

In this definition, $Plan_{Null}$ is used for catching exceptions. Figure 3.3 depicts an example of *PLAN*, morphisms $m{::}before{:}\ Plan_1 \rightarrow Plan_2$ and $n{::}before{:}\ Plan_2 \rightarrow Plan_3$ stand for $Plan_1$ is triggered earlier than $Plan_2$, and $Plan_2$ is triggered earlier than $Plan_3$. Similar to "*Plan* category" (Figure 2.2), any "non-direct arrows" or paths with length greater than one are not included in ***PLAN***. Suppose $Plan_1$ is trigged at time t_4, $Plan_2$ is trigged at time t_5 and $Plan_3$ is trigged at time t_6, where t_4, t_5 and t_6 are integers, then we have t_4 is less than t_5, and t_5 is less than t_6.

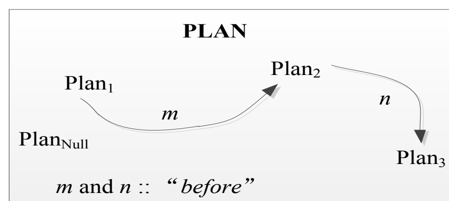


Figure 0.3: Representation of the PLAN category

There is a composition operation on morphisms: from *m::before:* $Plan_1 \rightarrow Plan_2$ and *n::before:* $Plan_2 \rightarrow Plan_3$, we have morphism *n o m::before:* $plan_1 \rightarrow plan_3$. The composition of *m* and *n* is meaningful as if captures the fact that $Plan_1$ is triggered earlier than $Plan_3$. ***PLAN*** satisfies *Associativity* and *Unit* laws (see Chapter 2). Therefore, the validity of the category ***PLAN*** is proved.

We will abstract the relations between categories: *Action*, *Plan* and *PLAN* as functors: *sequence _action, refined _by _plan,* and *self _PLAN.*

Definition 3.2.4 ***sequence _action*** *is a functor from* ***Action*** *(the category of isolated actions) to* ***Plan*** *(the category of sequenced actions). It provides a rule mapping all the "actions" of* ***Action*** *to "actions" of* ***Plan****, and all the identity morphisms of* ***Action*** *to identity morphisms of* ***Plan****.*

Definition 3.2.5 ***refined _by _plan*** *is a functor from* ***Plan*** *to* ***PLAN*** *(the category of plans). The functor "refined _by _plan" means actions in Plan are used to complete or build plans in PLAN. It provides a rule that maps all the "actions" of* ***Plan*** *to "plans" of* ***PLAN****, and all the morphisms (include identity) of* ***Plan*** *to identity morphisms of* ***PLAN****.*

Figure 3.4 illustrates the defined categories: ***Action***, ***Plan*** and ***PLAN***, and functors ***F, G:: sequence _action*** and ***P, Q:: refined _by _plan***. Note that, one agent has one ***Action***, one ***PLAN*** and at least one ***Plan*** categories. Identity morphisms are not displayed in the figure.

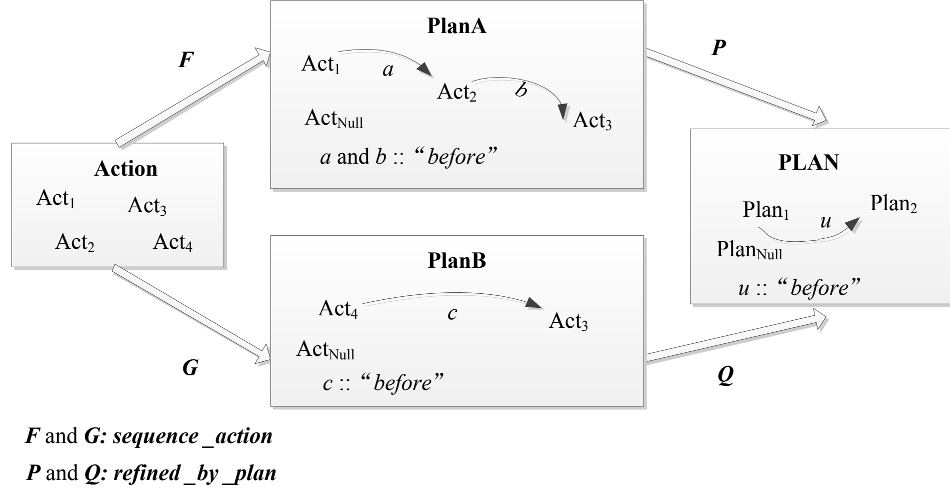


Figure 0.4: Representation of the sequence _action and refined _by _plan functors

Functor ***F:: sequence _action*** provides a rule that maps objects*:* Act_1, Act_2, Act_3 and Act_4 of ***Action*** to objects Act_1, Act_2, Act_3 and Act_{Null} of ***PlanA***; functor ***G:: sequence _action*** maps objects: Act_4, Act_3, Act_2 and Act_3 of ***Action*** to objects Act_4, Act_3, and Act_{Null} of ***PlanB***.

Functor ***P:: refined _by _plan*** provides a rule that maps objects: Act_1, Act_2 and Act_3 of ***PlanA*** to object $Plan_1$ of ***PLAN*** and morphisms: *a*, *b* and *identity* of ***PlanA*** to $Plan_1$'s *identity* morphism (id_{plan1}) of ***PLAN***; ***Q:: refined _by _plan*** maps objects: Act_4 and Act_3 of ***PlanB*** to object $Plan_2$ of ***PLAN*** and morphisms: *c* and *identity* of ***PlanB*** to $Plan_2$'s *identity* morphism (id_{plan2}) of ***PLAN***.

Definition 3.2.6 ***self _PLAN*** *is a functor from* ***PLAN*** *to itself (within the same agent), which maps plans (objects) of* ***PLAN*** *to plans (objects) of* ***PLAN****, and transforms morphisms of* ***PLAN*** *to morphisms of* ***PLAN****.*

Since agent's beliefs are dynamic and changeable, pre-conditions for plans can be various. Some plans may not be achievable anymore after their pre-conditions are changed. We suppose ***PLAN'*** is the new category after ***PLAN*** is translated by ***self***

_PLAN functor. The functor maps achievable plans in ***PLAN*** to plans in ***PLAN'***; maps non-achievable plans in ***PLAN*** to $Plan_{Null}$ in ***PLAN'***; maps $Plan_{Null}$ in ***PLAN*** to $Plan_{Null}$ in ***PLAN'***; maps morphisms "*before*" from achievable plans to achievable plans in ***PLAN*** to morphisms "*before*" in ***PLAN'***; maps morphisms "*before*" from achievable plans to non-achievable plans, or non-achievable plans to achievable plans, or non-achievable plans to non-achievable plans in ***PLAN*** to identity morphism of $Plan_{Null}$ in ***PLAN'***. Additionally, if the non-achievable plan is between two achievable plans, a new morphism ("*before*") will be created to link these achievable plans in ***PLAN'***.

Definition 3.2.7 ***Discrete-Time*** *is a category whose objects are abstracting time unit represented as integers and morphisms are of type "less than" denoted as "<".*

Definition 3.2.8 ***timing _action*** *is a functor from* ***Plan*** *to* ***Discrete-Time,*** *which maps objects (actions) of* ***Plan*** *to objects (time unit expressed as integers) of* ***Discrete-Time****, and maps morphisms of* ***Plan*** *(before) to morphisms "(<)" of* ***Discrete-Time****.*

Definition 3.2.9 ***timing _plan*** *is a functor from* ***PLAN*** *to* ***Discrete-Time,*** *which maps objects (plans) of* ***PLAN*** *to objects (time unit expressed as integers) of* ***Discrete-Time****, and maps morphisms of* ***PLAN*** *(before) to morphisms "(<)" of* ***Discrete-Time****.*

3.2.2. Illustration

The following figures show examples of representing the categories defined above. Figure 3.5 illustrates the definitions: 3.2.1, 3.2.2, 3.2.3, 3.2.4, 3.2.5, 3.2.7, 3.2.8, and 3.2.9, and Figure 3.6 illustrates the definition 3.2.6.

Figure 3.5 depicts a view of the categories ***Action***, ***PlanA***, ***PlanB, PLAN***, ***Discrete-Time***, and their relations (functors): ***sequence _action, refined _by _plan, timing _action*** and ***timing _plan***.

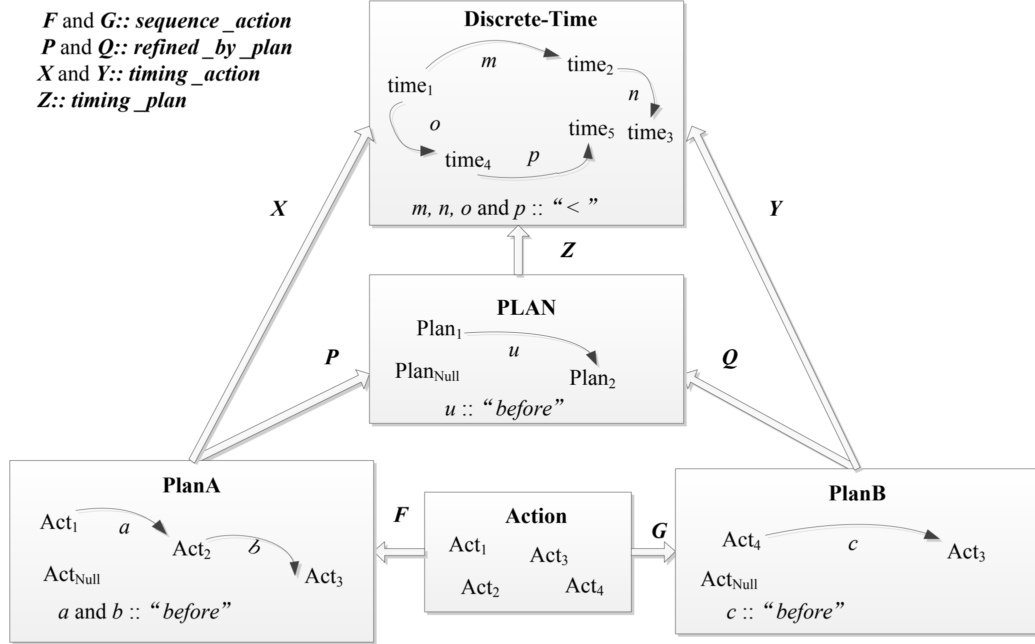


Figure 0.5: Illustration for Plan, PLAN and Discrete-Time example

Action, ***PlanA***, ***PlanB**, **PLAN**, **sequence _action***, and ***refined _by _plan*** have been described in Figure 3.4. ***Discrete-Time*** includes objects: $time_1$, $time_2$, $time_3$, $time_4$ and $time_5$, and morphisms: *m*, *n*, *o*, and *p* are of type "<". Functor ***timing _action*** gives a rule of mapping objects: Act_1, Act_2 and Act_3 of ***PlanA*** to objects: $time_1$, $time_2$ and $time_3$ of ***Discrete-Time**,* and mapping morphisms: *a* and *b* of ***PlanA*** to morphisms: *m* and *n* of ***Discrete-Time***. Similar to ***PlanA*** with functor ***TA***, ***PlanB***'s objects and morphism: Act_4, Act_3 and *c* can be mapped to $time_4$, $time_3$ and *p* in ***Discrete-Time***. Functor ***timing _plan*** gives a rule of mapping objects: $Plan_1$ and $Plan_2$ of ***PLAN*** to objects: $time_1$ and $time_4$ of ***Discrete-Time**,* and mapping morphism: *u* of ***PLAN*** to morphism: *o* of ***Discrete-Time***.

- "*sequence _action"* representation (***F*** and ***G***)

$$\boldsymbol{F}\ (Act_1) = \boldsymbol{PlanA}.\ Act_1$$

$$\boldsymbol{F}\ (Act_2) = \boldsymbol{PlanA}.\ Act_2$$

$$\boldsymbol{F}\ (Act_3) = \boldsymbol{PlanA}.\ Act_3$$

$$\boldsymbol{G}\ (Act_4) = \boldsymbol{PlanB}.\ Act_4$$

$$\boldsymbol{G}\ (Act_3) = \boldsymbol{PlanB}.\ Act_3$$

- "*refined _by _plan"* representation (***P*** and ***Q***)

$$\boldsymbol{P}\ (\boldsymbol{PlanA}.\ Act_1) = plan_1$$

$$\boldsymbol{P}\ (\boldsymbol{PlanA}.\ Act_2) = plan_2$$

$$\boldsymbol{P}\ (\boldsymbol{PlanA}.\ Act_3) = plan_2$$

$$\boldsymbol{P}\ (< a >) = < id_{plan1} >$$

$$\boldsymbol{P}\ (< b >) = < id_{plan1} >$$

$$\boldsymbol{Q}\ (\boldsymbol{PlanB}.\ Act_4) = plan_2$$

$$\boldsymbol{Q}\ (\boldsymbol{PlanB}.\ Act_3) = plan_2$$

$$\boldsymbol{Q}\ (< c >) = < id_{plan2} >$$

The above illustration shows that Act_1, Act_2 and Act_3 form a sequence of actions of $plan_1$, and Act_4 and Act_3 form a sequence of actions in $plan_2$. Act_1 and Act_4 are the *trigger actions* (definition 3.2.2) of $plan_1$ and $plan_2$. Morphisms "*a, b* and *c*" indicate that Act_1 occurs earlier than Act_2, Act_2 occurs earlier than Act_3, and Act_4 occurs earlier than Act_3. They are mapped to identity morphisms for $plan_1$ and $plan_2$.

- "*timing _action"* representation (***X*** and ***Y***)

$$\boldsymbol{X}\ (\boldsymbol{PlanA}.\ Act_1) = time_1$$

$$\boldsymbol{X}\ (\boldsymbol{PlanA}.\ Act_2) = time_2$$

$$\boldsymbol{X}\ (\boldsymbol{PlanA}.\ Act_3) = time_3$$

$$\boldsymbol{X}\ (< a >) = < m >$$

$$\boldsymbol{X}\ (< b >) = < n >$$

$$\boldsymbol{Y}\ (\boldsymbol{PlanB}.\ Act_4) = time_4$$

$$\boldsymbol{Y}\ (\boldsymbol{PlanB}.\ Act_3) = time_5$$

$$\boldsymbol{Y}\ (< c >) = < p >$$

The above illustration shows that in ***PlanA***, Act_1 is performed at time $time_1$, Act_2 is $time_2$, and Act_3 is $time_3$, and ***PlanA***'s morphisms "*m* and *n*" indicate that $time_1$ is less than $time_2$ and $time_2$ is less than $time_3$. In ***PlanB***, Act_4 and Act_3 are respectively performed at time $time_4$ and $time_5$, and ***PlanB***'s morphism "*p*" indicates that $time_4$ is less than $time_5$. Relation "*less than*" is denoted by "<" in ***Discrete-Time***.

- "*timing _plan"* representation (***Z***)

$$\boldsymbol{Z}\ (plan_1) = time_1$$

$$\boldsymbol{Z}\ (plan_2) = time_4$$

$$\boldsymbol{Z}\ (< u >) = < o >$$

The above illustration shows that $plan_1$ is triggered at $time_1$, and $plan_2$ is triggered at $time_4$. In Figure 3.6, there are categories ***PLAN*** and ***PLAN'***, and a functor "*self _PLAN*". ***PLAN*** represents agent's default plans and their relations; ***PLAN'*** represents the same agent's plans and their relations after one un-achievable plan has been removed. Functor "*self _PLAN*" provides a way of updating ***PLAN***'s objects and morphisms to ***PLAN'***.

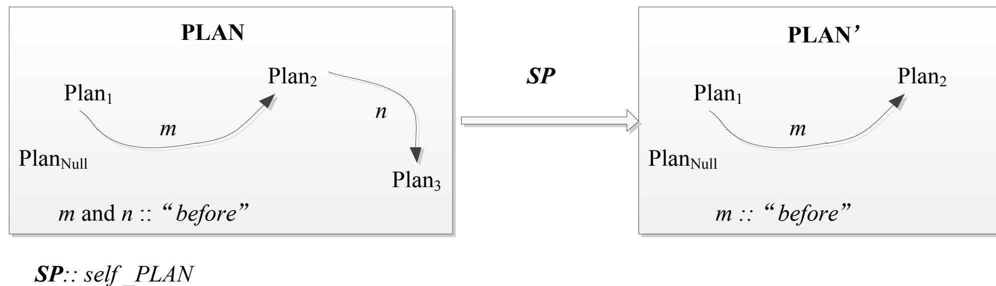


Figure 0.6: Self-update of PLAN

PLAN includes objects: $Plan_1$, $Plan_2$, $Plan_3$ and $Plan_{Null}$, and morphisms "*m* and *n*". ***PLAN'*** contains objects: $Plan_1$, $Plan_2$ and $Plan_{Null}$, and morphism "*m*" from $plan_1$ to $Plan_2$.

- "*self _PLAN"* representation (***SP***):

$$\mathbf{SP}\ (Plan_1) = Plan_1$$

$$\mathbf{SP}\ (Plan_2) = Plan_2$$

$$\mathbf{SP}\ (Plan_3) = Plan_{Null}$$

$$\mathbf{SP}\ (Plan_{Null}) = Plan_{Null}$$

$$\mathbf{SP}\ (< m >) = < m >$$

$$\mathbf{SP}\ (< n >) = < id_{PlanNull} >$$

The above example shows that $Plan_3$ is not achievable anymore for some reasons, such as agent's beliefs are changed. Functor *self _PLAN* provides agent's ***PLAN*** a way to self-updating (removing un-achievable plans). It keeps objects: $Plan_1$, $Plan_2$ and $Plan_{Null}$ in ***PLAN'***, and maps the non-achievable plan $Plan_3$ to $Plan_{Null}$ in ***PLAN'***. It keeps morphism "*m*" between achievable plans $Plan_1 \rightarrow Plan_2$, from ***PLAN*** to ***PLAN'***, and maps morphism "*n*" from an achievable plan to a non-achievable plan: $Plan_2 \rightarrow Plan_3$ to $Plan_{Null}$ identity morphism of ***PLAN'***.

3.2.3. Properties

The category modeling in this section captures some important properties of multi-agent systems such as *action sequentiality* and *plan self-updating*.

Functor ***timing _action*** is a structure-preserving mapping of the actions. Their sequential order relations (which are captured by the morphism "*before*") in ***Plan*** can be mapped into the time objects and their relations "<" in ***Discrete-Time***. Functor ***timing***

_plan is a structure-preserving mapping of the plans. Their sequential order (captured by the morphism "*before*") in ***PLAN*** can be mapped into the time objects and their relations "<" in ***Discrete-Time***. With this property, we are able to prove that a plan starts at the same time that its first action (*triggering action*) is performed, and all the following actions of this plan occur later in time. Let us take the example in Figure 3.5:

$$\boldsymbol{Z}\ (plan_1) = \boldsymbol{Z}\ (\boldsymbol{P}\ (\boldsymbol{PlanA}.\ Act_1)) = \boldsymbol{X}\ ((\boldsymbol{PlanA}.Act_1))$$

$$\boldsymbol{Z}\ (plan_2) = \boldsymbol{Z}\ (\boldsymbol{Q}\ (\boldsymbol{PlanB}.Act_4)) = \boldsymbol{Y}\ ((\boldsymbol{PlanB}.Act_4))$$

$$\boldsymbol{Z}\ (plan_1) < \boldsymbol{X}\ ((\boldsymbol{PlanA}.\ Act_2))$$

$$\boldsymbol{Z}\ (plan_1) < \boldsymbol{X}\ ((\boldsymbol{PlanA}.\ Act_3))$$

$$\boldsymbol{Z}\ (plan_1) < \boldsymbol{Y}\ ((\boldsymbol{PlanB}.\ Act_3))$$

Functor ***self _PLAN*** gives an agent the ability to update its plans, and provides a way for ***PLAN*** to remove its un-achievable plans. As mentioned earlier, plans may not be achievable anymore if their pre-conditions are changed. The achievable plans will be kept as plans in agent's ***PLAN'***, and the non-achievable ones will be thrown to the exception catcher, the $Plan_{Null}$ in ***PLAN'***.

Using category theory, the properties are verified by construction of categories ***Action Plan***, ***PLAN*** and ***Discrete-Time,*** and functors: ***sequence _action***, ***refined _by _plan***, ***timing _action***, ***timing _plan*** **and** ***self _PLAN***.

3.3. Representing Goals

Goals make up the agent's motivational stance and are the driving forces for its actions. Therefore, the representation and handing of goals is one of the main features of agents. In fact, each agent has a set of goals which are dispatched by plans [PB07]. This section

provides categorical definitions for "***GOAL***" and "***Dependencys***", and their relations. With these definitions, we are able to formalize agent's goals and classify them in different levels of priority.

3.3.1. Categorical Representation

Definition 3.3.1 ***GOAL*** *is a category whose objects are goals and morphisms are "depends". The definition of "depends" can be the domain of this morphism has higher or the same priority level than the co-domain. Inside every GOAL, there is a special goal, denoted by* $Goal_{Null}$*. A* $Goal_{Null}$ *stands for an empty object with no morphism from or to other goals.*

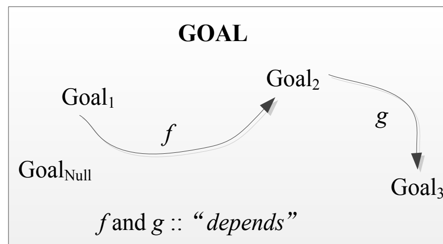


Figure 0.7: Representation of the GOAL category

$Goal_{Null}$ is used to capture exceptions. Figure 3.7 is an example of ***GOAL***, morphisms: $f: goal_1 \rightarrow goal_2$ and $g: goal_2 \rightarrow goal_3$ mean $goal_1$ has higher or the same priority level than $goal_2$, and $goal_2$ has higher or the same priority level than $goal_3$. Morphism $g \circ f: goal_1 \rightarrow goal_3$, the composition of f and g has a correct meaning: $goal_1$ has higher or the same priority level than $goal_3$. Thus, ***GOAL*** satisfies *Associativity* and *Unit* laws (see Chapter 2). Therefore, the validity of the category ***GOAL*** is proved.

Definition 3.3.2 ***Dependency*** *is a category whose objects are integers such as "1", "0", "-1" and "unsigned", and morphisms are bigger or equal to, denoted as "≥". Object "unsigned" doesn't have any relations (morphisms) with other objects.*

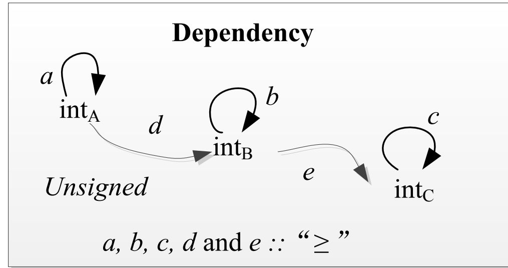


Figure 0.8: Representation of the Dependency category

Figure 3.8 illustrates an example of ***Dependency,*** *which includes objects: "1, 0, -1,* and *unsigned*" *and morphisms: "a, b, c, d and e". The composition of morphisms is meaningful, for example e o d:: "≥": 1 → -1*, the composition of *f* and *g* means that *1* is *"≥"* than *-1*. ***Dependency*** satisfies *Associativity* and *Unit* laws, from which the validity of the category ***Dependency*** follows.

Dependency category is used to set up the order of importance or urgency of different goals. Goals are depended by other goals need to be performed earlier.

Definition 3.3.3 ***assigned _dependency*** *is a functor from* ***GOAL*** *to* ***Dependency****. Functor "assigned _dependency" models the fact that objects (goals) in* ***GOAL*** *can be assigned to corresponding order in* ***Dependency****. And the morphisms in* ***GOAL*** *can be mapped to morphisms ("≥") in category Dependency.*

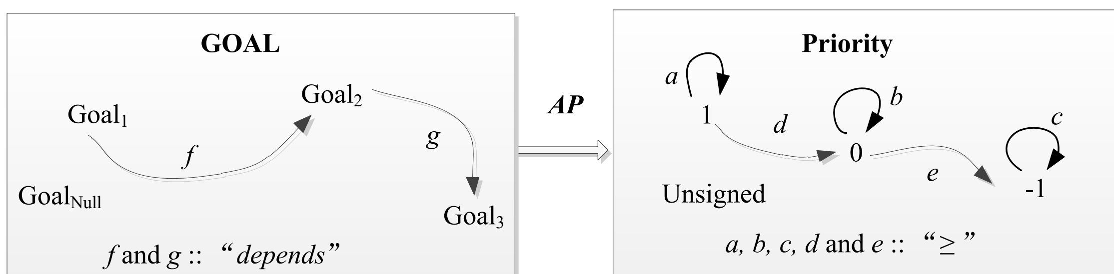


Figure 0.9: Representation of the assigned _depends functor

Figure 3.9 depicts an example of ***assigned _dependency**, which* provides a rule mapping all the objects (*$Goal_1$, $Goal_2$*, *$Goal_3$*, and *$Goal_{Null}$*) of ***GOAL*** to objects (*1, 0, 0* and *unsigned*) of ***Dependency***, and also mapping all the morphisms (*f* and *g*) of ***GOAL*** to morphisms (*d* and *b*) of ***Dependency***.

Definition 3.3.4 ***self _GOAL*** *is a functor from GOAL to itself (similar to **self _PLAN**).* Since agent's beliefs are dynamic and changeable, based on different beliefs, some goals may not be achievable any more. We suppose *GOAL'* is the new category after *GOAL* is translated by this functor. The functor maps achievable goals in *GOAL* to goals in *GOAL';* maps non-achievable goals in *GOAL* to *$Goal_{Null}$* in *GOAL'*; maps *$Goal_{Null}$* in *GOAL* to *$Goal_{Null}$* in *GOAL';* maps morphisms *"higher _ dependency"* from achievable goal to achievable goal in *GOAL* to morphisms *"higher _ dependency"* in *GOAL'*; maps morphisms *"higher _ dependency"* from achievable goal to non-achievable goal, or non-achievable goal to achievable goal, or non-achievable goal to non-achievable goal in *GOAL* to identity morphism of *$Goal_{Null}$* in *GOAL'*. Additionally, if the non-achievable goal is between two achievable goals, a new morphism ("*higher _ dependency*") will be created to link these achievable goals in ***GOAL'***.

3.3.2. Illustration

In this section, we will give some examples of representing the above defined categories. Figure 3.9 illustrates the Definitions: 3.3.1, 3.3.2, and 3.3.3, and Figure 3.10 illustrates the Definition 3.3.4.

In Figure 3.9 there are two categories ***GOAL*** and ***Dependency***, and one functor "***assigned _ dependency***". ***GOAL*** has objects: *$Goal_1$, $Goal_2$*, *$Goal_3$*, and *$Goal_{Null}$*, and morphisms: *f*

and *g*. ***Dependency*** contains objects: *1, 0, -1* and *unsigned*, and morphisms: *a, b, c, d,* and *e.*

- "*assigned _ dependency"* representation (*AD*)

$$\boldsymbol{AD}\ (Goal_1) = 1$$

$$\boldsymbol{AD}\ (Goal_2) = 0$$

$$\boldsymbol{AD}\ (Goal_3) = 0$$

$$\boldsymbol{AD}\ (Goal_{Null}) = unsigned$$

$$\boldsymbol{AD}\ (< f >) = < d >$$

$$\boldsymbol{AD}\ (< g >) = < b >$$

The above illustration encodes the following information: $Goal_1$ depends on $Goal_2$ and $Goal_2$ depends on $Goal_3$, $Goal_{Null}$ is unknown (object "*unsigned*"). The morphisms *f* indicates that $Goal_1$'s priority level in ***Dependency*** is bigger than $Goal_2$'s, and $Goal_2$'s priority level in ***Dependency*** is equal to $Goal_3$'s.

In Figure 3.10, there are two categories ***GOAL*** and ***GOAL'***, and a functor "*self _GOAL*". *GOAL* has objects: $Goal_1$, $Goal_2$, $Goal_3$ and $Goal_{Null}$, and morphisms *f*: $Goal_1 \rightarrow Goal_2$, and *g*: $Goal_2 \rightarrow Goal_3$. ***GOAL'*** contains objects: $Goal_1$, $Goal_2$ and $Goal_{Null}$, and morphisms *f*: $Goal_1 \rightarrow Goal_2$.

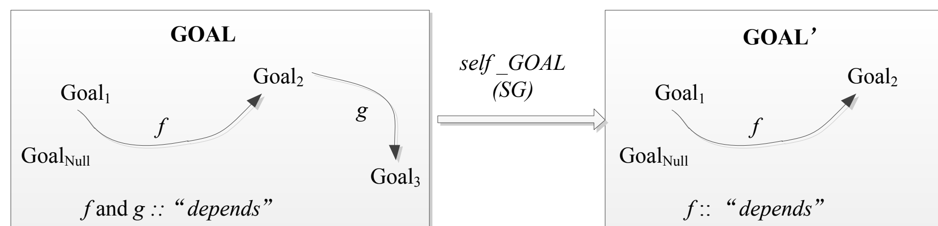


Figure 0.10: Self-update of GOAL

- "*self _GOAL*" representation (*SG*):

$$\boldsymbol{SG}\ (Goal_1) = Goal_1$$

$$\boldsymbol{SG}\ (Goal_2) = Goal_2$$

$$\boldsymbol{SG}\ (Goal_3) = Goal_{Null}$$

$$\boldsymbol{SG}\ (Goal_{Null}) = Goal_{Null}$$

$$\boldsymbol{SG}\ (< f >) = < f >$$

$$\boldsymbol{SG}\ (< g >) = < id_{GoalNull} >$$

The above example shows that $Goal_3$ cannot be achieved anymore for some reasons, such as agent's beliefs are changed. Functor ***self _GOAL*** provides a rule to remove the unachievable goal; it keeps achievable goals: $Goal_1$ and $Goal_2$ in ***GOAL'***, and maps the non-achievable goal $Goal_3$ to $Goal_{Null}$ in ***GOAL'***. It also maps the morphism "*f*" between achievable goals: $goal_1 \rightarrow goal_2$ to the same morphism in ***GOAL'***, and morphism "*g*" from achievable goal to non-achievable goal: $goal_2 \rightarrow goal_3$ to $Goal_{Null}$'s identity morphism in ***GOAL'***.

3.3.3. Properties

The category modeling in this section captures some important properties of multi-agent systems such as *goal dependency* and *goal self-updating*.

Goals can be classified into different levels of dependency by categories ***GOAL*** and ***Dependency*** and their functor ***assigned _ dependency***. Each goal has one corresponding level of dependency, which is denoted by ***Dependency*** objects: "*1*", "*0*", "*-1*" or "*unsigned*". Goals are depended by others should start first.

Functor ***self _GOAL*** gives the agent an ability to update its goals. It provides a way for ***GOAL*** to re-define (remove) its unachievable goals. As mentioned earlier in this

section, goals may not be achievable anymore for some reasons, such as environments are changed. The achievable goals will be kept as the same goals in agent's ***GOAL'*** and the unachievable goals will be trapped into $Goal_{Null}$ in ***GOAL'***.

Using category theory, the properties are verified by construction of categories ***GOAL*** and ***Dependency***, and functors ***assigned_dependency*** and ***self_GOAL***.

3.4. Representing Beliefs

Beliefs represent agent's knowledge or information about environment and itself. Beliefs are built from different information called *facts*, which are organized into different sets denoted as *fact sets*. This section provides definitions for "***BELIEF***" and "***FactSet***" categories, and their relations. With these definitions, we are able to formalize agent's beliefs and guarantee they are consistent within a single agent and in a group of agents (i.e. the system).

3.4.1. Categorical Representation

Definition 3.4.1 ***FactSet*** *is a discrete category where objects are "facts" and the only morphisms are identity morphisms. The facts are information or knowledge about the agent's environments and system.* Based on different usage, facts are classified into different ***FactSet*** categories. Two special ***FactSets*** categories need to be introduced: $\boldsymbol{FactSet_{Base}}$ *and* $\boldsymbol{FactSet_{Null}}$. $\boldsymbol{FactSet_{Base}}$ includes all the facts every other ***FactSet*** has, and $\boldsymbol{FactSet_{Null}}$ contains no facts at all or it's an empty set (see Figure 3.11 as an example). Inside ***FactSet*** (includes $\boldsymbol{FactSet_{Base}}$, except $\boldsymbol{FactSet_{Null}}$), we define a special object, denoted *as* $Fact_{Null}$. $Fact_{Null}$ is a null fact, which doesn't have morphisms. It is used for catching exceptions (see Figure 3.13).

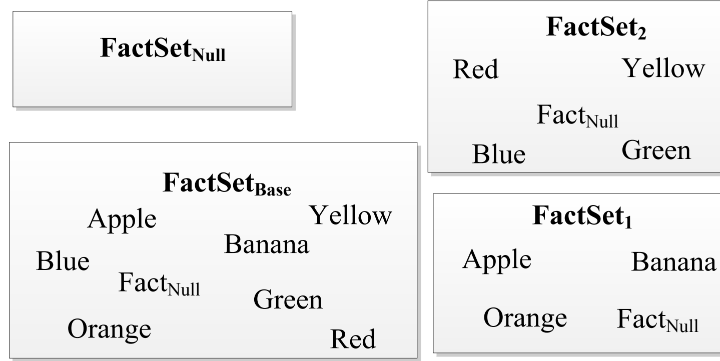


Figure 0.11: Representation of the FactSet category

Definition 3.4.2 ***BELIEF*** *is a category of Sets [Mac71], whose objects are categories* ***FactSets*** *(one* ***$FactSet_{Base}$*** *and one* ***$FactSet_{Null}$*** *are included as default), and the morphisms are "****subset _of****".* Any ***FactSet*** is a subset of ***$FactSet_{Base}$****,* and more formally, every fact within ***FactSet*** can be found in ***$FactSet_{Base}$***. Similarly, ***$FactSet_{Null}$*** has "subset _of" relations to every ***FactSet****.* Using the definitions of initial and terminal objects [Mac71], *NullSet* is the initial object and *BaseSet* is the terminal object in ***BELIEF***.

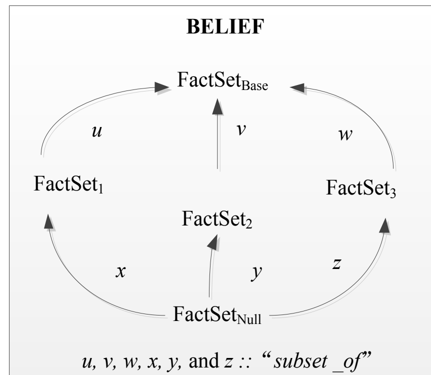


Figure 0.12: Representation of the BELIEF category

The ***BELIEF*** represents an Eiffel's inheritance structure, for example, in Figure 3.12 ***BELIEF*** has objects: $FactSet_1$, $FactSet_2$ and $FactSet_3$ also $FactSet_{Base}$ and $FactSet_{Null}$ as default. It has morphisms $u: FactSet_1 \rightarrow FactSet_{Base}$, $v: FactSet_2 \rightarrow FactSet_{Base}$, $w: FactSet_3 \rightarrow FactSet_{Base}$, $x: FactSet_{Null} \rightarrow FactSet_1$, $y: FactSet_{Null} \rightarrow FactSet_2$ and *z:*

$FactSet_{Null} \rightarrow FactSet_3$. This structure guarantees data's consistence because one agent has only one $FactSet_{Base}$, and all other fact sets are subset of $FactSet_{Base}$.

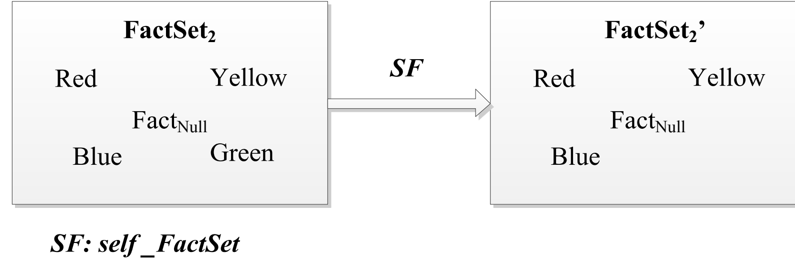
Definition 3.4.3 ***self _FactSet (SF)*** *is a functor from* ***FactSet*** *to itself.* Since agent's beliefs' facts are dynamic and changeable, some facts may not be true or exist anymore. We suppose ***FactSet'*** is the new category after *FactSet* is translated by this functor. *The functor maps non-changed facts in* ***FactSet*** to facts in ***FactSet'***; maps useless facts in ***FactSet*** to $Fact_{Null}$ in ***FactSet'***; and maps $Fact_{Null}$ in ***FactSet*** to $Fact_{Null}$ in ***FactSet'***.

Definition 3.4.4 ***self _BELIEF (SB)*** *is a functor from* ***BELIEF*** *to itself.* Since agent's fact sets are dynamic and changeable, some FactSets may need to be deleted. We suppose ***BELIEF'*** is the new category after ***BELIEF*** is translated by this functor. The functor maps non-changed FactSets in ***BELIEF*** to *FactSets* in ***BELIEF'****;* maps useless *FactSets* in ***BELIEF*** to *NullSet* in ***BELIEF'***; maps $FactSet_{Null}$ in ***BELIEF*** to $FactSet_{Null}$ in ***BELIEF***'; maps $FactSet_{Base}$ in ***BELIEF*** to $FactSet_{Base}$ in ***BELIEF'****;* keeps morphisms from non-changed FactSets to $FactSet_{Base}$ in ***BELIEF*** as in ***BELIEF***'; keeps morphisms from $FactSet_{Null}$ to non-changed in ***BELIEF*** as in ***BELIEF'***; maps morphisms related to useless FactSets in ***BELIEF*** to identity morphism of $FactSet_{Null}$ in ***BELIEF'***.

3.4.2. Illustration

In the following, we show examples of representing the defined category. Figure 3.13 illustrates Definitions: 3.4.1 and 3.4.3, and Figure 3.14 illustrates Definitions: 3.4.2 and 3.4.4.

Figure 3.13 is an example of updating ***$FactSet_2$*** to ***$FactSet_2$'***. ***$FactSet_2$*** includes objects: *Red*, *Yellow*, *Blue*, *Green* and $Fact_{Null}$. ***$FactSet_2$'*** contains objects: *Red*, *Yellow*, *Blue* and $Fact_{Null}$.



SF: self _FactSet

Figure 0.13: Self-update of FactSet

- "*self _FactSet"* representation (*SF*):

$$\boldsymbol{SF}\ (Red) = Red$$

$$\boldsymbol{SF}\ (Yellow) = Yellow$$

$$\boldsymbol{SF}\ (Blue) = Blue$$

$$\boldsymbol{SF}\ (Green) = Fact_{Null}$$

$$\boldsymbol{SF}\ (Fact_{Null}) = Fact_{Null}$$

The above example shows that *Green* will not be considered as a fact of $\boldsymbol{FactSet_2}$. Functor ***SF*** *(self _FactSet)* provides a way to self-updating (removing useless fact). It keeps objects (facts): *Red*, *Yellow*, *Blue* and $Fact_{Null}$, and moves the useless facts *Green* to $Fact_{Null}$ in $\boldsymbol{FactSet_2'}$.

In Figure 3.14, there are two categories ***BELIEF*** and ***BELIEF'***, and a functor "***self _BELIEF***". ***BELIEF*** has objects: $FactSet_1$, $FactSet_2$, $FactSet_3$, $FactSet_{Base}$ and $FactSet_{Null}$, and morphisms u: $FactSet_1 \rightarrow FactSet_{Base}$, v: $FactSet_2 \rightarrow FactSet_{Base}$, w: $FactSet_3 \rightarrow FactSet_{Base}$, x: $FactSet_{Null} \rightarrow FactSet_1$, y: $FactSet_{Null} \rightarrow FactSet_2$ and z: $FactSet_{Null} \rightarrow FactSet_3$. ***BELIEF'*** contains four objects: $FactSet_1$, $FactSet_3$, $FactSet_{Base}$ and $FactSet_{Null}$, and morphisms u: $FactSet_1 \rightarrow FactSet_{Base}$, w: $FactSet_3 \rightarrow FactSet_{Base}$, x: $FactSet_{Null} \rightarrow FactSet_1$, and z: $FactSet_{Null} \rightarrow FactSet_3$.

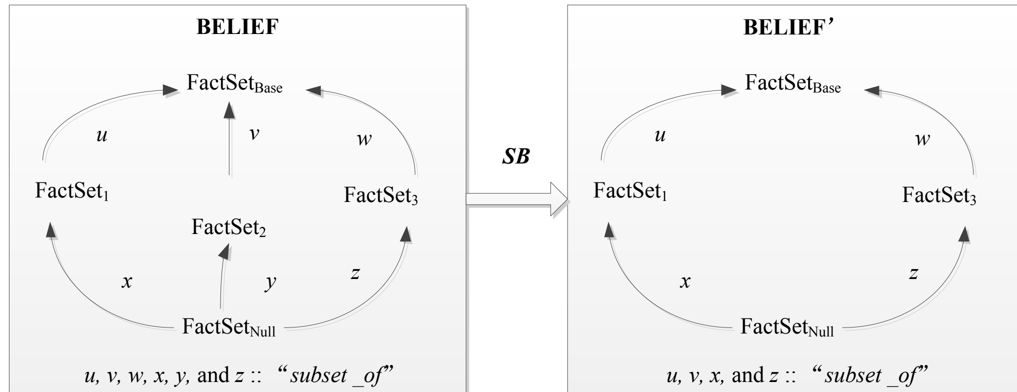


Figure 0.14: Self-update of BELIEF

- "*self _BELIEF*" representation (*SB*):

$$\boldsymbol{SB}\ (FactSet_1) = FactSet_1$$

$$\boldsymbol{SB}\ (FactSet_2) = FactSet_{Null}$$

$$\boldsymbol{SB}\ (FactSet_3) = FactSet_3$$

$$\boldsymbol{SB}\ (FactSet_{Base}) = FactSet_{Base}$$

$$\boldsymbol{SB}\ (FactSet_{Null}) = FactSet_{Null}$$

$$\boldsymbol{SB}\ (< u >) = < u >$$

$$\boldsymbol{SB}\ (< w >) = < w >$$

$$\boldsymbol{SB}\ (< x >) = < x >$$

$$\boldsymbol{SB}\ (< z >) = < z >$$

$$\boldsymbol{SB}\ (< v >) = < id_{FactSetNull} >$$

$$\boldsymbol{SB}\ (< y >) = < id_{FactSetNull} >$$

The above example shows that $FactSet_3$ will not be used anymore for some reasons, such as agent's beliefs are changed. Functor *self _BELIEF* provides the agent a way to self-updating (removing) $FactSet_3$; it keeps useable factSets as they are: $FactSet_1$ and

FactSet₃, and maps the useless factSets $FactSet_2$ to $FactSet_{Null}$ in ***BELIEF'***. It also keeps morphisms u: $FactSet_1 \rightarrow FactSet_{Base}$, w: $FactSet_3 \rightarrow FactSet_{Base}$, x: $FactSet_{Null} \rightarrow FactSet_1$, and z: $FactSet_{Null} \rightarrow FactSet_3$ as they are from ***BELIEF*** to ***BELIEF'***, and maps morphisms v: $FactSet_2 \rightarrow FactSet_{Base}$ and y: $FactSet_{Null} \rightarrow FactSet_2$ to $Plan_{Null}$ identity morphism of ***BELIEF'***.

3.4.3. Properties

The category modeling in this section captures some of the important properties of multi-agent systems such as *data-consistency* and *belief self-updating*.

The category structure ***BELIEF*** shows that every *FactSet* in the same ***BELIEF*** must have a subset relationship to $FactSet_{Base}$, which is the terminal object of ***BELIEF***. Based on the definitions of terminal object in category (Chapter 2), all the elements (facts) within each object (*FactSet*) should be found in $FactSet_{Base}$ of the same ***BELIEF***, and any change of data will cause the same change to *BaseFact* and related *FactSet(s)*.

Functor *self _FactSet* gives an agent the ability to update its *facts*, and provides a way for ***FactSet*** to re-define its *facts*. As we have mentioned above, facts may become not usable anymore given that the environments are changed. These usable facts will be kept as in agent's *FactSet*, and the non-usable facts will be mapped into to the $Fact_{Null}$ in *FactSet*.

Functor *self _BELIEF* gives an agent the ability to update its *factSets*, and provides a way for ***BELIEF*** to re-define its *factSets*. As we have mentioned above, *factSet* may become not usable anymore. These usable *factSets* will be kept as in agent's ***BELIEF***, and the non-usable *factSets* will be mapped into to the $FactSet_{Null}$ in ***BELEF***.

Using category theory, the properties are verified by construction of categories ***FactSet***, ***BELIEF*** and functors ***self _FactSet*** and ***self _BELIEF***.

3.5. Representing Agents

3.5.1. Introduction

An agent is a computer system that is situated in an environment, and designed to perform autonomous actions in this environment in order to meet its objectives [WJ95]. In this section, we will introduce some definitions, which will be used to represent agent by category theory. ***PLAN***, ***GOAL*** and ***BELIEF*** are categories as defined in Sections 3.2, 3.3 and 3.4, and the objective of this section is to relate them together.

3.5.2. Categorical Representation of Plan and Goal

Goals represent the concrete motivations that influence an agent's behavior. The concrete actions an agent may carry out to reach its goals are described in plans. A plan is a procedural recipe describing the actions to take in order to achieve a goal. In BDI systems, each plan must dispatch a goal, but the goal can be a null object. Basically, in an agent, the plans have to dispatch relevant goals.

Definition 3.5.1 ***plan _goal (PG)*** *is a functor from* ***PLAN*** *(definition 3.2.3) to* ***GOAL*** *(definition 3.3.1). The functor "plan _goal" captures the fact that every plan from* ***PLAN*** *category dispatches a goal from* ***GOAL*** *category. Every object (Plan) in* ***PLAN*** *can be mapped to one object (Goal) in* ***GOAL****, and morphisms "before" in* ***PLAN*** *can be mapped to morphisms "depends" in* ***GOAL****.*

The *plan _goal* functor grantees that: one plan can only dispatch one corresponding goal, and different plans can dispatch a same goal.

3.5.3. Illustration of Plan and Goal

Figure 3.15 illustrates the above category definitions. Two categories: ***PLAN*** and ***GOAL,*** and one functor: "*plan _goal*" are represented. ***PLAN*** has five objects: $Plan_1$, $Plan_2$, $Plan_3$ $Plan_4$ and $Plan_{Null}$, and morphisms $a: Plan_1 \rightarrow Plan_2$, $b: Plan_4 \rightarrow Plan_2$, and $c: Plan_2 \rightarrow Plan_3$. ***GOAL*** has four objects: $Goal_1$, $Goal_2$, $Goal_3$ and $Goal_{Null}$, and morphisms $g: Goal_1 \rightarrow Goal_2$ and $k: Goal_2 \rightarrow Goal_3$.

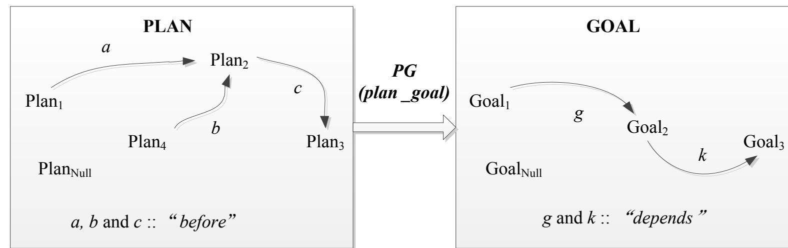


Figure 0.15: Functor plan _goal from PLAN to GOAL

- "*plan _goal*" representation (*PG*)

$$\boldsymbol{PG}\ (Plan_1) = Goal_1$$

$$\boldsymbol{PG}\ (Plan_2) = Goal_2$$

$$\boldsymbol{PG}\ (Plan_3) = Goal_3$$

$$\boldsymbol{PG}\ (Plan_4) = Goal_1$$

$$\boldsymbol{PG}\ (Plan_{Null}) = Goal_{Null}$$

$$\boldsymbol{PG}\ (< a >) = < g >$$

$$\boldsymbol{PG}\ (< b >) = < g >$$

$$\boldsymbol{PG}\ (c >) = < k >$$

This illustration shows ***PLAN***'s objects**:** $Plan_1$ and $Plan_4$ are mapped to the same goal $Goal_1$ of ***GOAL***, $Plan_2$, $Plan_3$ and $Plan_{Null}$ are mapped to goals $Goal_2$, $Goal_3$ and $Goal_{Null}$ of ***GOAL***. It also shows ***PLAN***'s morphisms**:** *a* and *b* are mapped to the same morphism *x*

of ***GOAL***, *c* is mapped to morphism *y* of ***GOAL***. From this example, we can see that functor *plan _goal* represents higher priority goal's plan (such as $Goal_1$ and $Plan_1$) must be performed earlier than lower priority goal's plan (such as $Goal_2$ and $Plan_2$).

3.5.4. Categorical Representation of Plan and Belief

Beliefs represent the agent's knowledge about its environment and itself. They are stored in a belief category, and can be accessed and modified from agent's plans through some fact set interface. Beliefs and plans have been defined as ***BELIEF*** and ***PLAN*** categories, and this section we will introduce a functor to communicate them together.

Definition 3.5.2 ***plan _belief (PB)*** *is a functor from* ***PLAN*** *(definition 3.2.3) to* ***BELIEF*** *(definition 3.4.2). The functor "plan _belief" means agent plans have access to read or write facts from agent's* ***BELIEF****.*

Suppose there are categories ***PLAN*** and ***BELIEF***, and a "*plan _belief*" functor ***PB****: PLAN → BELIEF*, then every "*plan"* in ***PLAN*** can be mapped to one "*FactSet"* (can be $FactSet_{Base}$ or $FactSet_{Null}$) in ***BELIEF***, and all the morphisms in ***PLAN*** are mapped to identity morphism of $FactSet_{Null}$ in ***BELIEF***.

In conclusion, "*plan _belief*" functor formalizes the communication from plans to beliefs. Through this functor, we are able to read and write facts in the agent plans from its belief's factSet.

3.5.5. Illustration of Plan and Belief

This section illustrates the above category definitions. In Figure 3.16 there are two categories: ***PLAN*** and ***BELIEF***, and one functor: "*plan _belief*". ***PLAN*** has objects: $Plan_1$, $Plan_2$, $Plan_3$, $Plan_4$ and $Plan_{Null}$, and morphisms $a: Plan_1 \rightarrow Plan_2$, $b: Plan_4 \rightarrow Plan_2$, and $c: Plan_2 \rightarrow Plan_3$. ***BELIEF*** has objects: $FactSet_1$, $FactSet_2$, $FactSet_3$,

$FactSet_{Base}$ and $FactSet_{Null}$, and morphisms $u: FactSet_1 \rightarrow FactSet_{Base}$, $v: FactSet_2 \rightarrow FactSet_{Base}$, $w: FactSet_3 \rightarrow FactSet_{Base}$, $x: FactSet_{Null} \rightarrow FactSet_1$, $y: FactSet_{Null} \rightarrow FactSet_2$, $z: FactSet_{Null} \rightarrow FactSet_3$, $m: FactSet_4 \rightarrow FactSet_{Base}$, $n: FactSet_5 \rightarrow FactSet_{Base}$, $o: FactSet_{Null} \rightarrow FactSet_4$ and $p: FactSet_{Null} \rightarrow FactSet_5$.

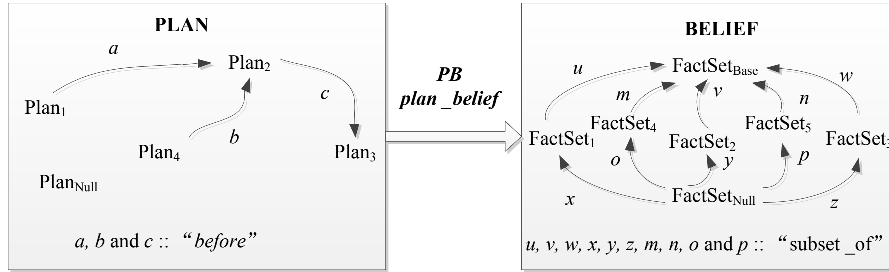


Figure 0.16: Functor plan _belief from PLAN to BELIEF

- "*plan _belief*" representation *(PB)*

$$\boldsymbol{PB}\ (Plan_1) = FactSet_1$$

$$\boldsymbol{PB}\ (Plan_2) = FactSet_2$$

$$\boldsymbol{PB}\ (Plan_3) = FactSet_{Base}$$

$$\boldsymbol{PB}\ (Plan_4) = FactSet_3$$

$$\boldsymbol{PB}\ (Plan_{Null}) = FactSet_{Null}$$

$$\boldsymbol{PB}\ (< a >) = < id_{FactSetNull} >$$

$$\boldsymbol{PB}\ (< b >) = < id_{FactSetNull} >$$

$$\boldsymbol{PB}\ (< c >) = < id_{FactSetNull} >$$

With functor "*plan _belief*" plans are able to access factSets, which are defined as categories containing information or knowledge of the agent's environment. By using category, the property "one plan can only access to one factset" can be captured.

3.5.6. Categorical Representation of Goal and Belief

Beliefs represent the agent's knowledge about its environment and itself. They are stored in a belief base set, and can be accessed from goals by using some fact set interface. Beliefs can be read as pre-conditions by goals, so that the agent is able to justify if the goal is achievable or not. Goals and fact set have been defined as ***GOAL*** and ***BELIEF*** as categories, and this section will introduce a functor to communicate them together.

Definition 3.5.3 ***goal _belief (GB)*** *is a functor from* ***GOAL*** *(Definition 3.3.1) to* ***BELIEF*** *(definition 3.4.2). It means every goal has an access to read facts or knowledge from agent beliefs.* If there are categories ***GOAL*** and ***BELIEF***, and a "*goal _belief*" functor ***GB****:* ***GOAL*** *→* ***BELIEF***, then every object "*Goal*" in ***GOAL*** will be mapped to an object "*FactSet*" in ***BELIEF***, and morphisms "*depends*" in ***GOAL*** will be mapped to identity morphism of $FactSet_{Null}$ in ***BELIEF***. The "*goal _belief*" functor formalizes the communication from goals to beliefs. Through this functor, goals are able to read data from beliefs and justify if they are able to be accomplished.

3.5.7. Illustration of Goal and Belief

In the following, we show examples of representing the above defined category definitions. In Figure 3.17 there are two categories: ***GOAL*** and ***BELIEF***, and functors: "*goal _belief*". ***GOAL*** has three objects: $Goal_1$, $Goal_2$ and $Goal_3$, and morphisms $k: Goal_1 \rightarrow Goal_2$ and $g: Goal_2 \rightarrow Goal_3$. ***BELIEF*** has objects: $FactSet_1$, $FactSet_2$, $FactSet_3$, $FactSet_{Base}$ and $FactSet_{Null}$, and morphisms $u: FactSet_1 \rightarrow FactSet_{Base}$, $v: FactSet_2 \rightarrow FactSet_{Base}$, $w: FactSet_3 \rightarrow FactSet_{Base}$, $x: FactSet_{Null} \rightarrow FactSet_1$, $y: FactSet_{Null} \rightarrow FactSet_2$, $z: FactSet_{Null} \rightarrow FactSet_3$, $m: FactSet_4 \rightarrow FactSet_{Base}$, $n: FactSet_5 \rightarrow FactSet_{Base}$, $o: FactSet_{Null} \rightarrow FactSet_4$ and $p: FactSet_{Null} \rightarrow FactSet_5$.

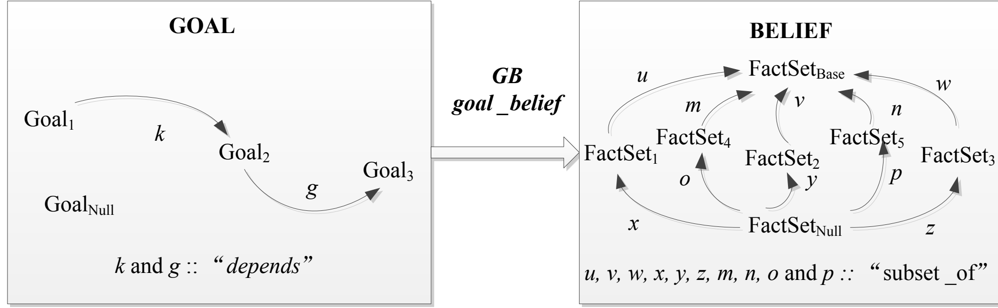


Figure 0.17: Functor goal _belief from GOAL to BELIEF

- "*goal _belief*" representation (*GB*)

$$\boldsymbol{GB}\ (Goal_1) = FactSet_5$$

$$\boldsymbol{GB}\ (Goal_2) = FactSet_2$$

$$\boldsymbol{GB}\ (Goal_3) = FactSet_4$$

$$\boldsymbol{GB}\ (< k >) = < id_{FactSetNull} >$$

$$\boldsymbol{GB}\ (< g >) = < id_{FactSetNull} >$$

GOAL and ***BELIEF*** are communicating using functor "*goal _belief*". Goals are able to access to belief's factSets, which are defined as categories containing information or knowledge of the agent's environment. After having access to their corresponding factSets, goals will update themselves to achievable or non-achievable through "*self _GOAL*" functor. By using category, the property "one goal can only access to one factset" can be captured.

3.5.8. Plan, Goal and Belief Together

The definitions of functors "*plan _goal*" "*plan _belief*" and "*goal _belief*" have been given on the previous sections, and these functors make plans, goals and beliefs communicate.

Agent in Category Representation:

An agent can be represented by categories: ***Action***, ***Plan***, ***PLAN***, ***GOAL***, ***BELIEF*** and ***FactSet****, and functors:* "***plan _goal***" "***plan _belief***" "***goal _belief***", "***refined _by _plan***" and "***sequence _action***".

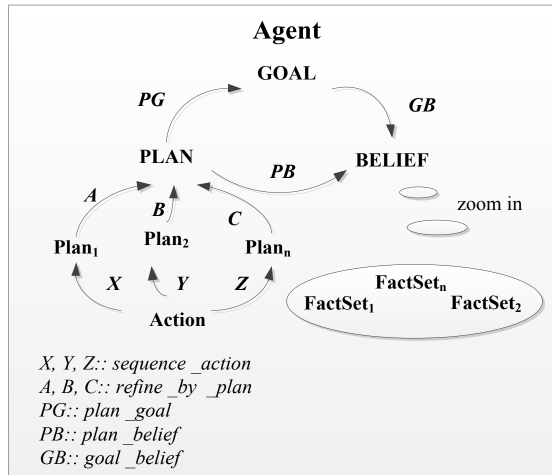


Figure 0.18: Representation of the Agent category

The Figure 3.18 shows that after a plan in ***PLAN*** is triggered, it will dispatch a corresponding goal from ***GOAL*** through "*plan _goal*" functor. Then this goal communicates with its related factSet from ***BELIEF***, which helps justify if the goal is achievable. If the goal is not achievable, functor "*self _Goal*" helps the goal update itself so that it removes non-achievable goal from ***GOAL***, and functor "*self _Plan*" helps the plan update itself so that it removes non-achievable plan from ***PLAN***. If the goal is achievable, plan will be performed continually. Based on different cases, the plan has ability to read or write fact values through ***BELIEF***'s factSet.

Agent Properties in Category Representation:

Each agent has only one $FactSet_{Base}$ in ***BELIEF***, and every *FactSets* is used as a subset of $FactSet_{Base}$, and this design guarantees the consistency of agent's data information and knowledge (see Section 3.4.3).

Each agent's goal has a relationship with ***Dependency*** category (Definition 3.3.2), by which agent's goals are classified as different levels of priority. ***Dependency*** guides the agent to decide which goal should be worked on first if multiple plans are triggered at the same time (see Section 3.3.4).

Each agent's action and plan have a relationship with ***Discrete Time*** category. ***Discrete Time*** is extremely useful to guarantee that *actions* of ***Plan*** and *plans* of ***PLAN*** occur in correct time order (see Section 3.2.4).

PLAN, ***GOAL***, ***BELIEF*** and ***FactSet*** have "*self* _*" functors, which allow each category to have ability to update (i.e. remove) their objects, For example, remove unreached goals from ***GOAL***, or remove unachievable plans in ***PLAN***.

3.6. Representing Multi-Agent Systems

We have defined agent's plans, goals, beliefs and their relationships by category theory representation. In this section, we use category theory to represent multi-agent systems. A system is called multi-agent system (MAS) if there are multiple intelligent agents interacting to each other. The interactions can be described as external trigger event messages passing. Agent's plan is defined as a sequence of actions (see Section 3.2), and the first action is to receive trigger event messages. The trigger event messages can be internal or external messages where the internal messages are sent by the agent itself and the external ones are sent by other agents.

3.6.1. Categorical Representation of MAS

MAS is a category whose objects are "*agents"* and morphisms are "*communicate*". The meaning of "*communicate*" is that one agent has activities of conveying information to another agent, and "*communicate*" can be differentiated by types. For example, from objects $Agent_1$ to $Agent_2$ there is a "*communicate*" morphism $f: Agent_1 \rightarrow Agent_2$, which represents the fact that $Agent_2$ is receiving trigger message(s) from $Agent_1$. In other words, $Agent_2$ has a plan trigged by $Agent_1$. Composition operation is satisfied. Suppose $Agent_3$ is another object in the same ***MAS*** and a morphism $g: Agent_2 \rightarrow Agent_3$, then morphism $g \circ f: Agent_1 \rightarrow Agent_3$ is the composition of f and g, which represents the fact that $Agent_1$ is able to communicate with $Agent_3$. ***MAS*** category also satisfies *Associativity* and *Unit* lows. ***MAS*** is stated as a valid category by the above axioms. Using this category, we are able to have an overall idea about agents' relationships in a system, such as which agents have the ability to communicate directly, which agents need other agents to delegate messages, and which type of communication is taking place between agents.

We can use our RAS [OK06] [KO08] based multi-agent system as an instance. *System manager agent* is the most essential part that acts as a brain for the overall system. It governs the entire system in terms of monitoring and controlling the other agents' actions. This agent also has the most global view, which allows it to communicate with any other agent whining the system. It guarantees the overall system running correctly. *Supervisor agent* exists within each group. It is the group leader that manages the group. It plays a similar role to the *system manager's role*, but with limited power and localized view of the entire system. Within multi-agent society, *worker* agents are the mass. Unlike *supervisor agent* or *system manager agent*, they perform actual works, obey orders and

report events. Since RAS based multi-agent system is a layered framework, each tier only can communicate with the same tier or the tier immediately above or below.

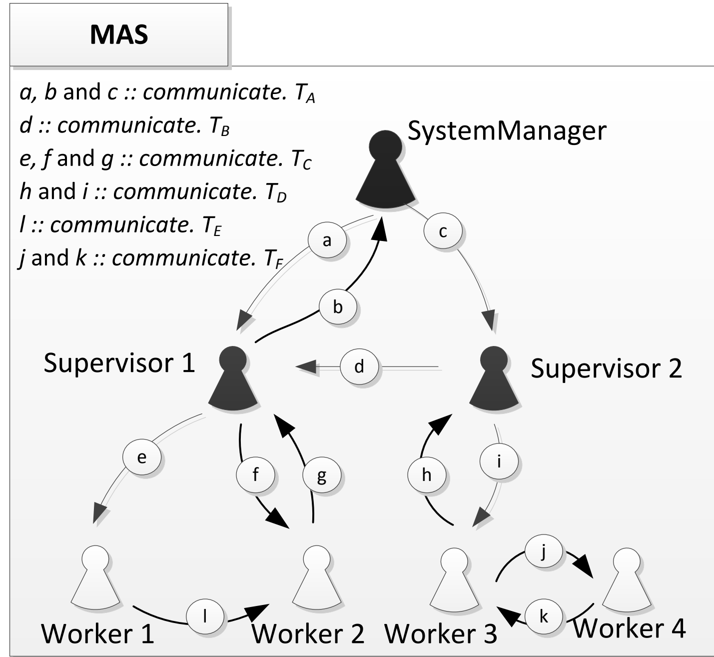


Figure 0.19: MAS category example

Figure 3.19 is a category representation for a multi-agent system, which models a RAS. It exhibits a high abstract view of the overall system. The basic component objects for the system are: *system manager agent*, *supervisor agent* and *worker agent*, and basic communication types are: T_A between *system manager* and *supervisors*, T_B between *supervisor*$_1$ and *supervisor*$_2$, T_C between *supervisor*$_1$ and its *works*, T_D between *supervisor*$_2$ and its *works*, T_E between *work*$_1$ and *work*$_2$, and T_F between *work*$_3$ and *work*$_4$.

3.6.2. Repository Agent

Within each RAS-based multi agent system, there exists one special agent. This agent is used to store the entire multi agent system information and it is in a position as a system persistent storage. Repository agent contains copies of every agent's information, such as goals, plans and beliefs.

3.6.3. Repository Type

Repository Type is a type category (see Chapter 2) whose objects are categories that represent the types of agents, and whose morphisms are "*communicate"*, which represents the types of communication channels from agents to agents. For example, one ***Repository Type*** includes objects $Type_1$ and $Type_2$ and morphism $f\colon Type_1 \rightarrow Type_2$. It means agents can be of $Type_1$ or $Type_2$, and agents of $Type_2$ have channels that are open to agents of $Type_1$. In other words, agents of $Type_1$ have the ability to access to or communicate with agents of $Type_2$. Since there is no morphism $Type_2 \rightarrow Type_1$, agents of $Type_2$ do not have the ability to access to or communicate with agents of $Type_1$.

Zoom into each category, it includes objects: *ActionType*, *PlanType*, *PLANType*, *GOALType*, *FactSetType* and *BELIEFType*. **Type*'s objects represent the types of objects of agents' ***Action, Plan, PLAN, GOAL, FactSet*** and ***BELIEF***, and **Type*'s morphisms represent the types of morphisms between objects within ***Action, Plan, PLAN, GOAL, FactSet*** and ***BELIEF***.

3.6.4. MAS and Repository Type

From ***MAS*** to ***Repository Type***, there exists a functor (***F***) with some additional properties. In general a functor only maps objects to objects, and morphisms to morphisms, but since each object is defined as a category in this thesis, we need this functor with a special property to zoom into each object and do a mapping too. ***F*** maps every object (*agent*) of ***MAS*** to object (*category*) of ***Repository Type***, and it also maps every morphism (*communicate*) to morphism (*communicate type*). The additional properties of ***F*** describe relations between ***MAS*** and ***Repository Type*** objects, which are shown in the following example (Figure 3.20) and represented by Table 3.1.

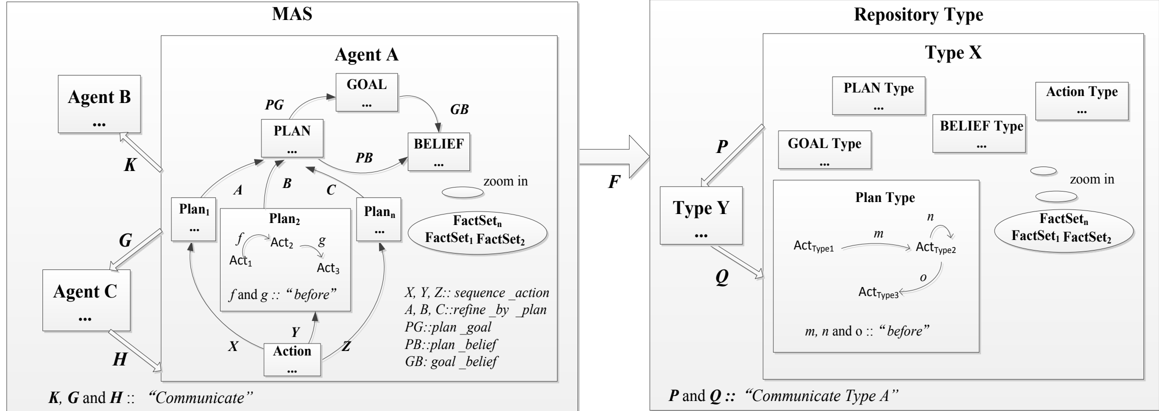


Figure 0.20: MAS to Repository Type

Suppose ***MAS*** (***M***) has three objects (*agents*): *Agent A, Agent B* and *Agent C*, and morphisms (*communicate*): *K: A →B, G: A→ C* and *H: C→ A* (Figure 3.20). ***Repository Type*** (RT) has two objects (*types of agent*) *Type X* and *Type Y*, and morphism (*types of morphisms*) *P: Type X→ Type Y* and *Q: Type Y→ Type X*. As we have defined: *Agent A, Agent B* and *Agent C* are agents specified using ***Action***, ***Plan***, ***PLAN***, ***GOAL***, ***FactSet*** and ***BELIEF*** **categories**. In addition, ***F*** provides a rule to transfer objects and morphisms within *Agent A, Agent B* and *Agent C'* s ***Action***, ***Plan***, ***PLAN***, ***GOAL*** and ***BELIEF*** to objects and morphisms within *Type X* and *Type Y*'s ***ActionType***, ***PlanType***, ***PLANType***, ***GOALType*** and ***BELIEFType***. Here we will only show *Agent A*'s ***Plan***$_2$ in Table 3.1 and more details will be given in Chapter 4, Section 4.2.

	Rep Type										**Counter**
MAS	Type X							Type Y			
Agent A	Plan Type						…	…			
Plan$_2$	Act$_{Type1}$	Act$_{Type2}$	Act$_{Type3}$	m	n	o	…	…	…	…	
Act$_1$	1	0	0	0	0	0					1

Act_2	0	1	0	0	0	0					1
Act_3	0	1	0	0	0	0					1
F	0	0	0	1	0	0					1
G	0	0	0	0	1	0					1

Table 0.1: Additional properties of F

The left side of Table 3.1 shows objects and morphisms of each *Agent A*'s categories in ***MAS***. For example, Act_1 means *action* object Act_1 of *Agent A*'s $Plan_2$ category; and f means *before* morphism: $Act_1 \rightarrow Act_2$ of *Agent A*'s $Plan_2$ category. In the middle, the table shows objects and morphisms of each type Agent's categories in ***Repository Type***. We use "1" if there is a match from object or morphism to a type; otherwise "0" is marked. On the right side, *counter* represents the sum of marked numbers in the same row. Counter equals to "1" is the only acceptable result, which shows that one object or morphism is only allowed to be of one type.

Chapter 4: Fault-Tolerance Properties in Multi-Agents System Categorical Model

In this chapter, we will introduce some Fault-Tolerance properties with category theory for multi-agent systems, which have been defined and illustrated in Chapter 3. We use the robotic case study discussed in Chapter 2.

4.1. A Categorical Model for Robotic Case Study

In this chapter, the following agents from the robotic case study will be used to represent fault-tolerance properties. They are repository agent ***Repository***, and repository type, ***Repository Type***, supervisor agent ***Supervisor***, and carry agents $\boldsymbol{Carry_1}$ and $\boldsymbol{Carry_2}$.

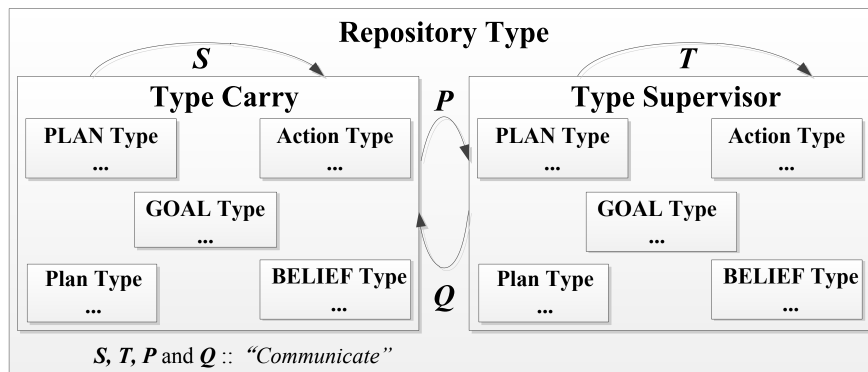


Figure 0.1: Repository Type categories in case study

Repository Type has objects that represent agent types, such as *Supervisor* and *Carry*. It also has "*communicate*" morphisms that represent communication channels from one type of agent to another, such as *Carry* → *Supervisor*, *Supervisor* → *Carry, Supervisor*

→ *Supervisor* and *Carry* → *Carry*. Each object (*type of agent*) within ***Repository Type*** contains five type categories: *ActionType*, *PlanType*, *PLANType*, *GOALType* and *BELIEFType* (see Chapter 3 for the definition of type category). Figure 4.1 illustrates these categories.

Figure 4.2 shows an example of *Type Carry*. *Action Type* contains objects representing the following actions: $Act_{Trigger}$, Act_{Move}, Act_{Load}, and Act_{Unload}. *Plan Type* includes objects representing the following actions: $Act_{Trigger}$, Act_{Move}, Act_{Load}, and Act_{Unload}, and morphisms $a: Act_{Trigger} \rightarrow Act_{Move}$, $b: Act_{Move} \rightarrow Act_{Load}$, $c: Act_{Load} \rightarrow Act_{Move}$, $d: Act_{Move} \rightarrow Act_{Unload}$, and $e: Act_{Trigger} \rightarrow Act_{Unload}$. *PLAN Type* includes objects representing the following plans: $Plan_{CarryOre}$ and $Plan_{Move}$, and morphisms $f: Plan_{CarryOre} \rightarrow Plan_{CarryOre}$ and $g: Plan_{Move} \rightarrow Plan_{Move}$. *GOAL Type* contains objects representing the following goals: $Goal_{CarryOre}$ and $Goal_{Move}$, and morphisms $m: Goal_{CarryOre} \rightarrow Goal_{CarryOre}$ and $n: Goal_{Move} \rightarrow Goal_{Move}$. *BELIEF Type* includes objects representing the following fact sets: $FactSet_{CarryOre}$, $FactSet_{MoveArea}$, $FactSet_{Base}$ and $FactSet_{Null}$, and morphisms $h: FactSet_{CarryOre} \rightarrow FactSet_{Base}$, $i: FactSet_{Null} \rightarrow FactSet_{CarryOre}$, $j: FactSet_{MoveArea} \rightarrow FactSet_{Base}$, and $k: FactSet_{Null} \rightarrow FactSet_{MoveArea}$.

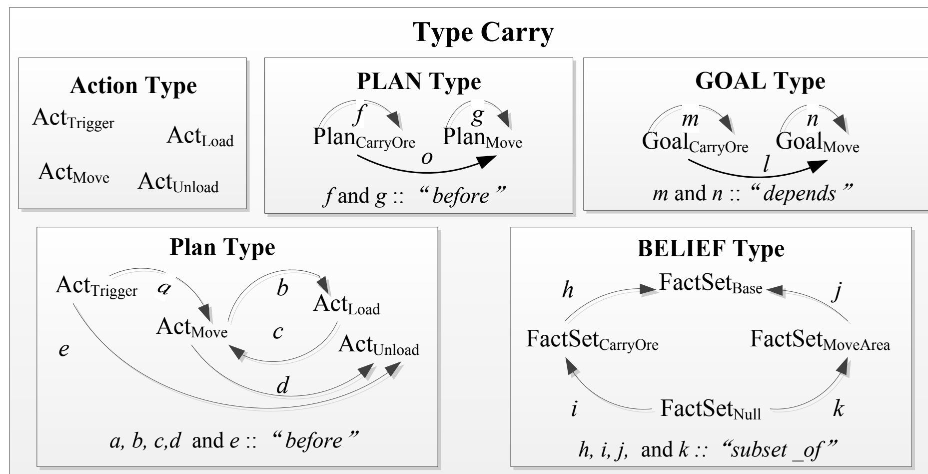


Figure 0.2: Type carry agent

Repository Agent stores copies of each agent's categories, which are useful for restarting damaged agents (This aspect is detailed in Section 4.3).

$\boldsymbol{Carry_1}$ (Figure 4.3) is defined by objects which are categories: $\boldsymbol{Action_1}$, $\boldsymbol{Plan_{1_A}}$, $\boldsymbol{Plan_{1_B}}$, $\boldsymbol{PLAN_1}$, $\boldsymbol{GOAL_1}$, $\boldsymbol{FactSet_{1_A}}$, $\boldsymbol{FactSet_{1_B}}$ and $\boldsymbol{BELIEF_1}$, and morphisms "*sequence _action*", "*refined _by _plan*", "*plan _goal*", "*goal _ belief*" and "*plan _belief*". Where $\boldsymbol{Action_1}$ has the objects: $Act_{StartCarry}$, $Act_{LoadOre}$, $Act_{MoveToTargetA}$, $Act_{MoveToTargetB}$ and $Act_{MoveToBase}$. $\boldsymbol{Plan_{1_A}}$ has the objects: $Act_{StartCarry}$, $Act_{LoadOre}$, $Act_{MoveToTargetA}$, $Act_{MoveToBase}$ and Act_{Null}, and morphisms: l_{1_A}: $Act_{StartCarry} \rightarrow Act_{MoveToTargetA}$, s_{1_A}: $Act_{MoveToTargetA} \rightarrow Act_{LoadOre}$ and t_{1_A}: $Act_{LoadOre} \rightarrow Act_{MoveToBase}$. $\boldsymbol{Plan_{1_B}}$ has the objects: $Act_{StartCarry}$, $Act_{LoadOre}$, $Act_{MoveToTargetB}$, $Act_{MoveToBase}$ and Act_{Null}, and morphisms: l_{1_B}: $Act_{StartCarry} \rightarrow Act_{MoveToTargetB}$, s_{1_B}: $Act_{MoveToTargetB} \rightarrow Act_{LoadOre}$ and t_{1_B}: $Act_{LoadOre} \rightarrow Act_{MoveToBase}$. $\boldsymbol{PLAN_1}$ includes the objects: $Plan_{CarryOreFromTargetA}$ and $Plan_{CarryOreFromTargetB}$, and morphisms: p_1: $Plan_{CarryOreFromTargetA} \rightarrow Plan_{CarryOreFromTargetA}$, o_1: $Plan_{CarryOreFromTargetA} \rightarrow Plan_{CarryOreFromTargetB}$ and q_1: $Plan_{CarryOreFromTargetB} \rightarrow Plan_{CarryOreFromTargetB}$. $\boldsymbol{GOAL_1}$ contains the objects: $Goal_{CarryOreFromTargetA}$ and $Goal_{CarryOreFromTargetB}$, and morphisms: i_1: $Goal_{CarryOreFromTargetA} \rightarrow Goal_{CarryOreFromTargetA}$, k_1: $Goal_{CarryOreFromTargetA} \rightarrow Goal_{CarryOreFromTargetB}$ and j_1: $Goal_{CarryOreFromTargetB} \rightarrow Goal_{CarryOreFromTargetB}$. $\boldsymbol{BELIEF_1}$ contains objects: $FactSet_{1_A}$, $FactSet_{1_B}$, $FactSet_{Base}$ and $FactSet_{Null}$. It also has the morphisms u_1: $FactSet_{1_A} \rightarrow FactSet_{Base}$, v_1: $FactSet_{1_B} \rightarrow FactSet_{Base}$, x_1: $FactSet_{Null} \rightarrow FactSet_{1_A}$ and y_1: $FactSet_{Null} \rightarrow FactSet_{1_B}$. "***Zoom In***" is not a functor, it substitutes the objects $FactSet_{1_A}$ and $FactSet_{1_B}$ in $\boldsymbol{BELIEF_1}$ with their corresponding content. $\boldsymbol{FactSet_{1_A}}$ contains objects: *targetALocation, baseLocation, targetAOreAmount* and

$Fact_{Null}$. ***FactSet$_{1_B}$*** contains objects: *targetBLocation, baseLocation, targetBOreAmount* and $Fact_{Null}$.

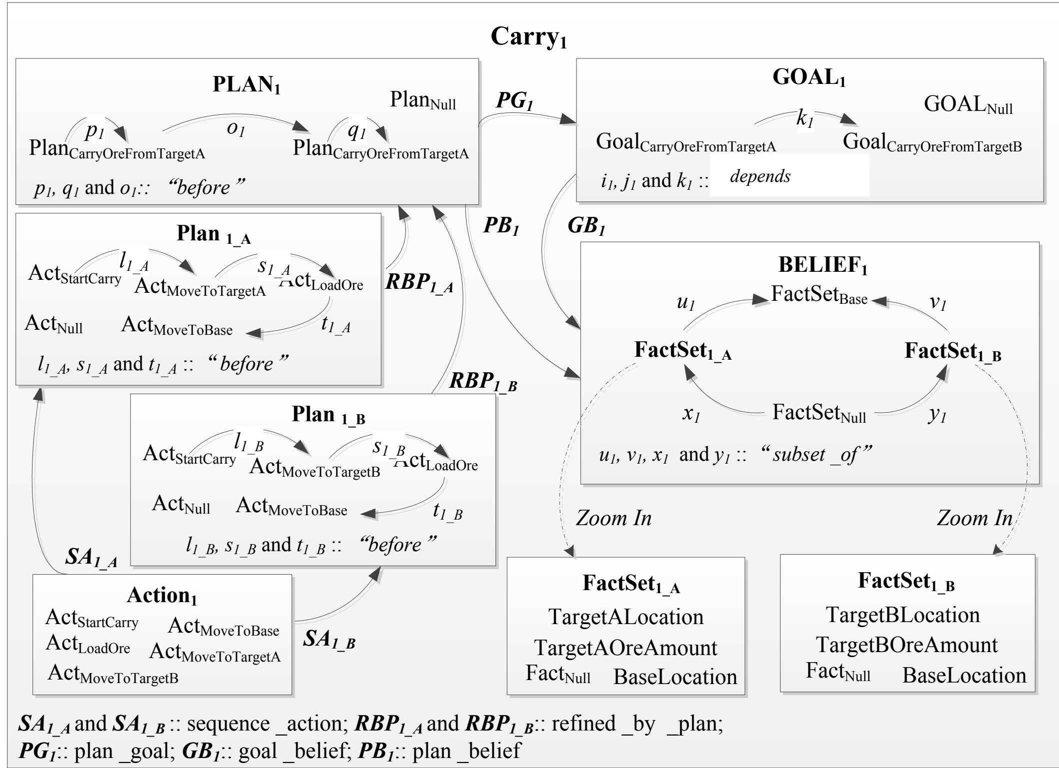


Figure 0.3: Carry$_1$ agent

With functor (***F***) and its additional properties (See Section 3.6.4 in Chapter 3) objects in ***Action$_1$***, ***Plan$_{1_A}$***, ***Plan$_{1_B}$, PLAN$_1$***, ***GOAL$_1$*** and ***BELIEF*** can be one to one mapped to ***Type Carry*'s** objects in ***Action Type***, ***Plan Type***, ***PLAN Type***, ***GOAL Type*** and ***BELIEF Type***. Morphisms within ***Action$_1$***, ***Plan$_{1_A}$***, ***Plan$_{1_B}$, PLAN$_1$***, ***GOAL$_1$*** and ***BELIEF*** can be one to one mapped to ***Type Carry*'s** objects in ***Action Type***, ***Plan Type***, ***PLAN Type***, ***GOAL Type*** and ***BELIEF Type*** (See Tables 4.1 to 4.6).

	Rep Type	**Counter**
MAS	*Type Carry*	
Carry$_1$ (Agent)	Action Type	

$Action_1$	$Act_{Trigger}$	Act_{Move}	Act_{Load}	Act_{Unload}	
$Act_{StartCarry}$	1	0	0	0	1
$Act_{LoadOre}$	0	0	1	0	1
$Act_{MoveToTarget}$	0	1	0	0	1
$Act_{MoveToBase}$	0	1	0	0	1

Table 0.1: F Additional properties map $Action_1$ to Action Type

	Rep Index										**Counter**
MAS	*Type Carry*									
$Carry_1$	Plan Type									
$Plan_{1\text{-}A}$	$Act_{Trigger}$	Act_{Move}	Act_{Load}	Act_{Unload}	*a*	*b*	*c*	*d*	*e*	
$Act_{StartCarry}$	1	0	0	0	0	0	0	0	0	1
$Act_{LoadOre}$	0	0	1	0	0	0	0	0	0	1
$Act_{MoveToTargetA}$	0	1	0	0	0	0	0	0	0	1
$Act_{MoveToBase}$	0	1	0	0	0	0	0	0	0	1
l_{1_A}	0	0	0	0	1	0	0	0	0	1
s_{1_A}	0	0	0	0	0	1	0	0	0	1
t_{1_A}	0	0	0	0	0	0	1	0	0	1

Table 0.2: F Additional properties map $Plan_{1_A}$ to Plan Type

	Rep Index										**Counter**
MAS	*Type Carry*									
$Carry_1$	Plan Type									
$Plan_{1\text{-}B}$	$Act_{Trigger}$	Act_{Move}	Act_{Load}	Act_{Unload}	*a*	*b*	*c*	*d*	*e*	

$Act_{StartCarry}$	1	0	0	0	0	0	0	0	0	1
$Act_{LoadOre}$	0	0	1	0	0	0	0	0	0	1
$Act_{MoveToTargetB}$	0	1	0	0	0	0	0	0	0	1
$Act_{MoveToBase}$	0	1	0	0	0	0	0	0	0	1
l_{1_B}	0	0	0	0	1	0	0	0	0	1
s_{1_B}	0	0	0	0	0	1	0	0	0	1
t_{1_B}	0	0	0	0	0	0	1	0	0	1

Table 0.3: F Additional properties map $Plan_{1_B}$ to Plan Type

	Rep Index					**Counter**
MAS	*Type Carry*					
$Carry_1$	PLAN Type					
$PLAN_1$	$Plan_{CarryOre}$	$Plan_{Move}$	*f*	*g*	*o*	
$Plan_{CarryOreFromTargetA}$	1	0	0	0	0	1
$Plan_{CarryOreFromTargetB}$	1	0	0	0	0	1
p_1	0	0	1	0	0	1
q_1	0	0	1	0	0	1
o_1	0	0	1	0	0	1

Table 0.4: F Additional properties map $PLAN_1$ to PLAN Type

	Rep Index					**Counter**
MAS	*Type Carry*					
$Carry_1$	GOAL Type					
$GOAL_1$	$Goal_{CarryOre}$	$Goal_{Move}$	*m*	*n*	*l*	

$Goal_{CarryOreFromTargetA}$	1	0	0	0	0	1
$Goal_{CarryOreFromTargetB}$	1	0	0	0	0	1
k_1	0	0	1	0	0	1

Table 0.5: F Additional properties map $GOAL_1$ to GOAL Type

	Rep Index								**Counter**
MAS	*Type Carry*								
$Carry_1$	BELIEF Type								
$BELIEF_1$	$FactSet_{CarryOre}$	$FactSet_{MoveArea}$	$FactSet_{Base}$	$FactSet_{Null}$	*h*	*i*	*j*	*k*	
$FactSet_{Base}$	0	0	1	0	0	0	0	0	1
$FactSet_{Null}$	0	0	0	1	0	0	0	0	1
$FactSet_{1_A}$	1	0	0	0	0	0	0	0	1
$FactSet_{1_B}$	1	0	0	0	0	0	0	0	1
u_1	0	0	0	0	1	0	0	0	0
v_1	0	0	0	0	1	0	0	0	1
x_1	0	0	0	0	0	1	0	0	1
y_1	0	0	0	0	0	1	0	0	1

Table 0.6: F Additional properties map $BELIEF_1$ to BELIEF Type

Similar to ***$Carry_1$***, ***$Carry_2$*** is defined by categories: ***$Action_2$***, ***$Plan_{2_A}$***, ***$Plan_{2_B}$***, ***$Plan_{2_move}$***, ***$PLAN_2$***, ***$GOAL_2$***, ***$FactSet_{2_A}$***, ***$FactSet_{2_B}$***, ***$FactSet_{2_Move}$*** and ***$BELIEF_2$***, and morphisms "*sequence _action*", "*refined _by _plan*", "*plan _goal*", "*goal _ belief*" and "*plan _belief*" (see Figure 4.4).

With functor (***F***) and its additional properties (see Section 3.6.4), objects in ***$Action_2$***, ***$Plan_{2_A}$***, ***$Plan_{2_B}$***, ***$Plan_{2_move}$***, ***$PLAN_2$***, ***$GOAL_2$*** and ***$BELIEF_2$*** can be one to one mapped to

Type Carry*'s** objects in ***Action Type, ***Plan Type***, ***PLAN Type***, ***GOAL Type*** and ***BELIEF Type***. Morphisms within ***Action$_2$***, ***Plan$_{2_A}$***, ***Plan$_{2_B}$***, ***Plan$_{2_move}$***, ***PLAN$_2$***, ***GOAL$_2$*** and ***BELIEF$_2$*** can be one to one mapped to ***Type Carry*'s** objects in ***Action Type***, ***Plan Type***, ***PLAN Type***, ***GOAL Type*** and ***BELIEF Type*** (See table 4.7 to table 4.13)

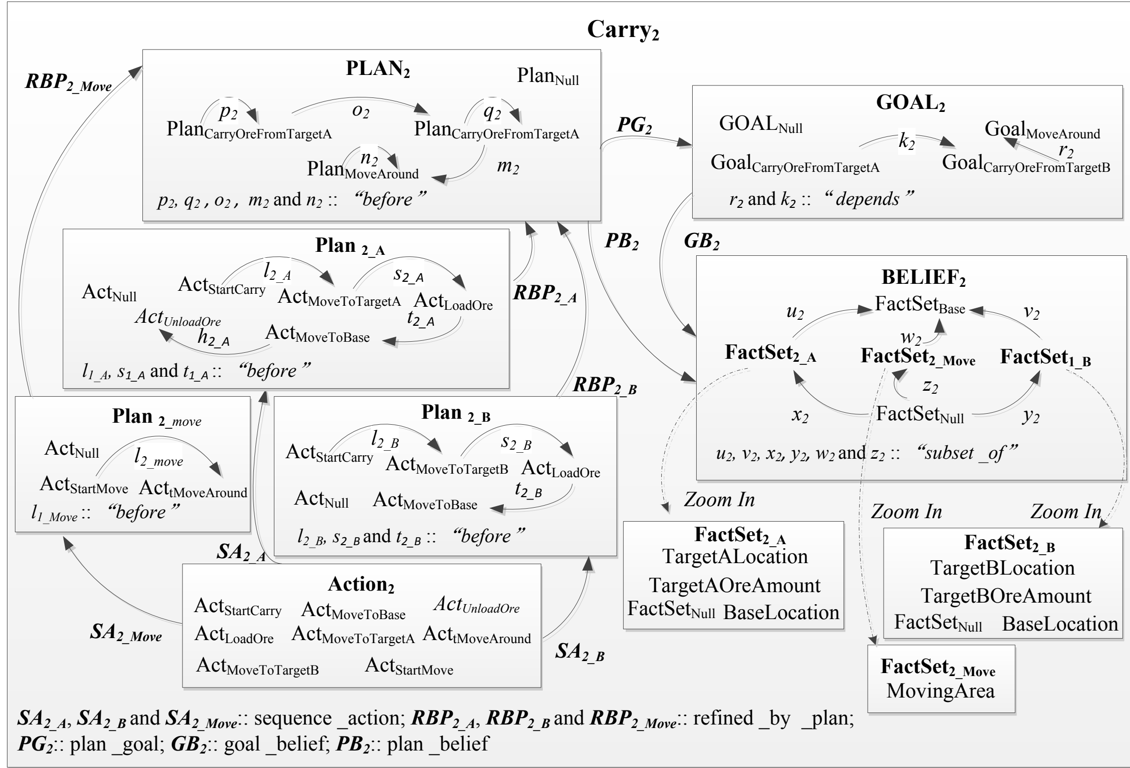


Figure 0.4: Carry$_2$ Agent

	Rep Index				**Counter**
MAS	*Typed Carry*				
Carry$_2$	Action Type				
Action$_2$	Act$_{Trigger}$	Act$_{Move}$	Act$_{Load}$	Act$_{Unload}$	
Act$_{StartCarry}$	1	0	0	0	1
Act$_{LoadOre}$	0	0	1	0	1

$Act_{MoveToTarget}$	0	1	0	0	1
$Act_{MoveToBase}$	0	1	0	0	1
$Act_{StartMove}$	1	0	0	0	1
$Act_{MoveAround}$	0	1	0	0	1
$Act_{UnloadOre}$	0	0	0	1	0

Table 0.7: F Additional properties map $Action_2$ to Action Type

	Rep Index									**Counter**
MAS	*Type Carry*									
$Carry_2$	Plan Type									
$Plan_{12\text{-}A}$	$Act_{Trigger}$	Act_{Move}	Act_{Load}	Act_{Unload}	*a*	*b*	*c*	*d*	*e*	
$Act_{StartCarry}$	0	1	0	0	0	0	0	0	0	1
$Act_{LoadOre}$	1	0	0	0	0	0	0	0	0	1
$Act_{MoveToTargetA}$	1	0	0	0	0	0	0	0	0	1
$Act_{MoveToBase}$	0	1	0	0	0	0	0	0	0	1
$Act_{UnloadOre}$	0	0	0	1	0	0	0	0	0	1
l_{2_A}	0	0	0	0	1	0	0	0	0	1
s_{2_A}	0	0	0	0	0	1	0	0	0	1
t_{2_A}	0	0	0	0	0	0	1	0	0	1
h_{2_A}	0	0	0	0	0	0	0	1	0	1

Table 0.8: F Additional properties map $Plan_{2_A}$ to Plan Type

	Rep Index	**Counter**
MAS	*Type Carry*	

$Carry_2$	Plan Type									
$Plan_{2\text{-}B}$	$Act_{Trigger}$	Act_{Move}	Act_{Load}	Act_{Unload}	a	b	c	d	e	
$Act_{StartCarry}$	0	1	0	0	0	0	0	0	0	1
$Act_{LoadOre}$	1	0	0	0	0	0	0	0	0	1
$Act_{MoveToTargetB}$	1	0	0	0	0	0	0	0	0	1
$Act_{MoveToBase}$	0	0	0	1	0	0	0	0	0	1
l_{2_B}	0	0	0	0	1	0	0	0	0	1
s_{2_B}	0	0	0	0	0	1	0	0	0	1
t_{2_B}	0	0	0	0	0	0	1	0	0	1

Table 0.9: F Additional properties map $Plan_{2_B}$ to Plan Type

	Rep Index									**Counter**
MAS	*Type Carry*									
$Carry_2$	Plan Type									
$Plan_{2\text{-}Move}$	$Act_{Trigger}$	Act_{Move}	Act_{Load}	Act_{Unload}	a	b	c	d	e	
$Act_{StartMove}$	1	1	0	0	0	0	0	0	0	1
$Act_{MoveAround}$	0	1	0	0	0	0	0	0	0	1
l_{2_Move}	0	0	0	0	1	0	0	0	0	1

Table 0.10: F Additional properties map $Plan_{2_Move}$ to Plan Type

	Rep Index					**Counter**
MAS	*Type Carry*					
$Carry_2$	PLAN Type					
$PLAN_2$	$Plan_{CarryOre}$	$Plan_{Move}$	f	g	o	

$Plan_{CarryOreFromTargetA}$	1	0	0	0	0	1
$Plan_{CarryOreFromTargetB}$	1	0	0	0	0	1
$Plan_{MoveAround}$	0	1	0	0	0	1
p_2	0	0	1	0	0	1
q_2	0	0	1	0	0	1
o_2	0	0	1	0	0	1
m_2	0	0	0	0	1	1
n_2	0	0	0	1	0	1

Table 0.11: F Additional properties map $PLAN_2$ to PLAN Type

	Rep Index								**Counter**
MAS	*Type Carry*								
$Carry_2$	BELIEF Type								
$BELIEF_2$	$FactSet_{CarryOre}$	$FactSet_{MoveArea}$	$FactSet_{Base}$	$FactSet_{Null}$	*h*	*i*	*j*	*k*	
$FactSet_{Base}$	0	0	1	0	0	0	0	0	1
$FactSet_{Null}$	0	0	0	1	0	0	0	0	1
$FactSet_{2_A}$	1	0	0	0	0	0	0	0	1
$FactSet_{2_B}$	1	0	0	0	0	0	0	0	1
$FactSet_{2_Move}$	0	1	0	0	0	0	0	0	1
u_1	0	1	0	0	0	0	0	0	1
v_1	0	0	0	0	1	0	0	0	1
x_1	0	0	0	0	0	1	0	0	1
y_1	0	0	0	0	0	1	0	0	1
w_1	0	0	0	0	0	0	1	0	1

z_1	0	0	0	0	0	0	0	1	1

Table 0.12: F Additional properties map $BELIEF_2$ to BELIEF Type

4.2. Fault-Tolerance

As addressed in [KD03], autonomic systems have the following important self-managing characteristics: a) self-configuration: the ability of configuring system automatically according to the changing of environment; b) self-healing: the ability of detecting, managing and repairing bugs or failures in software as well as hardware systems; c) self-optimization: the ability of improving system operations and make themselves more efficient in performance or cost; and d) self-protection: the ability of protecting the whole system against malicious attacks and failures uncorrected by self-healing. This thesis uses fault-tolerance as a mechanism in order to model self-healing property with category theory. Fault-tolerance is defined as a property enables a system to continue operating properly in the event of the failure of (or one or more faults within) some of its components. The following sections take case study "Marsworld" and use case "recover damaged carry agent" (see Chapter 2) to illustrate the fault-tolerance properties: restarting and taking over, and using category theory to as a formal model.

4.2.1. Fault-Tolerance Property- Restart The Same Agent

In a multi-agent system, if an agent is not functional, the first basic solution to recover the system is restarting this agent. Before showing how this solution can be modeled in CAT, we recall the following concept:

Isomorphism *[Mac71]**:** An isomorphism* $\boldsymbol{T: C \rightarrow B}$ *of categories is a functor* $\boldsymbol{T}$ *from* $\boldsymbol{C}$ *to* $\boldsymbol{B}$*, which is a bijection, both on objects and on morphisms. In other words, a function*

***T: C→B** is an isomorphism if and only if there is a functor **S: C→B** for which both composites **(S o T)** and **(T o S)** are identity functors.*

Definition 4.1 ***Restart***: *An agent can be restarted, if and only if this agent's categories **Action**, **Plan**, **PLAN**, **GOAL**, **FactSet*** and ***BELIEF*** are *isomorphic to **repository agent**'s categories. These categories within repository exist as default before the agent is created, and can be updated during system runtime.* If this agent is restart-able, its supervisor agent will recreate the agent, otherwise, the agent's stored categories will be removed from the repository by the supervisor agent. We write *isomorphism (A, B) == TRUE* to indicate that category *A* is isomorphic to category *B*, otherwise we use *isomorphism (A, B) == FALSE*.

As we have defined in Chapter 3, each category ***Plan***, ***PLAN***, ***GOAL***, ***FactSet*** and ***BELIEF*** has a *self _functor*, which models the agent ability to update itself, such as removing objects and morphisms. If all of the agent's current (or up to date) categories are isomorphic to their corresponding repository agent's categories, then this agent can be restarted. For example (Figure 4.5): Agent *A* includes ***Plan'***, ***PLAN'***, ***GOAL***, ***FactSet*** **and** ***BELIEF***, where ***Plan'*** and ***PLAN'*** are updated from ***Plan*** and ***PLAN***, and ***GOAL***, ***FactSet*** and ***BELIEF*** are the same as default. Inside repository there are categories ***AgentA.Plan, AgentA.PLAN, AgentA.GOAL, AgentA.FactSet*** **and** ***AgentA.BELIEF***.

Figure 0.5: Fault-tolerance property- restart in agent A

From Definition 4.1, the following C-like statements check if agent *A* is able to be restarted.

if (*isomorphism (Action, AgentA.Action) == TRUE*

&& isomorphism (Plan', AgentA.Plan) == TRUE

&& isomorphism (PLAN', AgentA.PLAN) == TRUE

&& isomorphism (GOAL, AgentA.GOAL) == TRUE

&& isomorphism (FactSet, AgentA.FactSet) == TRUE

*&& isomorphism (BELIEF, AgentA.BELIEF) == TRUE****)***

then ***Agent can be restarted***

else ***Agent cannot be restarted***

4.2.2. Robotic Case Study: Restart the Same Carry Agent

In this section we will present the implementation of fault-tolerance in the robotic case study by using restart property. The detailed scenario of replacing damaged carry agent is described in section 2.6. In the robotic case study, there is an agent called $\boldsymbol{Carry_1}$ that does not perform its tasks correctly. Its supervisor agent will try to restart $\boldsymbol{Carry_1}$ from its

default stage. The supervisor agent needs to communicate with both *$Carry_1$* and repository agent and checks if ***$Carry_1$*** satisfies the conditions of Definitions *4.1*.

a) Isomorphism in Action

$Action_1$ is defined as an Action category (Chapter 3, Definition 3.2.1) of ***$Carry_1$***. ***$Carry_1$.Action*** is defined as information storage for ***$Carry_1$***'s Action in ***repository***. Suppose ***$Carry_1$.Action*** includes exactly the same objects as in ***$Action_1$***, then we have

*isomorphism (**$Action_1$**, **$Carry_1$.Action**) == TRUE*

b) Isomorphism in ***Plan***

$Plan_1$ is defined as a Plan category (Chapter 3, Definition 3.2.3) of ***$Carry_1$***. ***$Carry_1$.Plan*** is defined as information storage for ***$Carry_1$***'s Plan in ***repository***. Suppose ***$Carry_1$.Plan*** includes exactly the same objects and morphisms as in *$Action_1$*, then

*isomorphism (**$Plan_1$, Carry1.Plan**) == TRUE*

By using the same assumption as a) and b), we can have

c) Isomorphism in ***PLAN:*** *isomorphism (**$PLAN_1$, $Carry_1$.PLAN**) == TRUE*

d) Isomorphism in ***GOAL:*** *isomorphism (**$GOAL_1$**, **$Carry_1$.GOAL**) == TRUE*

e) Isomorphism in ***FactSet***: *isomorphism (**$FactSet_1$**, **$Carry_1$.FactSet**) == TRUE*

f) Isomorphism in ***BELIEF***: *isomorphism (**$BELIEF_1$**, **$Carry_1$.BELIEF**) == TRUE*

With above (a ~ f) conditions, ***$Carry_1$*** can be restart/recreated by its supervisor agent.

But Suppose ***$Carry_1$*** doesn't satisfy one of the (a ~ f) conditions, for example, ***$BELIEF_1$*** has more objects than ***$Carry_1$.BELIEF***, or ***$Plan_1$*** contains less morphisms than ***$Carry_1$.Plan***, then by definition of *isomorphism*,

*isomorphism (**$BELIEF_1$**, **$Carry_1$.BELIEF**) == False*, or

*isomorphism (**$Plan_1$**, **$Carry_1$.Plan**) == False.*

This means ***$Carry_1$*** cannot be restarted or recreated by its supervisor agent.

4.2.3. Fault-Tolerance Property- Takeover by Inclusion Agent

If the damaged agent cannot be replaced by an equivalent agent, the supervisor agent will try to find an inclusion agent (Definition 4.3) to takeover (definition 4.4) the damaged agent.

Definition 4.2 ***Include****: Let* ***C_1*** *and* ***C_2*** *be two categories.* ***C_2*** *is said to be included in* ***C_1*** *if and only if 1)* ***C_1*** *includes the same object and morphism types as* ***C_2*** *does; 2)* ***C_1*** *contains at least the same number of objects of each type as* ***C_2*** *does; and 3)* ***C_1*** *includes at least the same number of morphisms for each object as* ***C_2*** *does.*

We use ***include (C_1, C_2)*** *== TRUE* to denote that category ***C_1*** includes ***C_2***, and ***include (C_1, C_2)*** *== FALSE* to denote the negation.

Definition 4.3 ***Inclusion Agent:*** *Let A and B be two agents. If all the following categories:* ***Action, Plan, PLAN, GOAL, FactSet,*** *and* ***BELIEF*** *defined in A* ***include*** *B's* ***Action, Plan, PLAN, GOAL, FactSet,*** *and* ***BELIEF****, we say agent A is an* ***Inclusion Agent*** *of agent B.*

We use ***IncAgent (A, B) == TRUE*** to denote that agent *A* is an inclusion agent of agent *B*, otherwise, we write ***IncAgent (A, B) == FALSE***.

Definition 4.4 ***Takeover:*** *An agent A can take over (i.e. replace) an agent B if and only if IncAgent (A, B) == TRUE.*

For example (Figure 4.6): Agent *A* includes ***Action_ A***, ***Plan_ A***, ***PLAN_ A***, ***GOAL_ A***, and ***BELIEF_ A***. Agent B includes ***Action_ B***, ***Plan_ B***, ***PLAN_ B***, ***GOAL_ B*** and ***BELIEF_ B***.

Figure 0.6: Fault-tolerance property takeover by inclusion agent

From **Definitions 4.2, 4.3** and **4.4**, the following statements check if agent *A* can be taken over (i.e. replaced) by agent *B* (we user *take_over (Agnet X, Agent Y)* to denote *Agent X* can be taken over by *Agent Y*).

*IncAgent (**Agent B, Agent A**) ==*

***(**include (Action_B, Action_A)*

&& include (Plan_B, Plan _A)

&& include (PLAN_B, PLAN _A)

&& include (GOAL_B, GOAL _A)

*&& include (BELIEF_B, BELIEF _A)**)***

if (**IncAgent **(Agent B, Agent A))

then** take_over **(Agent A, Agent B)

else ¬**take_over **(Agent A, Agent B)

4.2.4. Robotic Case Study: Takeover Damaged Carry Agent by Inclusion Agent

In this section we will present the implementation of fault-tolerance in the robotic case study by using takeover property. The detailed scenario of replacing damaged carry agent is described in section 2.6. In this specific configuration of the robotic multi agent system, there is a damaged agent called $\boldsymbol{Carry_1}$. If $\boldsymbol{Carry_1}$ cannot be restarted and there is no equivalent agent to substitute it, then its supervisor agent will communicate with other agents in its group to try to find an agent to take-over the duties of $\boldsymbol{Carry_1}$.

a) ***Include*** in ***Action***

$\boldsymbol{Action_1}$ is defined as an Action category (Chapter 3, definition 3.2.9) of $\boldsymbol{Carry_1}$. It has three *types* of objects: $Act_{Trigger}$, Act_{Move} and Act_{Load}. And its objects $Act_{StartCarry}$ of type $Act_{Trigger}$, $Act_{MoveToTargetA}$, $Act_{MoveToTargetB}$ and $Act_{MoveToBase}$ of type Act_{Move}, and $Act_{LoadOre}$ of type Act_{Load} (Table 4.1).

$\boldsymbol{Action_2}$ is defined as an Action category of $\boldsymbol{Carry_2}$, and it contains three *types* of objects: $Act_{Trigger}$, Act_{Move}, Act_{Load} and Act_{Unload}. $\boldsymbol{Action_2}$ contains objects: $Act_{StartCarry}$ and $Act_{StartMove}$ of type $Act_{Trigger}$, $Act_{MoveToTargetA}$, $Act_{MoveToTargetB}$, $Act_{MoveToBase}$ and $Act_{MoveAround}$ of type Act_{Move}, $Act_{LoadOre}$ of type Act_{Load} and $Act_{UnloadOre}$ of type Act_{Unload} (Table 4.6) (See Figure 4.6 Include in Action in Case Study).

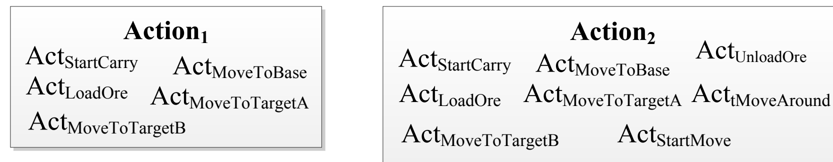


Figure 0.7: Include in action in the case study

$\boldsymbol{Action_2}$ ***includes*** $\boldsymbol{Action_1}$. $\boldsymbol{Action_2}$ contains all the three types of objects as in $\boldsymbol{Action_1}$ ($Act_{Trigger}$, Act_{Move} and Act_{Load}), and $\boldsymbol{Action_2}$ contains the same number of objects of each type as in $\boldsymbol{Action_1}$ ($Act_{StartCarry}$, $Act_{StartMove}$, $Act_{MoveToTargetA}$, $Act_{MoveToTargetB}$, $Act_{MoveToBase}$, $Act_{MoveAround}$, $Act_{LoadOre}$ and $Act_{UnloadOre}$).

b) ***Include*** in ***Plan***

$\boldsymbol{Plan_{1_A}}$ is defined as a Plan category (Chapter 3, Definition 3.2.1) of $\boldsymbol{Carry_1}$. It has three *types* of objects: $Act_{Trigger}$, Act_{Move} and Act_{Load}, and one type of morphism: *before*. $\boldsymbol{Plan_{1_A}}$ And its objects $Act_{StartCarry}$ of type $Act_{Trigger}$, $Act_{MoveToTargetA}$ and $Act_{MoveToBase}$ of type Act_{Move}, and $Act_{LoadOre}$ of type Act_{Load}. $\boldsymbol{Plan_{1_A}}$ contains morphisms l_{1_A}: $Act_{StartCarry} \rightarrow Act_{MoveToTargetA}$, s_{1_A}: $Act_{MoveToTargetA} \rightarrow Act_{LoadOre}$ and t_{1_A}: $Act_{LoadOre} \rightarrow Act_{MoveToBase}$ (Table 4.2).

$\boldsymbol{Plan_{2_A}}$ is defined as a Plan category of $\boldsymbol{Carry_2}$, and it has three *types* of objects: $Act_{Trigger}$, Act_{Move}, Act_{Load} and Act_{Unload}, and one type of morphism: *before*. $\boldsymbol{Plan_{2_A}}$ contains objects: $Act_{StartCarry}$ of type $Act_{Trigger}$, $Act_{MoveToTargetA}$ and $Act_{MoveToBase}$ of type Act_{Move}, $Act_{LoadOre}$ of type Act_{Load}, and $Act_{UnloadOre}$ of type Act_{Unload}. $\boldsymbol{Plan_{2_A}}$ contains morphisms: l_{2_A}: $Act_{StartCarry} \rightarrow Act_{MoveToTargetA}$, s_{2_A}: $Act_{MoveToTargetA} \rightarrow Act_{LoadOre}$, t_{2_A}: $Act_{LoadOre} \rightarrow Act_{MoveToBase}$ and h_{2_A}: $Act_{MoveToBase} \rightarrow Act_{UnloadOre}$, (Table 4.7). Figure 4.8 illustrates this case.

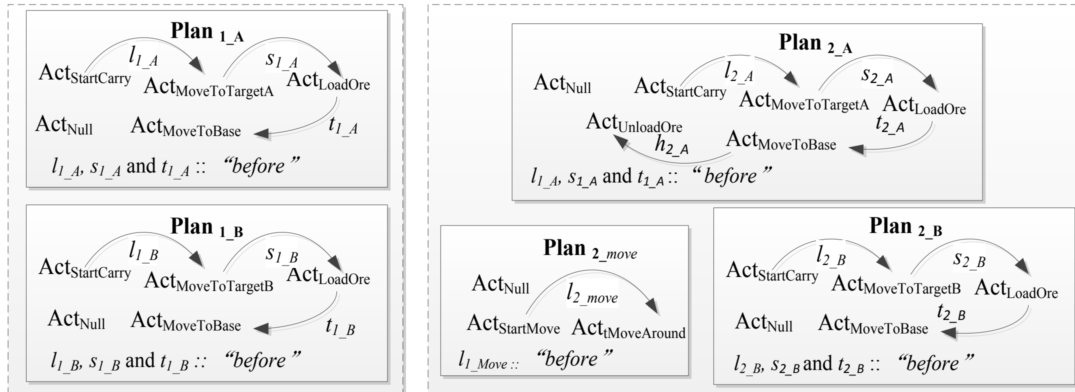


Figure 0.8: Include in plan in the case study

$\boldsymbol{Plan_{2_A}}$ ***includes*** $\boldsymbol{Plan_{1_A}}$. $\boldsymbol{Plan_{2_A}}$ contains all the three types of objects as in $\boldsymbol{Plan_{1_A}}$ ($Act_{Trigger}$, Act_{Move} and Act_{Load}), and $\boldsymbol{Plan_{2_A}}$ contains same number of objects of each type as in $\boldsymbol{Plan_{1_A}}$ ($Act_{StartCarry}$, $Act_{StartMove}$, $Act_{MoveToTargetA}$, $Act_{MoveToBase}$ and $Act_{LoadOre}$). $\boldsymbol{Plan_{2_A}}$ contains all the types of morphisms as in $\boldsymbol{Plan_{1_A}}$ (*before*), and $\boldsymbol{Plan_{2_A}}$ has the same number of

corresponding morphisms as in ***Plan**$_{1_A}$* ($Act_{StartCarry} \rightarrow Act_{MoveToTargetA}$, $Act_{MoveToTargetA} \rightarrow Act_{LoadOre}$ and $Act_{LoadOre} \rightarrow Act_{MoveToBase}$). Similar to ***Plan**$_{1_A}$* and ***Plan**$_{2_A}$*, ***Plan**$_{2_B}$ **includes** **Plan**$_{1_B}$ **Plan**$_{2_B}$*.

c) ***Include*** in **PLAN**

***PLAN**$_1$* is defined as a PLAN category (Chapter 3, Definition 3.2.2) of ***Carry**$_1$*. It has one type of objects: $Plan_{CarryOre}$, and one type of morphism: *before*. ***PLAN**$_1$* contains objects: $Plan_{CarryOreFromTargetA}$ and $Plan_{CarryOreFromTargetB}$ of type $Plan_{CarryOre}$. ***PLAN**$_1$* contains morphisms: p_1: $Plan_{CarryOreFromTargetA} \rightarrow Plan_{CarryOreFromTargetA}$ (*carryOreFromTargetA* can be repeated), q_1: $Plan_{CarryOreFromTargetB} \rightarrow Plan_{CarryOreFromTargetB}$ (*carryOreFromTargetB* can be repeated) and o_1: $Plan_{CarryOreFromTargetA} \rightarrow Plan_{CarryOreFromTargetB}$ (Table 4.3).

***PLAN**$_2$* is defined as a PLAN category of ***Carry**$_2$*, and it has two types of objects: $Plan_{CarryOre}$ and $Plan_{Move}$, and one type of morphism: *before*. ***PLAN**$_2$* contains objects: $Plan_{CarryOreFromTargetA}$, $Plan_{CarryOreFromTargetB}$ and $Plan_{MoveAround}$. ***PLAN**$_2$* contains morphisms p_2: $Plan_{CarryOreFromTargetA} \rightarrow Plan_{CarryOreFromTargetA}$, q_2: $Plan_{CarryOreFromTargetB} \rightarrow Plan_{CarryOreFromTargetB}$, o_2: $Plan_{CarryOreFromTargetA} \rightarrow Plan_{CarryOreFromTargetB}$ m_2: $Plan_{CarryOreFromTargetB} \rightarrow Plan_{MoveAround}$ and n_2: $Plan_{MoveAround} \rightarrow Plan_{MoveAround}$ (Table 4.8). Figure 4.9 illustrates this case.

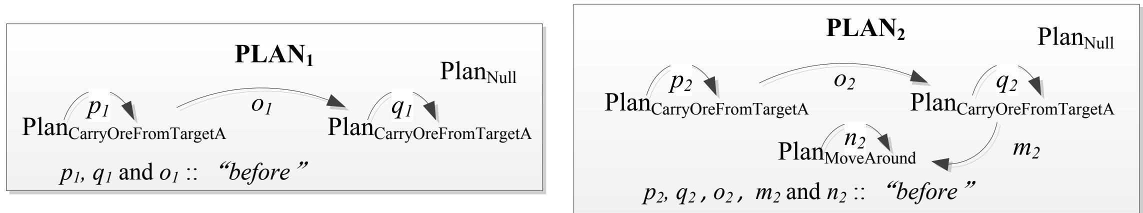


Figure 0.9: Include in PLAN in the case study

***PLAN**$_2$ **includes** **PLAN**$_1$*. ***PLAN**$_2$* contains all the types of objects as in ***PLAN**$_1$* ($Plan_{CarryOre}$), and ***PLAN**$_2$* contains the same number of objects of each type as in ***PLAN**$_1$*

($Plan_{CarryOreFromTargetA}$ and $Plan_{CarryOreFromTargetB}$). $\boldsymbol{PLAN_2}$ contains all the types of morphisms as in $\boldsymbol{PLAN_1}$ (*before*), and $\boldsymbol{PLAN_2}$ contains the same number of corresponding morphisms as in $\boldsymbol{PLAN_1}$ ($Plan_{CarryOreFromTargetA} \rightarrow Plan_{CarryOreFromTargetA}$, $Plan_{CarryOreFromTargetA} \rightarrow Plan_{CarryOreFromTargetB}$ and $Plan_{CarryOreFromTargetB} \rightarrow Plan_{CarryOreFromTargetB}$).

d) ***Include*** in ***GOAL***

$\boldsymbol{GOAL_1}$ is defined as a GOAL category (Chapter 3, Definition 3.3.1) of $\boldsymbol{Carry_1}$. It has one type of objects: $Goal_{CarryOre}$, and one type of morphism: *higher _priority*. $\boldsymbol{GOAL_1}$ contains objects: $Goal_{CarryOreFromTargetA}$ and $Goal_{CarryOreFromTargetB}$ of type $Goal_{CarryOre}$. $\boldsymbol{GOAL_1}$ contains morphisms: $Goal_{CarryOreFromTargetA} \rightarrow Goal_{CarryOreFromTargetB}$ (Table 4.4).

$\boldsymbol{GOAL_2}$ is defined as a GOAL category of $\boldsymbol{Carry_2}$, and it has two types of objects: $Goal_{CarryOre}$ and $Goal_{Move}$, and one type of morphism: *depends*. $\boldsymbol{GOAL_2}$ contains objects: $Goal_{CarryOreFromTargetA}$, $Goal_{CarryOreFromTargetB}$ and $Goal_{MoveAround}$. $GOAL_2$ contains morphisms: $Goal_{CarryOreFromTargetA} \rightarrow Goal_{CarryOreFromTargetB}$ and $Goal_{CarryOreFromTargetB} \rightarrow Goal_{MoveAround}$ (Table 4.10). Figure 4.10 illustrates this case.

Figure 0.10: Include in GOAL in the case study

$GOAL_2$ includes $GOAL_1$. In definition 4.2, $\boldsymbol{GOAL_2}$ contains all the types of objects as in $\boldsymbol{GOAL_1}$ ($Goal_{CarryOreGoal}$), and $\boldsymbol{GOAL_2}$ contains the same number of objects of each type as in $\boldsymbol{GOAL_1}$ ($Goal_{CarryOreFromTargetA}$ and $Goal_{CarryOreFromTargetB}$). $\boldsymbol{GOAL_2}$ contains all

the types of morphisms as in $\boldsymbol{GOAL_1}$ (*depends*), and $\boldsymbol{GOAL_2}$ contains the same number of corresponding morphisms as in $\boldsymbol{GOAL_1}$ ($Goal_{CarryOreFromTargetA} \rightarrow Goal_{CarryOreFromTargetB}$).

e) ***Include*** in ***BELIEF***

$\boldsymbol{BELIEF_1}$ is defined as a BELIEF category (Chapter 3, Definition 3.4.2) of $\boldsymbol{Carry_1}$. It has three types of objects: $FactSet_{CarryOre}$, $FactSet_{Base}$ and $FactSet_{Null}$, and one type of morphism: *subset _of*. $\boldsymbol{BELIEF_1}$ contains objects: $FactSet_{1_A}$ and $FactSet_{1_B}$ of type $FactSet_{CarryOre}$, $FactSet_{Base}$ of type $FactSet_{Base}$, and $FactSet_{Null}$ of type $FactSet_{1_B}$. $\boldsymbol{BELIEF_1}$ contains morphisms: u_1: $FactSet_{1_A} \rightarrow FactSet_{Base}$, v_1: $FactSet_{1_B} \rightarrow FactSet_{Base}$, x_1: $FactSet_{Null} \rightarrow FactSet_{1_A}$, and y_1: $FactSet_{Null} \rightarrow FactSet_{1_B}$ (Table 4.5).

$\boldsymbol{BELIEF_2}$ is defined as a BELIEF category of $\boldsymbol{Carry_2}$. It has four types of objects: $FactSet_{CarryOre}$, $FactSet_{MoveArea}$, $FactSe_{Base}$ and $FactSet_{Null}$, and one type of morphism: *subset _of*. $\boldsymbol{BELIEF_2}$ contains objects: $FactSet_{2_A}$ and $FactSet_{2_B}$ of type $FactSet_{CarryOre}$, $FactSet_{Move}$ of type $FactSet_{MoveArea}$, $FactSe_{Base}$ of type $FactSe_{Base}$, and $FactSet_{Null}$ of type $FactSet_{Null}$. $\boldsymbol{BELIEF_2}$ contains morphisms: u_2: $FactSet_{2_A} \rightarrow FactSet_{Base}$, v_2: $FactSet_{2_B} \rightarrow FactSet_{Base}$, w_2: $FactSet_{Move} \rightarrow FactSet_{Base}$, x_2: $FactSet_{Null} \rightarrow FactSet_{2_A}$, y_2: $FactSet_{Null} \rightarrow FactSet_{2_B}$ and z_2: $FactSet_{Null} \rightarrow FactSet_{Move}$ (Table 4.10). Figure 4.11 illustrates this case.

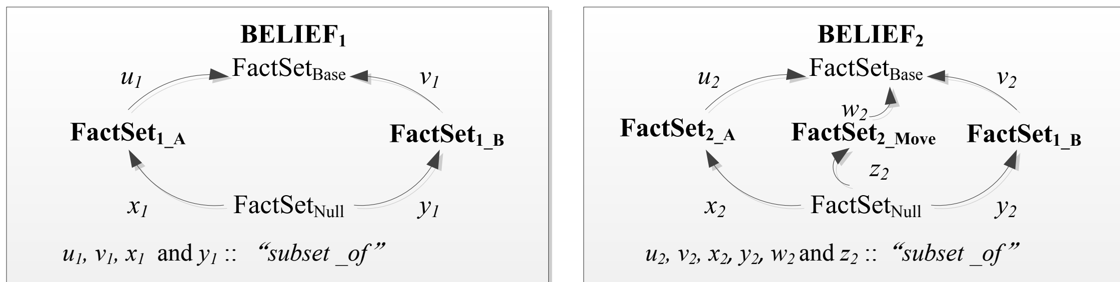


Figure 0.11: Include in BELIEF in case study

$\boldsymbol{BELIEF_2}$ ***includes*** $\boldsymbol{BELIEF_1}$. $\boldsymbol{BELIEF_2}$ contains all the types of objects as in $\boldsymbol{BELIEF_1}$ ($FactSet_{CarryOre}$, $FactSet_{Base}$ and $FactSet_{Null}$), and $\boldsymbol{BELIEF_2}$ contains the same number of objects of each type as in $\boldsymbol{BELIEF_1}$ ($FactSet_{1_A}$, $FactSet_{1_B}$, $FactSet_{Base}$ and $FactSet_{Null}$). $\boldsymbol{BELIEF_2}$ contains all the types of morphisms as in $\boldsymbol{BELIEF_1}$ (*subset _of*), and $\boldsymbol{BELIEF_2}$ contains the same number of corresponding morphisms as in $\boldsymbol{BELIEF_1}$ ($FactSet_{Null} \rightarrow FactSet_{1_A}$, $FactSet_{Null} \rightarrow FactSet_{1_B}$, $FactSet_{1_A} \rightarrow FactSet_{Base}$ and $FactSet_{1_B} \rightarrow FactSet_{Base}$).

With above (a ~ e) conditions, and by Definition 4.3, $\boldsymbol{Carry_2}$ is an *Inclusion Agent* of $\boldsymbol{Carry_1}$, and $\boldsymbol{Carry_2}$ is able to ***take-over*** $\boldsymbol{Carry_1}$. But Suppose $\boldsymbol{Carry_2}$ doesn't satisfy one of the (a ~ e) conditions, then it will not be an *Inclusion Agent* of $\boldsymbol{Carry_1}$ and it will not be able to ***take-over*** $\boldsymbol{Carry_1}$. For example, $\boldsymbol{GOAL_2}$ does not have object type $Goal_{CarryOre}$, or $\boldsymbol{GOAL_2}$ does not have object *carryOreFromTargetAGoal,* or $GOAL_2$ does not have *higher _dependency* morphism: $Goal_{CarryOreFromTargetA} \rightarrow Goal_{CarryOreFromTargetB}$, then by Definition 4.2, $\boldsymbol{GOAL_2}$ doesn't include $\boldsymbol{GOAL_1}$ and $\boldsymbol{Carry_2}$ is not an *Inclusion Agent* of $\boldsymbol{Carry_1}$, this means the tasks of $\boldsymbol{Carry_1}$ cannot be taken by $\boldsymbol{Carry_2}$.

Chapter 5: Related Work and Conclusions

In this chapter, we discuss the related work on using category theory to formalize multi-agent systems, list the contributions and conclude the thesis by outlining the future work directions.

5.1. Related Work and Significance of the Proposed Research

Category theory has been used as a formal model in computer science and software engineering for many years, and some of the related work can be summarized as follows:

In [GV79], the authors have applied the category theory as a conceptual tool to model general systems through the abstract representation of systems, which take objects, systems, interconnection, and behavior as a basis. The authors present a Behavioral Theorem, stating that the behavior of an interconnection between objects can be considered as the behaviors of individual objects; they also indicate that the notion of autonomy, interaction, cooperation, and self-organization are relevant to their study.

In [Hil93], the author has introduced architecture for system configuration that is independent of various approaches of system specification, design, and coding. The architecture focused on configuring those systems from reusable modules at any stage during system development. The module is precisely defined as an instance of a textual specification, and the configuration takes place in a mathematical framework that is based on category theory.

In [JD01], the author have illustrated how to use category theory as a meta-ontology for information systems research through some examples, which include system specification, definitions of views along with their updates, and system interoperations.

The related work has stayed that Category Theory (CAT) has more advantages on formalizing complex systems than other theory or modeling languages.

Domain theory is introduced as a study of special kinds of partially ordered sets (or posets) in mathematics, these sets are called domains. A partially ordered set (poset) formalizes and generalizes the intuitive concept of an ordering, sequencing, or arrangement of the elements of a set. “Partially order” means not every pair of elements need be related: for some pairs, it may be that neither element precedes the other in the same poset [AJ94]. In comparison to category theory, it has a limitation of not being expressive enough to capture relations between posets， such as “*depends on*”, since dependency is not a ordering, sequencing, or arrangement relation. Domain theory cannot be used to model self-relationships of elements within a poset, which is well defined in category theory as identity morphism. Moreover, with category theory and its own properties, automation can be achieved, for example, the composition of two specifications can be derived automatically, and this is not addressed in the domain theory.

Logics, such as first order logic, has been used to modeling multi-agent systems [Woo09]. In comparison to category theory, instead of capturing the structure and properties, it models the reasoning of properties that are shared by objects.

Few research papers toward modeling MAS with CAT, and they can be summarized as follows:

In [Pfa05], the author has introduced a MAS category. In that category the objects are agents and the morphisms represent all kinds of relations between the agents.

In [PS07], the author has also introduced typed category into multi-agent systems, and instead of defining a category with agent types as objects and communication types as morphisms as we did in Chapter 3, he applied sets of agent types and sets of communication types as the objects in one category, and agent types and communication types are generated by using two Push-out category approach [Mac71], called Double Push Approach (DPA). This approach provided a way to related agents with types, and communication with types, but didn't address the relationship between agent types and communication types.

In [CG06], the authors have introduced an Agent Modeling Language (AML) along with a demonstration on how AML can be applied to efficiently, accurately, and comprehensively model the Prospecting Asteroid Mission (PAM) [RT07] system. A selection of the AML models that specify the PAM domain, goals, architecture, as well as behaviors are also presented in this paper. However, this language lacks theoretical foundations, which makes proving the isomorphism of two different models relative to two equivalent systems practically impossible.

In terms of modeling and formalizing multi-agent systems using category theory, only a very high abstract level has been considered, such as modeling the whole system as a category, where agents are objects and communications [Pfa05] [PS07]. To our best knowledge, no work has considered the refinement of the categorical representation of agents into components using the BDI model of agents and the interaction between these components in the definition of agent architecture, as it was described in this thesis. In

fact, accounting for agent architecture in the categorical representation of MAS by zooming into single agent and analyzing the relationships among agent plans, goals and beliefs allows capturing the core of multi-agent systems and thus providing a fully formal representation on both the multi-agent system structure and autonomic computing properties. Furthermore, no previous work has considered the formalization of fault-tolerance property of multi-agent systems using category theory as it as described in Chapter 4. This property modeling work shows our research can be adapted to implementation level in IT industry easily.

5.2. Conclusions

This thesis begins with an introduction and brief dissuasion of software complexities in integrating and managing computing systems, follows with a comprehensive view for the autonomic computing paradigm, an introduction of the agent-based computing technology and a background for the category theory. We carry on with previous works on reactive autonomic systems framework (*RASF*) [KO08], and implement category theory (*CAT*) as a formal method to specify and model multi-agent system (*MAS*) in *RASF*. We have proposed our approach with the purpose of providing solutions for the following research questions:

1. How can each agent be modeled with CAT?
    a. What are the components of each agent?
    b. How do we model each component with CAT?
    c. How do we model the relations among the components with CAT?
2. How can MAS be modeled with CAT?

a. What are objects and morphisms to be used to capture the transformation from MAS to CAT?

b. Since agents and their communication can be classified into different types, how do we model these types with CAT?

3. How can CAT represent MAS properties?

The highlighted part of ***RASF*** project diagram depicted in Figure 5.1 was accomplished via this thesis.

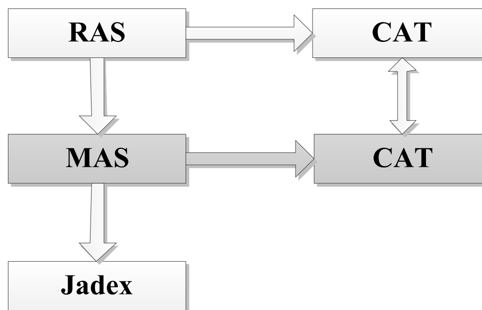


Figure 0.1: Reactive autonomic systems framework project with research coverage

5.3. Contributions

This thesis proposed a formal modeling of multi-agent systems (MAS) with category theory (CAT). This formal transformation helps us to focuses on the morphisms or relationships between objects i.e. as agents, rather than concentrates on these objects' representations. Moreover, besides mapping the overall MAS system into CAT, we are the first one in the related research field, to zoom into each agent, and model each internal component (such as plan) into CAT. This way guaranties our work is a fully CAT module. The main contributions of this thesis are listed below:

1. Modeling Agent with CAT [Chapter 3]

a. Modeling agent's plans, goals, and beliefs with CAT

b. Modeling relations between plans and goals, plans and beliefs, and goals and plans with CAT

2. Modeling MAS with CAT [Chapter 3]

   a. Modeling relations between agents

   b. Applying Type Category in MAS

3. Modeling robotic fault-tolerance with CAT [Chapter 4]

5.4. Future Work

This thesis is about the formalizing Multi-agent systems with Category theory, which brings us several related research opportunities. The flowing listed topics could be considered as the future work:

1. We can work on implement CAT by using Extensible Markup Language (XML). XML is machine readable language that has a high adaptability to many different environments and platforms. Adding CAT XML codes inside systems will improve the system efficiency since it's simple, easy to modify and most modem programming languages have the ability to understand XML.
2. We can work on developing a model transformation tool to automatically transfer MAS based XML [Sha11] to CAT based XML, with which, the mapping from MAS to CAT can be done by system-self.
3. We can work on modeling other self-managing properties with CAT. This thesis modeled fault-tolerance property with CAT. By the definition of autonomic systems,

there are more self-managing properties, such as self-configuration and self-optimization need to be modeled. This future work will be an extension of this thesis.

4. We can also work on other Reactive Autonomic Systems Framework (RASF) projects, such as modeling Reactive Autonomic Systems (RAS) to CAT, or proving of MAS based CAT model is isomorphism to RAS based CAT model.

References

[KO08] Heng Kuang and Olga Ormandjieva. *Self-Monitoring of Non-Functional Requirement in Reactive Autonomic System Framework: A Multi-Agent Systems Approach.* Third International Multi-Conference on Computing in the Global Information Technology (ICCGI'08).

[KO09] Heng Kuang, Olga Ormandjieva, Stan Klasa, N. Khurshid, and J.Bentahar, *Towards specifying reactive autonomic systems with a categorical approach: a case study.* Studies in Computational Intelligence, Volume 253/2009, Springer Berlin/Heidelberg, November 2009, 119-134.

[Hp01] P. Horn, "*Autonomic Computing: IBM Perspective on the State of Information Technology*", Presented at AGENDA 2001, IBM T. J. Watson Labs, October 2001.

[Sha11] Nassir Shafiel-Dizaji, "*Multi-Agent Approach To Modeling And Implementing Fault-Tolerance in Reactive Autonomic Systems*", Master Thesis of Concordia University, 2011.

[Mur04] Murch, R. "*Autonomic Computing*". IBM Press, 2004.

[TC04] G. Tesauro, D. M. Chess, W. E. Walsh, R. Das, A. Segal, I. Whalley, J. O. Kephart, and S. R. White, "*A Multi-Agent Systems Approach to Autonomic Computing*", Proceedings of the 3rd International Joint Conference on Autonomous Agents and Multi-Agent Systems, July 2004, Page 464 – 471.

[WH03] T. D. Wolf and T. Holvoet, "*Towards Autonomic Computing: Agent-Based Modeling, Dynamical Systems Analysis, and Decentralised Control*", Proceedings of the 1st International Workshop on Autonomic Computing Principles and Architectures, August 2003, Page 470 – 479.

[Fia98] Jose Luiz Fiadeiro. *Categories for Software Engineering*. Springer Berlin Heidelberg New York. ACM Computing Classification, 1998.

[Wir90] M. Wirsing, *Algebraic Specification*, Handbook of Theoretical Computer Science, Volume B, Elsevier and MIT Press, July 1990, Page 675 – 788.

[FM92] J. Fiadeiro and T. Maibaum, *Temporal Theories as Modularisation Units for Concurrent System Specification*, Formal Aspects of Computing, Volume 4, No. 3, May 1992, Page 239 – 272.

[OQ08] O. Ormandjieva and J. Quiroz, "*Methodology for Automatic Generation of Exhaustive Behavioral Models in Reactive Autonomic Systems*", Proceedings of the International Conference on Software Engineering Theory and Practice, Orlando, FL, USA, July 2008, Page 95 – 104.

[KC03] J. O. Kephart and D. M. Chess, "*The Vision of Autonomic Computing*", Computer, Volume 36, No. 1, IEEE Computer Society Press, Los Alamitos, CA, USA, January 2003, Page 41 – 50.

[KB10] HengKuang, Jamal Bentahar, Olga Ormandjieva, Nassir Shafieidizaji and Stan Klasa. *Formal Specification of Substitutability Property for Fault-Tolerance in Reactive Autonomic Systems.* V 9th International Conference on Software Methodologies, Tools and Techniques (SoMeT'2010), Yokohama, Japan, Sept. 29 to Oct. 1st, 2010.

[NO10] Noorulain Khurshid, Olga Ormandjieva, Stan Klasa. *Towards a Tool Support for Specifying Complex Software Systems by Categorical Modeling Language*. Book Chapter in Studies in Computational Intelligence, LNCS, 2010.

[OK06] O. Ormandjieva, H. Kuangabd E. Vassev, *Reliability Self-Assessment in Reactive Autonomic Systems: Autonomic System-Time Reactive Model Approach.* International Transactions on Systems Science and Applications, Volume 2, No. 1, September 2006, Page 99-104.

[WJ95] Wooldridge, M. and Jennings, N. R. Intelligent agents: Theory and practice. *The knowledge Engineering Review*, 10(2):115-152.(1995).

[Woo09] Michael Wooldridge. *An Introduction to Multi Agent Systems*. John Wiley & Sons, 2009

[Syc98] K. P. Sycara.*"Multi Agent Systems"*, AI Magazine, Volume 19, No. 2, July 1998, Page 79 – 92.

[PB07] Alexander Pokahr and Lars Braubach. *Jadex User Guide*. Distributed Systems Group, University of Hamburg, Germany. (2007)

[Mac71] S. Mac Lane. *Categories for the Working Mathematician*. Springer–Verlag: New York, Heidelberg, Berlin. (1971).

[Eas99] Steve Easterbrook. *An introduction to Category Theory for Software Engineers*. http://www.cs.toronto.edu/~sme/presentations/cat101.pdf/, 1999, Page 5–13.

[AWO06] S. Awodey. *Category Theory*. Oxford University Press, July 2006.

[OMG] Object Management Group (OMG); Object Constraint Language OMG Available Specification Version 2.0, May 2006

[Pfa05] Jochen Pfalzgraf. *On Categorical and Logical Modeling in Multiagent Systems.* In George E. Lasker and D.M. Dubois, editors, Anticipative and Predictive Models in Systems Science, volume 1. IIAS, 2005.

[PS07] Jochen Pfalzgraf, Thomas Soboll. *On a General Notion of Transformationfor Multiagent Systems.* Integrated Design and Process Technology, IDPT-2007 Printed in the United States of America, June, 2007.

[WE98] V. Wiels and S. Easterbrook, *Management of Evolving Specifications Using Category Theory*, Proceedings of the 13th IEEE International Conference on Automated Software Engineering, October 1998, Page 12 – 21.

[Awo06] S. Awodey, *Category Theory*, Oxford University Press, July 2006.

[JS98] N. R. Jennings, K. P. Sycara, M. Wooldridge, "*A Roadmap of Agent Research and Development*", International Journal of Autonomous Agents and Multi-Agent Systems, Volume 1, Issue 1, July 1998, Page 7 – 38.

[Kps98] K. P. Sycara, "*Multi Agent Systems*", AI Magazine, Volume 19, No. 2, July 1998, Page 79 – 92.

[WJ94] M. Wooldridge and N. R. Jennings, "*Agent Theories, Architectures, and Languages: a Survey*", Proceedings of the Workshop on Agent Theories, Architectures, and Languages on Intelligent Agents, August 1994, Page 1 – 39.

[PB05] A. Pokahr, L. Braubach, and W. Lamersdorf, "*Jadex: a BDI Reasoning Engine*", Multi-Agent Programming, Springer, September 2005, Page 149 – 174.

[SB02] R. Sterritt, D. Bustard, "*Towards Autonomic Computing: Effective Event Management*", Proceedings of the 27th Annual NASA Goddard/IEEE Software Engineering Workshop, December 2002, Page 40 – 47.

[KD03] J. O. Kephart, D. M. Chess, "*The Vision of Autonomic Computing*", Computer, Volume 36, No. 1, January 2003, Page 41 – 50.

[SI07] B. Solomon, D. Ionescu, M. Litoiu, and M. Mihaescu, "*Towards a Real-Time Reference Architecture for Autonomic Systems*", Proceedings of the International Workshop on Software Engineering for Adaptive and Self-Managing Systems, May 2007, Page 10 – 19.

[IBM01] *IBM Autonomic Computing Website*. http://www.research.ibm.com/autonomic/overview/

[Fau11] *Fault-tolerant systems*. http://en.wikipedia.org/wiki/Fault-tolerant_system

[GV79] J. A. Goguen and F. J. Verela, "*Systems and Distinctions, Duality and Complementarity*", International Journal of General Systems, Volume 5, No. 1, January 1979, Page 31 – 43.

[Hil93] G. Hill, "*Category Theory for the Configuration of Complex Systems*", Proceedings of the 3rd International Conference on Methodology and Software Technology, June 1993, Page 193 – 200.

[JD01] M. Johnson and C. N. G. Dampney, "*On Category Theory as a (meta) Ontology for Information Systems Research*", Proceedings of the International Conference on Formal Ontology in Information Systems, October 2001, Page 59 – 69.

[CG06] R. Cervenka, D. Greenwood, and I. Trencansky, "*The AML Approach to Modeling Autonomic Systems*", Proceedings of the 2nd International Conference on Autonomic and Autonomous Systems, July 2006, Page 29 – 34.

[RT04] P. E. Clark, M. L. Rilee, W. Truszkowski, G. Marr, S. A. Curtis, C. Y. Cheung, and M. Rudisill, "PAM: Biologically Inspired Engineering and Exploration

Mission Concept, Components, and Requirements for Asteroid Population Survey", Proceedings of the 55th International Astronautical Congress, October 2004, IAC-04-Q5.07.

[Fer99] J. Ferber, "*Multi-agent systems: an introduction to distributed artificial intelligence*". Addison-Wesley, 1999.

[AJ94] S. Abramsky, A. Jung (1994). "*Domain theory*". In S. Abramsky, D. M. Gabbay, T. S. E. Maibaum, editors, (PDF). Handbook of Logic in Computer Science. III. Oxford University Press. ISBN 0-19-853762-X. Retrieved 2007-10-13.