AN ALGEBRAIC FRAMEWORK FOR COMPOSITIONAL DESIGN OF AUTONOMOUS AND ADAPTIVE MULTIAGENT SYSTEMS

by

WALAMITIEN HERVÉ OYENAN

B.S., Université des Sciences et Technologies de Lille, 2001
M.S., Kansas State University, 2003

AN ABSTRACT OF A DISSERTATION

submitted in partial fulfillment of the requirements for the degree

DOCTOR OF PHILOSOPHY

Department of Computing and Information Sciences
College of Engineering

KANSAS STATE UNIVERSITY
Manhattan, Kansas

2010

Abstract

Organization-based Multiagent Systems (OMAS) have been viewed as an effective paradigm for addressing the design challenges posed by today's complex systems. In those systems, the organizational perspective is the main abstraction, which provides a clear separation between agents and systems, allowing a reduction in the complexity of the overall system. To ease the development of OMAS, several methodologies have been proposed. Unfortunately, those methodologies typically require the designer to handle system complexity alone, which tends to lead to ad-hoc designs that are not scalable and are difficult to maintain. Moreover, designing organizations for large multiagent systems is a complex and time-consuming task; design models quickly become unwieldy and thus hard to develop.

To cope with theses issues, a framework for organization-based multiagent system designs based on separation of concerns and composition principles is proposed. The framework uses category theory tools to construct a formal composition framework using core models from the Organization-based Multiagent Software Engineering (O-MASE) framework. I propose a formalization of these models that are then used to establish a reusable design approach for OMAS. This approach allows designers to design large multiagent organizations by reusing smaller composable organizations that are developed separately, thus providing them with a scalable approach for designing large and complex OMAS.

In this dissertation, the process of formalizing and composing multiagent organizations is discussed. In addition, I propose a service-oriented approach for building autonomous, adaptive multiagent systems. Finally, as a proof of concept, I develop two real-world examples from the domain of cooperative robotics and wireless sensor networks.

AN ALGEBRAIC FRAMEWORK FOR COMPOSITIONAL DESIGN OF AUTONOMOUS AND ADAPTIVE MULTIAGENT SYSTEMS

by

WALAMITIEN HERVÉ OYENAN

B.S., Université des Sciences et Technologies de Lille, 2001
M.S., Kansas State University, 2003

A DISSERTATION

submitted in partial fulfillment of the requirements for the degree

DOCTOR OF PHILOSOPHY

Department of Computing and Information Sciences
College of Engineering

KANSAS STATE UNIVERSITY
Manhattan, Kansas

2010

Approved by:

Major Professor
Scott A. DeLoach

Abstract

Organization-based Multiagent Systems (OMAS) have been viewed as an effective paradigm for addressing the design challenges posed by today's complex systems. In those systems, the organizational perspective is the main abstraction, which provides a clear separation between agents and systems, allowing a reduction in the complexity of the overall system. To ease the development of OMAS, several methodologies have been proposed. Unfortunately, those methodologies typically require the designer to handle system complexity alone, which tends to lead to ad-hoc designs that are not scalable and are difficult to maintain. Moreover, designing organizations for large multiagent systems is a complex and time-consuming task; design models quickly become unwieldy and thus hard to develop.

To cope with theses issues, a framework for organization-based multiagent system designs based on separation of concerns and composition principles is proposed. The framework uses category theory tools to construct a formal composition framework using core models from the Organization-based Multiagent Software Engineering (O-MASE) framework. I propose a formalization of these models that are then used to establish a reusable design approach for OMAS. This approach allows designers to design large multiagent organizations by reusing smaller composable organizations that are developed separately, thus providing them with a scalable approach for designing large and complex OMAS.

In this dissertation, the process of formalizing and composing multiagent organizations is discussed. In addition, I propose a service-oriented approach for building autonomous, adaptive multiagent systems. Finally, as a proof of concept, I develop two real-world examples from the domain of cooperative robotics and wireless sensor networks.

Table of Contents

List of Figures

List of Definitions

Acknowledgements

This dissertation would not have been possible without the help, support and guidance of my advisor Dr. Scott DeLoach. He has given me the encouragements, responsibility and freedom I needed to complete this work. His comments, critiques and jokes have helped me tremendously during the process of completing this degree. During the times of frustration, he knew how to guide me and during the times of confidence, he knew how to challenge me. I really feel privileged to have worked under his supervision.

I also thank the members of my committee, Dr. Gurdip Singh, Dr. David Gustafson, and Dr. Steve Warren for the time, insight, and input they provided to me during my proposal. In particular, many thanks to Dr. Gurdip Singh for giving me many wonderful opportunities to work in the field of wireless sensor networks and for his precious collaboration that gave me a lot of great ideas.

Special thanks go to my colleagues at the Multiagent and Cooperative Lab from which I drew inspiration, ideas and friendships: JC Ojeda, Jorge Valenzuela, Scott Harmon, Chris Zhong, and Matt Miller. They have also filled my graduate life with the much needed distraction.

A very special thank you goes to my mother, father, brothers and sister. They have been a great support and I clearly would not be where I am today without their help.

Dedication

To my parents. Their love and encouragement has been a constant support for me.

CHAPTER 1 - INTRODUCTION

"There is no such thing as a long piece of work, except one that you dare not start."
—*Charles Baudelaire*

""Begin at the beginning,", the King said, very gravely, "and go on till you come to the end: then stop" " —*Lewis Carroll, Alice in Wonderland*

1.1 Motivation

Developing large and complex systems has always been a challenging problem to tackle. In Object-Oriented, Booch [8] has suggested decomposition to handle this complexity. In general, decomposition is considered as a key property to tackle the growing complexity of software. In addition, software systems are expected to be intelligent and autonomous in order to adapt to unpredictable situations. Multiagent Systems (MAS) have been seen as a new paradigm to cope with the increasing need for complex applications that adapt to unpredictable situations. This shift of concept from the object paradigm to the agent paradigm allows system designers to replace passive objects by autonomous agents that can have their own goals and interact with each other and their environment [87]. These attributes make agents-based systems a natural mean for building complex systems [66]. System designers can then decompose their systems into individual tasks that can be achieved by agents [31]. As a result, large MAS are often composed of several autonomous agents engaging in complex interactions. Consequently, as pointed out by Wester-Ebbinghaus et al. [118], providing a correct and effective design for such systems is a difficult task. To reduce this complexity, Organization-based Multiagent Systems (OMAS) have been introduced. They use the organization paradigm, which provides better

abstractions for addressing the design challenges of large and complex MAS [39, 125]. In OMAS, the organizational perspective is the main abstraction, which provides a clear separation between agents and system, allowing a reduction in the complexity of the system. To support the development of OMAS, several methodologies have been proposed [36].

Nonetheless, one of the major problems associated with the wide-scale adoption of OMAS for the development of large-scale industrial and commercial applications is that, so far, most methodologies proposed work well for small in-house applications but are not well suited for developing complex applications. In fact, designing agent organizations for large real life systems can be a very complex and time-consuming task and design models can quickly become huge and difficult to develop and maintain. For instance, when designing a tracking application for Wireless Sensor Networks, the designer needs to handle not only the tracking application requirements, but also additional requirements linked to crucial tasks such as routing, aggregation, and time synchronization. Most of the time, designers tend to incorporate those secondary tasks in the main application goals or roles, which does not offer much reusability and maintainability.

For that reason, it has been long suggested that decomposition in OMAS would help cope with the complexity of systems [13]. However, most of the current methodologies just suggest the decomposition of large organizations into smaller ones and fail to provide a rigorous process to easily recombine them. For instance, Ferber et al. proposes partitioning a multiagent system into groups [39]. Those groups can interact with each other by having a gatekeeper agent participating in multiple groups. However, there is no formal description on the way those groups are aggregated into one consistent system. Similarly, Zambonelli et al. propose a methodology based on organizational abstraction for multiagent systems [125]. They recognize the importance of reusability in OMAS and propose dividing the system into loosely-coupled sub-organizations to reduce design complexity. Nonetheless, reconnection of those sub-organizations is left to the designer who needs to know the internal behavior of each sub-organization in order to assemble them appropriately. Hence, in most cases, the designer informally uses the agent interaction mechanisms to integrate multiagent organization designs. As a result, system designers are often required to handle all of the complexity alone, which may lead to an ad-hoc design that is not scalable and is very difficult to maintain. In addition, even if OMAS were designed with reuse in mind, the process is often not repeatable and the

functionalities of the resulting application are very difficult to verify due to an informal architectural design.

Therefore, there is a need for extending the current methodologies in order to support the design of large-scale multiagent systems.

1.2 Thesis Statement

The thesis of this dissertation is:

Agent Oriented methodologies can be devised to provide reusability and flexibility in the design of Adaptive Organization-based Multiagent Systems by incorporating a formal compositional approach to design composable multiagent organizations.

Within the context of this dissertation, I define *reusability* of a design model as the number of its design components that have been reused from previous projects [4, 42, 103]. I also define *flexibility* of a design model as the number of its design components that need to be modified in order to add a new requirement to the system [4, 34].

1.3 Goals

In order to address the thesis stated above, I developed a framework that allows a compositional design of Multiagent Organizations. This framework facilitates the design of large-scale OMAS by exploiting the interesting features of separation of concerns and reusability.

Hence, this dissertation has three main goals that represent its main contributions:

- **Goal 1:** Develop a general algebraic composition framework that formally characterizes the composition of two or more organization design models.
- **Goal 2:** Derive a specific service-oriented composition framework based on design models from the O-MASE process framework.
- **Goal 3:** Demonstrate the usefulness and validity the proposed framework for developing adaptive organization-based multiagent systems.

1.4 Research Approach

In this section I give a brief idea on how I achieved the goals. I formulate three research questions that match the three goals of this dissertation:

Research Question 1: How can we compose organization design models?

I use Category Theory [5] to formalize the composition of organizations. This mathematical framework allows us to formally represent organization design models and derive their compositions. I focus on one organizational framework, the Organization Model for Adaptive Computational Systems (OMACS), but my approach can be adapted and used with other model-based organization approach proposed in the literature such as Tropos, Gaia or Prometheus [36]. The design of applications following the OMACS model is supported by a rigorous methodology tailored from the O-MaSE process framework. This methodology defines several design models but only the two main design models are considered: the goal model and the role model. Those models are chosen because they are sufficient to define an organization.

In the composition framework proposed, the goal models and the role models are composed separately and then the composition for complete organizations is derived. I propose a formalization of those two design models using categorical concepts and show a construction to build the composition of two organizations.

I chose a category theory approach because the composition of two organizations is constructive. Hence, if the organizations used in the composition satisfy some properties, this construction becomes trivial and can be automatically derived. Moreover, the composition construction is guaranteed to result in a correct organization. Alternatively, due to the complexity of the organization models, giving the concrete details of the same construction in set theory is cumbersome and proving that the construction produces the correct result would be tedious. However, proofs in category theory are often short and elegant as they follow common construction patterns.

Research Question 2: What entities are needed to reuse composable organizations during the design phase?

During the design process, composing any two organizations may result in an arbitrary organization that may not be meaningful and useful to the designer. This situation is obviously not desirable. Hence, when composing organizations, designers must be able to preserve semantic properties that insure that the composed organization behaves as expected. For this reason, I propose a service-oriented framework to help designing OMAS. In this framework, multiagent organizations are viewed as reusable components that *use* and *provide* services. I define and formalize all the entities required to develop *reusable organizations.* These entities, which are elements from the design models, represent generic interfaces that are used to compose reusable organizations into a single composite organization.

Research Question 3: How can the composition framework be used for developing adaptive intelligent applications?

To demonstrate the usefulness and validity of the compositional framework to design real-world applications, I develop two applications with several services. For the first application, I design several cooperative robotic services and show how they can be reused to build other organizations. I also exemplify how several services can be composed to create a complex organization design. The second application is a Wireless Sensor Networks application that uses one service. I provide two different designs of the same service and show how the service-oriented framework allows the permutation of those two designs without any modifications to the main design. Furthermore, I implement one design of this application in order to demonstrate some adaptive properties of the system.

1.5 Evaluation

The formal composition of OMAS design models is validated through formal analysis and rigorous proofs. In particular, I prove that the composition of two organizations result in a unique organization that preserve the structure of the initial organizations.

Furthermore, the service-oriented framework, which allows designers to define reusable organizations, is built based on the general composition framework and is validated through two examples from different domains. I demonstrate through these examples how the service-oriented framework can be used and I put forward the benefits of using my proposed approach.

1.6 Assumptions

The number of available agents is *limited* and they can enter and leave the organization at any time. In addition, this work considers *open systems* in which agents are *cooperative* and work together for the achievement of the main organization goal. Hence, agents do not have any other goals (like their own goals) other than the organization goals.

Only design models are considered during the composition. Hence, the composition framework exposed in this dissertation is only applicable *at design time*. The resulting composite organization design can then be populated with agents in order to have a concrete organization instance at runtime.

1.7 Summary

This chapter is an introduction to the goals of my research. The rest of this dissertation is organized as follows:

Chapter 2 establishes the background necessary for this dissertation.

Chapter 3 introduces basic category theory constructions that are used throughout this dissertation.

Chapter 4 presents a category-theoretic framework for the compositional design of multiagent organizations.

Chapter 5 discusses a service-oriented approach that helps designers to build valid composite organizations.

Chapter 6 demonstrates the usefulness and validity of the compositional framework to design applications.

Chapter 7 concludes this dissertation and provides new directions for future research.

CHAPTER 2 - BACKGROUND

If I have seen further than others, it is by standing upon the shoulders of giants.” —*Isaac Newton*

“"The time has come," the walrus said, "to talk of many things: Of shoes and ships - and sealing wax - of cabbages and kings”
—*Lewis Carroll, Alice in Wonderland*

In this chapter, I outline of the basic concepts necessary in order to provide a better understanding of the work presented in this dissertation. I define the concepts of agent and multiagent systems and review some of the most notable models and methodologies for Organization-based Multiagent Systems (OMAS). I also present some compositional frameworks from other domains.

2.1 Agents and Multiagent Systems

Multi-agent systems are a natural fit for handling complexity of modern software systems, [66]. Research in multiagent system draws its inspiration from other scientific fields like sociology, linguistics or cognition research.

In the next two subsections, the concept of agent is first introduced and then the notion of a multiagent system is described.

2.1.1 Agents

In the literature, there is not a common and unique definition of agents. Russell and Norvig [100] view an agent as:

"anything that can be viewed as perceiving its environment through sensors and acting upon that environment through effectors".

Wooldridge [120] defines an agent as:

"a computer systems that is situated in some environment, and that is capable of autonomous action in this environment in order to meet its design objectives".

While an agent could be a simple control program, the type of agent used in this work is an intelligent agent. According to Wooldridge [117], an intelligent agent is an agent that can exhibit three types of autonomous actions:

- reactivity: Reactive agents respond directly to change in the environment in order to achieve their design goals.
- proactivity: Proactive agents have the ability to take goal-orientated initiatives.
- social ability: social agents are able to interact with other agents or humans (negotiations, cooperation) to accomplish their objectives.

Several architectures have been proposed for intelligent agent systems. Those architectures mainly aim at helping the agent decide what action to take in order to best accomplish its design goal. Surveys of general agent architectures can be found in [62]. The agent research community considers three different types of paradigms for intelligent agent architectures [122]:

- reactive architectures;
- deliberative architectures;
- hybrid architectures.

The *reactive architectures* consider reactive agents and are based on the perception/action capabilities of the agents. These architectures do not include a global model of the environment and consequently, agents cannot plan nor have a goal to pursue. A typical

architecture is the subsumption architecture of Brooks [14]. On the other hand, *deliberative architectures* have a symbolic representation of the environment and the desired behavior, which helps agents reason and construct plans. A typical architecture is the belief-desire-intention architecture (BDI) [10, 96] that takes inspiration from how humans make decisions by reasoning upon beliefs, desires, and intentions. Finally, *hybrid architectures* integrate reasoning with reactivity in order to combine the advantages of both architectures. Hybrid architectures are becoming important for multiagent systems (cf. Section 2.1.2) in which low level tasks can be done with *reactive agents* and high-level tasks (like group management) can be with *reasoning agents*.

A good way to understand the concept of agent is to compare it to that of objects. Although agents and objects (from object-oriented programming) have several similarities, they differ mainly on the notion of autonomy and interaction [87, 117]. An object can be viewed as passive as it has no control over its behavior. Its methods are always executed whenever another object invokes it. On the other hand, an agent can be viewed as active. It has the ability to choose which behavior to execute based on its view of the environment and its goals [87, 117]. However, this difference is not always clear as object can be implemented to be more active and agents can be implemented without autonomy. Moreover, the objects interact only via method calls whereas agents are social entities that exhibit more complex interaction mechanism (e.g. negotiations) [87, 117].

2.1.2 Multiagent Systems

A multiagent system (MAS) is a system composed of a group of agents interacting with each other to achieve a common or individual goal. [67]. In [111], Sycara identifies four major characteristics of MAS:

- Each agent has incomplete information and restricted capabilities to solve the problem
- System control is decentralized,
- Data is decentralized
- Computation is asynchronous

Modern systems are inherently distributed and control is scattered among several entities. Hence, MAS then become an effective approach to modeling these complex interactions between entities in the system [66].

The agents in MAS can be homogenous or heterogeneous, they might cooperate or compete with each other, and they might be in a hierarchical or flat structure [37, 117]. This dissertation is mainly concerned with heterogeneous agent working cooperatively to achieve a common goal.

2.2 Organization-based Multiagent Systems (OMAS)

Large Multiagent Systems (MAS) are often composed of several agents engaging in complex interactions with each other and their environment. Consequently, providing a correct and effective design for such systems is a difficult task [118]. To reduce this complexity, Organization-based Multiagent Systems (OMAS) have been introduced and are viewed as an effective paradigm for addressing the design challenges of large and complex MAS [39, 106, 125]. In OMAS, the organizational perspective is the main abstraction, which provides a clear separation between agents and system, allowing a reduction in the complexity of the system.

The aim of OMAS is to apply principles from Organization Theory [19, 41] to provide a more systematic and scalable way to design multiagent systems. In fact, the concept of organization provides a natural approach for managing groups. It adds a structure to agents and helps define and enforce norms. Agent organizations have various structures and designs. They can be defined as hierarchies, matrices, holons, coalitions, teams, or federations (see [55] for a summary). In order to design OMAS, several organization model/metamodel have been proposed. In what follows, I briefly introduce some of the most important while the next section describes in more details the OMACS model that my work is based on.

2.2.1 AGR

AGR (Agent, Group, Role) [39] is one of the first is organizational metamodel proposed in the literature. It is based on the agent, group and role concepts. An agent is an autonomous

entity playing roles within several groups. AGR does not prescribe any agent architecture, thus allowing any type of agents in the organization. A group constitutes a context of interaction for agents and is used for partitioning organizations. Agents may communicate if and only if they belong to the same group. A role is a functional position of an agent. The role encapsulates the way an agent should act within a group. Roles are tied to a particular group and are requested by agents.

AGR is a very simplistic model that only focuses on the structure of the organization and do not handle dynamic interactions.

2.2.2 Moise+

MOISE+ (Model of Organization for multI-agent SystEms) [56] is an organizational model that models the structural, functional and deontic aspects of multiagent organizations. The structural aspect of MOISE+ is similar to the AGR model, defining the organizational structure via roles, groups, and links. The functional aspect describes how organization goals are achieved, i.e., how these goals are decomposed and allocated to agents. Finally, the deontic aspect describes the permissions and obligations of roles.

Moise+ introduces a reorganization process that allows the organization to adapt to environmental changes [57]. This reorganization is based on the static description of the organization and the current state of one instance of this organization. Hence Moise+ allows runtime changes at the structural level (creation of groups, changes in roles) and at the functional level (changes in group members, changes in permissions).

2.2.3 OperA

In OperA [30], Dignum presents a three-part framework consisting of an organizational model, a social model and an interaction models. The Organizational Model describes the desired behavior of the organization, by defining the roles, norms, interactions and communication frameworks that are available in the domain. The Social Model, instantiated at run-time, maps organizational roles to specific agents. The necessary conditions that allow an agent to enact a role are defined in social contracts. The Interaction Model, also created at run-

time, specifies the interaction agreements between role-enacting agents as interaction contracts, which include the potential reward and penalties.

OperA mainly focuses on open systems and interaction and represents the structure of the organization independently from the internal design of the agents. By separating the Social Model and the Interaction Model, OperA allows the design of flexible organizations as they can have several possible concrete interactions.

2.2.4 Omni

OMNI (Organizational Model for Normative Institutions) [32] is an integrated framework for norms, structure, interaction and ontologies used to model OMAS. It is one of the most complete organization models. It combines two other models: OperA [30] and HarmonIA [114]. The OMNI framework consists of a Normative Dimension, an Organizational Dimension, and an Ontological Dimension, each of which has an Abstract, Concrete, and Implementation Level. The Abstract Level defines the main objectives of the organization. The Concrete Level refines the definitions of the Abstract Level further by defining the norms and rules of the organization, the roles in the organization, landmarks, and concrete ontological concepts. And finally, the Implementation Level implements the definitions from the Concrete Level.

One of the main strength of OMNI is that it can model both closed and open systems. Moreover, OMNI proposes a formal semantic which ensures consistency between the different organizational aspects of a system.

2.3 Organization Model for Adaptive Computational System

While there has been several organization models proposed, none have been specifically targeted towards providing a general mechanism that allows the system to reorganize in order to adapt to its environment and changing capabilities.

The Organization Model for Adaptive Computational Systems (OMACS) provides the foundation for organization-based multiagent metamodel in which the analysis and design concepts are directly related to run-time concepts.

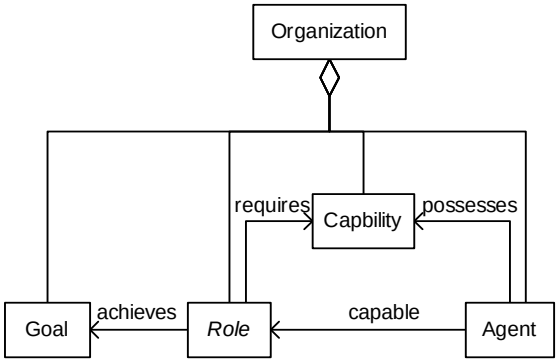

Figure 2.1. Simplified OMACS metamodel

Essentially, OMACS defines the required organizational structure that allows multiagent teams to reconfigure autonomously at runtime, thus enabling them to cope with unpredictable situations in a dynamic environment. Specifically, OMACS specifies the type of knowledge required for a multiagent system to be able to reason about its own state and configuration. Hence, multiagent teams are not limited by a predefined set of configurations and can have the appropriate information about their team, enabling them to reconfigure in order to achieve their team goals more efficiently and effectively. During the design of an OMACS-based system, the designer only provides high-level guidance about the organization, which then allows the system to self-configure based on the current goals and team capabilities. These characteristics make OMACS ideal for designing adaptive multiagent systems.

2.3.1 The OMACS metamodel

The OMACS metamodel is the metamodel upon which adaptive systems are designed. Figure 2.1 shows a simplified OMACS metamodel. Only the entities discussed in this dissertation are shown. OMACS defines an organization as a set of goals that the team is attempting to accomplish, a set of roles that must be played to achieve those goals, a set of capabilities required to play those roles, and a set of agents who are assigned to roles in order to achieve organization goals. In essence, each organization is an instance of the OMACS

metamodel presented in Figure 2.1 and is subject to all the constraints defined by OMACS. At runtime, the assignments of agents to play roles to achieve goals represent the key functionality that allows the system to be autonomous. There are more entities defined in OMACS that are not relevant for this dissertation. The reader is referred to [28] for the complete model.

2.3.2 Goals

Goals describe a desired state of the world and thus provide a high-level description of what the system is supposed to do [100]. Typically, each organization has a top-level goal that is decomposed into sub-goals. Eventually, this top-level goal is refined into a set of leaf goals that are pursued by agents in the organization. The set of all organizational goals is denoted as G. The active goal set, Ga, is the current set of goals that an organization is currently trying to achieve. Ga changes dynamically as new goals are created or existing goals are achieved.

2.3.3 Roles

Roles are a high-level description of the behavior required to achieve particular goals [38]. In OMACS, each organization has a set of roles that it can use to achieve its goals. The *achieves* function, which associates a score between 0 and 1 to each ⟨goal, role⟩ pair, tells how well that particular role can be used to achieve that goal (1 being the maximum score). In addition, each role requires a set of capabilities and agents must possess all the required capabilities to be considered as a potential candidate to assume that role.

2.3.4 Capabilities

In OMACS, *capabilities* are fundamental in determining which agents can be assigned to what roles in the organization [82]. In fact, agents are capable of playing a role only if they posses all the required capabilities. However, the decision whether or not a capable agent is actually going to assume a role is made at runtime. Agents may possess two types of capabilities:

hardware capabilities like actuator or effectors, and *software capabilities* like computational algorithms or resources.

2.3.5 Agents

OMACS *agents* are computational systems that can communicate with each other and play roles that match their capabilities [28]. Each agent is responsible for managing its own state and its interactions with the environment and with other agents. Once the system assigns a goal and role, the agent determines the low-level behavior necessary to fulfill the role and achieve the goal. This low-level behavior is generally provided either as part of the role definition or by a unique agent behavior specified by the designer. To capture a given agent's capabilities, OMACS defines a *possesses* function, which maps each ⟨agent, capability⟩ pair to a value between 0 and 1, describing the quality of the capability possessed by an agent (1 representing the maximum quality).

2.3.6 Assignment Process

In OMACS, a tuple ⟨a,r,g⟩ represents the assignment of agent a to play role r in order to achieve goal g. The *assignment set*, denoted by Φ, represents the set of all the current assignments in the organization.

The set of active goals along with the agents and their capabilities can change over time. For this reason, the process of assigning agents to play roles in order to achieve specific goals is not predefined but rather performed dynamically at runtime. This process takes into consideration the quality of each capability possessed by agents along with how well roles can achieve goals. For example, if a new goal is instantiated within the organization, a greedy algorithm could compute a new assignment by first choosing the best role for that goal then the best agent capable of playing the chosen role. However, OMACS does not prescribe any particular algorithm for computing assignments and several algorithms been investigated for this purpose [126].

2.4 Agents-Oriented Software Engineering

As agent systems evolved from agent-centered to organization-based, there was a need for developing methodologies that would help with the design of such system. Early agent-oriented methodologies focused primarily on the individual agents. A review of agent oriented methodologies is provided in [62, 121]. On the other hand, Organization-based Multiagent Systems (OMAS) have been viewed as an effective paradigm for addressing the design challenges posed by complex systems [39, 125]. In those systems, the organizational perspective is the main abstraction, which provides a clear separation between agents and system, allowing a reduction in the complexity of the overall system. To ease the development of OMAS, several methodologies have been proposed [36]. While a comprehensive study of each methodology is out of the scope of this dissertation, I give a brief overview of some of the most established methodologies in the following subsections.

2.4.1 Gaia

The Gaia methodology was one of the first methodologies proposed for the design and development of multiagent systems [123, 125]. Gaia encompasses life cycle phases from the analysis phase to the design phase. It adopts an organizational metaphor where each agent may play a variety of roles and where the agents may need to cooperate with each other to accomplish a common organizational goal. The Gaia methodology defines an agent based upon the roles it can assume. Each role is specified by four attributes: responsibilities, permissions, activities, and protocols. Responsibilities determine the functionality of a role and are divided into liveness and safety properties. Liveness properties describe what the agent that has been assigned to the role must do. Safety properties describe a behavior that the agent must maintain across all states of execution of the role. In order to realize responsibilities, a role has a set of permissions that identify the resources that are available to that role. Activities are computations associated with a role and may be carried out by the agent without interacting with other agents. Finally, protocols define role interactions. Moreover, organizations in Gaia are characterized by organizational rules, organizational structures and organizational patterns. Organizational rules are constraints

imposed on roles and protocols. Organizational structures cover the *topology* and the *control* of the organization. Organizational patterns represent predefined reusable structures.

2.4.2 Tropos

The Tropos methodology [12] is based on the notions that agents have goals and plans (according to the BDI architecture [96]) and covers all phases of software development from early requirements engineering to actual implementation. The methodology consists of four main phases: Early Requirements, Late Requirements, Architectural Design and Detailed Design. Tropos adopts the i* organizational modeling framework [124] for modeling requirements through the creation of goal model diagrams. The Early Requirements identify the environment in which the system would function and the Late Requirements analyses the functional and non-functional requirements of the system. The Architectural Design identifies the various components of the system based on the functional requirements whereas the Detailed Design specifies the internals of those components. Tropos is supported by a visual modeling tool called Tool for Agent Oriented Modeling for Eclipse (TAOM4E) [84] that can be used to create the various diagrams.

2.4.3 Prometheus

Prometheus is an agent oriented methodology that defines a detailed process for analysis, design and implementation of multi agent systems [90]. The agent model used within Prometheus closely resembles the BDI model of agenthood with several additions such as messages and percepts for improved representation of practical systems. Prometheus consists of three main phases namely System specification, Architectural design and Detailed Design with each phase consisting of artifacts that define related aspects of the agent model. The System specification phase defines the system goals, scenarios, basic system functionalities (called roles), inputs (referred to as percepts) and actions performed by the system. The outcome of the System specification is used within the Architectural design phase to determine the types of agents required and their interactions. The Detailed design phase looks at the internal details of each agent type with respect to constructs such as events, plans and beliefs. Prometheus is

supported by the Prometheus Design Tool (PDT), a visual tool for generating various Prometheus models using the graphical notation [112]. PDT facilitates the creation of various diagrams in all three phases of the methodology and also includes features that assist the user with the notation.

2.4.4 INGENIA

INGENIAS [93] provides a graphical notation for modeling multi-agent systems and a methodology based on the Unified Software Development Process (USDP) [63] for guiding the application development process. INGENIAS takes a model driven approach where the methodology assists in the creation of models of a MAS from which execution code is automatically generated using tools. In INGENIAS a multi-agent system (MAS) is defined with five meta-models, namely Organization, Agent, Goals/tasks, Interactions and Environment, which provide five different viewpoints of MAS. Each viewpoint is represented using the INGENIAS graphical notation.

2.4.5 PASSI

PASSI (Process for Agent Societies Specification and Implementation), is a requirement-to-code methodology for designing and developing multi-agent societies [21]. PASSI characterizes the system development using five process components (models) that are divided into phases and described using UML diagrams. Chella et al. [20] develop an agile version (Agile PASSI) that exploits the features of reusable patterns. In order to facilitate implementation, a PASSI ToolKit (PTK), was developed as a plug-in for IBM's commercial tool Rational Rose [23].

2.5 Organization-based Multiagent System Engineering Process Framework

In this section, I give a brief overview of the Organization-based Multiagent System Engineering (O-MaSE) Process Framework [44]. O-MaSE is a framework that allows designers to create custom agent-oriented development processes. This custom agent-oriented process is generated following a process metamodel and then instantiated from a set of method fragments and guidelines by using a method engineering approach [13]. Method engineering is an approached that has been proposed to allow the development of software methodologies from several fragments.

Thus, O-MaSE defines a metamodel, a repository of method fragments and a set of guidelines. The O-MaSE metamodel defines general concepts used in multiagent systems along with their relationships and is based on an organizational approach. In fact, there is a 1:1 projection of the OMACS metamodel onto the O-MaSE metamodel, which allows systems developed using appropriate O-MaSE method fragments to produce valid instances of the OMACS metamodel. Organizations developed using an O-MaSE compliant process produce a set of models that specify valid instances of the O-MaSE metamodel. Method fragments are a set of activities, tasks and work products extracted from existing agent methodologies and stored in a repository. They are later combined to create a methodology instance that is used on a project. O-MaSE method fragments currently cover the requirements, analysis and design phases of a multiagent development lifecycle. Finally, O-MaSE Process Construction Guidelines specify a set of constraints that must be maintained when combining method fragments to create valid O-MaSE processes.

Therefore, designing a custom O-MaSE compliant process requires process engineers to select a set of methods that suit their needs from the repository and combine them into a complete process such that the constraints of each fragment are satisfied. O-MaSE provides some guidelines to help choose fragments but does not guarantee that all processes created are necessarily efficient. However, the O-MaSE Process Framework does allow designers to develop rigorous and repeatable processes suitable for their particular needs.

The O-MaSE Process Framework is supported by the agentTool Process Editor, which is part of the agentTool III (aT^3) development environment. The agentTool Process Editor (APE)

allows process designers to create custom O-MaSE processes, which can then be analyzed and designed using the (aT^3) development environment. Further details on aT^3 and APE can be found in [43].

2.6 Modularity in Software Engineering

Modularity is an important Software Engineering principle. It applies the principle of *separation of concerns* by dividing a complex system into simpler and more manageable modules. Modularization involves a *composition* phase in which modules need to be put together to form a larger system. There have been a lot of composition mechanisms proposed at various stage of the software lifecycle, but mainly at the design and implementation phase. Hence, composition concerns various concepts (architecture, models, components, code, etc…). The main challenge of the composition is to determine how to integrate the selected concepts in order to obtain the desired results [64]. For that, several integration mechanisms have been proposed based on the concepts.

Software architectures are most commonly composed using architecture description languages (ADL), which provide a notational foundation for representing architectures. Architectures are then connected using architectural connectors [83].

Component-based systems can be composed according to several component models that fall into three categories [75]:

- Models that only use programming languages. For example JavaBeans and EJB are solely defined in Java [81];
- Models in which an Interface Definition Language (IDL) is used. For example COM and .NET use Microsoft IDL [98], while Corba Component Model (CCM) uses OMG IDL [6].
- Models in which components are defined by ADLs like UML2.0 [88] and KobrA [2] that both use the UML notation.

In Service oriented frameworks [33], web services are similar to components and can be composed via service composition languages like BPML [94], BPEL4WS [70], WS-CDL [71].

In this dissertation, I propose a composition process that combines design models. Although this composition approach concerns the domain of organization-based multiagent systems, it is similar to other composition approaches proposed in the literature in the sense that they all require a connection mechanism to bind two entities. In fact, as pointed out by Achermann [1] and Jeanneret [64], composition mechanisms have various semantics but all rely on the existence of some kind of connector, most of the time expressed as a language or other simpler mechanisms like function calls.

2.7 Summary

In this chapter, I outlined of the basic concepts necessary in order to provide a better understanding of the work presented in this dissertation. I defined the concepts of agent and multiagent systems and reviewed some of the most notable organization based models and methodologies for OMAS. In particular, I described the OMACS model, on which my work is based. OMACS propose an organizational model to design adaptive systems. It defines the required organizational structure that allows multiagent teams to reconfigure autonomously at runtime. OMACS is supported by the O-MaSE process framework, which provides a rigorous methodology for developing OMACS-based systems. Finally, I presented some compositional frameworks from other domains. These frameworks are very diverse but present some high-level similarities in the way the composition is done.

In the next chapter, I introduce the concepts of category theory that are necessary to understand the rest of this dissertation.

CHAPTER 3 - CATEGORY THEORY PRELIMINARIES

"Let all laws be clear, uniform and precise: to interpret laws is almost always to corrupt them. "
—*Francois Voltaire*, Philosophical Dictionary (1764)

""When I use a word," Humpty Dumpty said in rather a scornful tone, "it means just what I choose it to mean - neither more nor less.""
—*Lewis Carroll, Alice in Wonderland*

Category theory is a mathematical tool designed to describe various structural concepts from different mathematical fields in a uniform way. In computer science, category theory is very helpful and can be applied in areas such as algebraic specification, type theory, automata theory, programming language semantics, and graph rewriting. In this chapter, I define the category theory notions necessary to understand the work proposed in this dissertation. An extensive introduction to category theory can be found in [5]. I first introduce the notion of graphs and graph homomorphisms. Then I give the formal definition of a category. Finally I present some categorical constructs.

3.1 Graphs and Graph Homomorphisms

Most design models can be viewed as graphs. Hence I start by giving some graph related definitions. A good introduction to Graph Theory can be found in [47]. The following definitions are adapted from [51] and [47].

Definition 3.1: Graph

A *Graph* $G = \langle V, E \rangle$ is a mathematical structure consisting of two finite sets V and E. The elements of V are called *vertices* (or *nodes*) and the elements of E are called *edges*. Each edge has two vertices associated to it, which are called endpoints. If two vertices u and v are joined by an edge, this edge is denoted $|u,v|$.

G is a *directed graph* if the set of edges contains ordered pairs of vertices. A *path* represents a sequence of vertices such that from each vertex there is an edge to the next vertex in the sequence. A *cycle* is a path such that the start vertex and end vertex are the same. A graph is called *connected* if every pair of distinct vertices in the graph can be connected through some path.

Definition 3.2: Rooted Tree

A *rooted tree* $T = \langle V, E, r \rangle$ is a connected acyclic graph $\langle V, E \rangle$ in which vertex r has been designated the root.

Definition 3.3: Graph Homomorphism

Given two graphs $G_1 = \langle V_1, E_1 \rangle$ and $G_2 = \langle V_2, E_2 \rangle$, a *graph homomorphism* h from G_1 to G_2 consists of two functions $f : V_1 \rightarrow V_2$ and $g : E_1 \rightarrow E_2$, such that:

- if $e = |a,b| \in E_1$ then $g(e) = |f(a), f(b)| \in E_2$ (preserve edges)

Definition 3.4: Tree Homomorphism

Given two rooted trees $T_1 = \langle V_1, E_1, r_1 \rangle$ and $T_2 = \langle V_2, E_2, r_2 \rangle$, a *tree homomorphism* f from T_1 to T_2 consists of two functions $f : V_1 \rightarrow V_2$ and $h: E_1 \rightarrow E_2$, such that:

- $f(r_1) = r_2$ (preserve root)
- if $e=|a,b| \in E_1$ then $h(e) = |f(a),f(b)| \in E_2$ (preserve edges).

3.2 Category: definition and examples

Category theory is a mathematical tool originally used to establish a uniform framework in order to study the relations between different mathematical structures appearing in various areas of mathematics such as algebra, topology and logic [46, 79].

There is a clear difference in approaches between set theory and category theory. Set theory characterizes a mathematical object by describing its inner structure, its members. However, category theory takes a different approach. Mathematical objects are black boxes only defined by their interactions with other objects. For this reason, Fiadeiro [40] talks about "the social life of objects" as the basis of category theory. Hence, category theory can be viewed as more "abstract" than set theory. Since in this language there is no way to look at the internal membership structure of objects, all the concepts must be defined by their relations with other objects, and these relations are established by the existence and the equality of particular morphisms. In computer science, category theory is very helpful and can be applied in areas such as algebraic specification, type theory, automata theory, programming language semantics, and graph rewriting [5].

In this section, I briefly introduce the key notions of category theory that are used in this research. Those preliminaries do not constitute a proper introduction to category theory. The reader is referred to [46, 79] for a more elaborate introduction to category theory concepts. A computer science introduction to category theory is provided in [3, 5, 40, 101]. The definitions in this section are adapted from [40].

Definition 3.5: Category

A *category* C is given by a collection of *objects* and a collection of *morphisms* ("arrows") that have the following structure:

- Each morphism has a domain and a codomain that are objects; we write $f : X \to Y$ if $X = dom(f)$ and $Y = cod(f)$;
- Given two morphisms f and g such that $cod(f) = dom(g)$, the composition of f and g, written $g \circ f$, is defined and has domain dom(f) and codomain cod(g);
- The composition is associative, that is: given $f : X \to Y$, $g : Y \to Z$ and $h : Z \to W$, $h \circ (g \circ f) = (h \circ g) \circ f$;

- For every object X there is an identity morphism $id_x : X \rightarrow X$, satisfying $id_x \circ g = g$ for every $g : Y \rightarrow X$ and $f \circ id_x$ for every $f : X \rightarrow Y$.

Essentially, a category is a mathematical structure that has objects and morphisms, with an associative composition operation on the morphisms and an identity morphism for each object. In other words, categories are graphs (with multiple directed edges) with a composition and identity structure.

Example 3.1

The basic example of a category is that of sets, SET, whose objects are sets and whose morphisms are total functions. The composition operation is the composition of functions, which we know is associative. Identities are simply identity functions that map each set to itself. Figure 3.1 shows an example of a category with three sets.

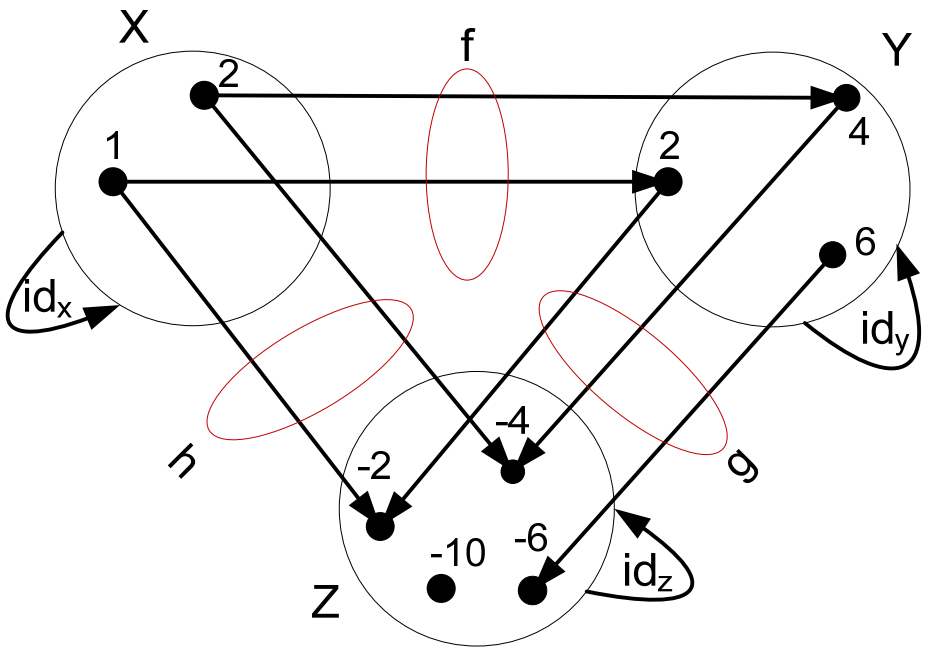

Figure 3.1. Example of category with sets

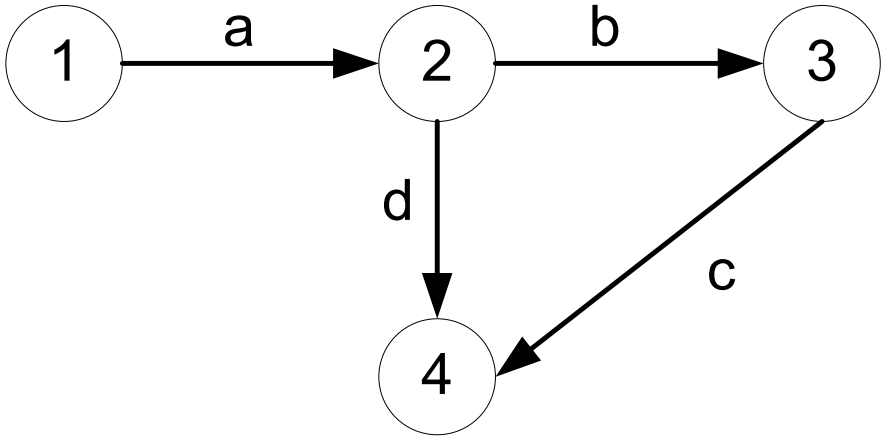

Figure 3.2. Example of directed graph representing a category

Proof:

Let us verify that sets X, Y, Z along with morphisms f, g, h, id_x, id_y, id_z form a category.

Identities

id_x, id_y, id_z are the identity functions for X, Y and Z respectively.

Composition

We have: $g \circ f = h$, $f \circ id_x = f$; $g \circ id_y = g$; $h \circ id_x = h$;

Associativity

We have: $id_z \circ (g \circ f) = g \circ f$. And $(id_z \circ g) \circ f = g \circ f$. Hence $id_z \circ (g \circ f) = (id_z \circ g) \circ f$. □

Example 3.2

Any directed graph can be represented as category. Nodes in the graph represent the objects, and paths represent the morphisms. Concatenating the paths to bring about longer paths is equivalent to composition.

Proof:

Let us verify that the graph in Figure 3.2 represents a category. A path is denoted by the concatenation of all the edges' labels. Hence a path involving edges a, b, c will be abc.

Identities

The empty paths constitute the identity morphisms.

Composition

We have: $b \circ a = ab$, $d \circ a = ad$; $c \circ b = bc$;

Note: I have omitted all the trivial paths involving identity morphisms.

Associativity

We have:

$c \circ (b \circ a) = c \circ ab = abc$. And $(c \circ b) \circ a = bc \circ a = abc$. Hence $c \circ (b \circ a) = (c \circ b) \circ a$. □

Definition 3.6: Isomorphism

A morphism $f: X \to Y$ in a category C is said to be an *isomorphism* if there exits a morphism $g : Y \to X$ of C such that : $g \circ f = id_x$ and $f \circ g = id_y$. The morphism g is unique and denoted f^{-1}.

Two objects X and Y are said to be *isomorphic*, denoted $X \cong Y$, if there exists an isomorphism between them. In category theory, isomorphic object are considered the "same", hence all constructions are defined up to isomorphism.

Example 3.3

In the category SET, isomorphisms are bijective functions and isomorphic objects are sets with the same cardinal number.

Proof:

In fact, if $f: X \to Y$ is a bijective function then $f^{-1} \circ f = id_x$ and $f \circ f^{-1} = id_y$. Moreover, all the sets need to have the same cardinal number in order for all morphisms to be bijections. □

3.3 Categorical Constructions

Definition 3.7: Initial Object

An object X of a category C is said to be *initial* if there is exactly one morphism from X to every other objects in the category.

Hence, any two initial objects in a given category are isomorphic. Similarly, any object isomorphic to an initial object is also initial (see proof in [40]).

Example 3.4

In SET, the initial object is the empty set.

Proof:

The empty set can be mapped to any other set only by the empty function. The empty function is a function whose domain is the empty set and whose codomain is any set. Hence for each set X, there is one empty function f_x such that $f_x : \varnothing \to X$. □

Definition 3.8: Terminal Object

An object is Y of a category C is said to be *terminal* if there is exactly one morphism from every other object in the category to Y.

Hence, any two terminal objects in a given category are isomorphic. (see proof in [40])

Example 3.5

In SET, the terminal objects are the singletons.

Proof:

There is only one way to map any given set to a singleton; it is by mapping all the elements of the source to the singleton. Hence, there is a unique morphism from any set to a singleton. □

Definition 3.9: Sum

Let C be a category and X, Y objects of C. An object Z is a *sum* (or *coproduct*) of X and Y with morphism $f : X \to Z$ and $g : Y \to Z$ (called injections) iff for any object V and

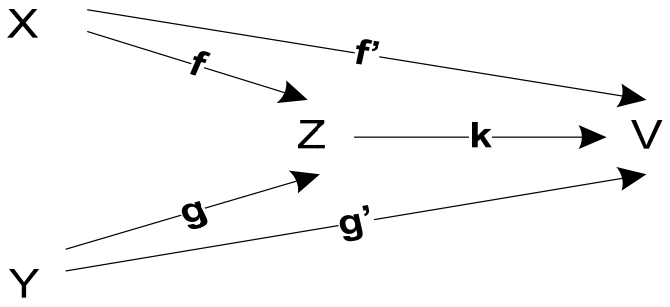

Figure 3.3. Sum in a category

pair of morphisms f': X→V and g' : Y → V there is a unique morphism k : Z → V in C such that k ○ f = f' and k ○ g = g' .

The *sum* is denoted by x+y and is unique up to isomorphism. This definition is illustrated in Figure 3.3.

Example 3.6

In the category SET, the disjoint union X⊕Y is the sum of X and Y.

Proof: (adapted from [40]).

Existence: consider an arbitrary object V and pair of morphisms f' : X → V, g' : Y → V. Define k : X⊕Y → V as follows: given A ∈ X⊕Y, let k(A)=f(a) if A = f(a) with a ∈ X and k(A) = g(a) if A=g(a) with a ∈ Y. This is a proper definition of a total function because, on the one hand, every element of X⊕Y is either in the image of X through f or the image of Y through g and, on the other hand, these two images are disjoint (which removes any conflict of choice between which case to apply). The conditions k ○ f = f' and k ○ g = g' are satisfied by construction.

Uniqueness: given any other total function k': X⊕Y, the conditions k' ○ f =f' and k' ○ g = g' define k' completely (and equal to k). □

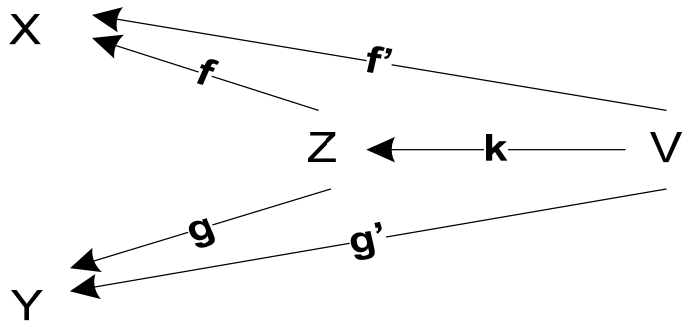

Figure 3.4. Product in a category

Definition 3.10: Product

Let C be a category and X, Y objects of C, an object Z is said to be a *product* of X and Y with morphisms f : Z → X and g : Z → Y (called projections) iff for any object V and pair of morphisms f': V → X, g' : V → Y of C there is a unique morphism k : V → Z in C such that f ∘ k = f' and g ∘ k = g'.

This definition is illustrated in Figure 3.4.

Example 3.7

In the category SET, the Cartesian product X × Y (with corresponding projections) is the product of X and Y. This proof is similar to the proof of disjoint union in SET.

Definition 3.11: Pushout

Let f : X → Y and g : X → Z be morphisms of a category C. A *pushout* of f and g consists of an object W and a pair of morphisms f': Y → W and g': Z → W such that:

- f' ∘ f = g' ∘ g

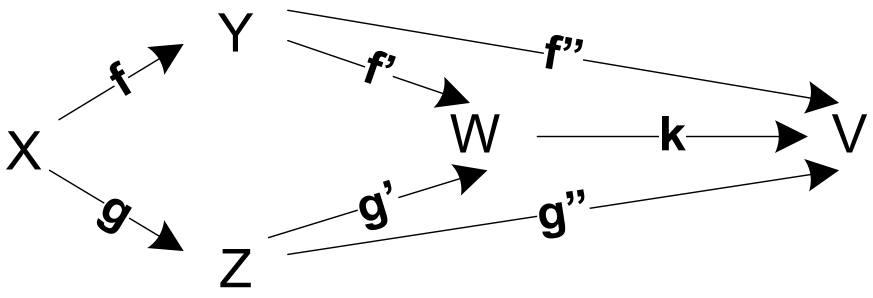

Figure 3.5. Pushout in a category

- For any other morphisms f'': Y → V and g'': Z → V such that f'' ∘ f = g'' ∘ g, there is a unique morphism k: W → V in C such that k ∘ f' = f'' and k ∘ g' = g'' .

This definition is illustrated in Figure 3.5.

Examples 3.8

Figure 3.6 shows a pushout in SET. The proof that this diagram represents a pushout is outlined as follows. We have f: X → Y and g : X → Z , f': Y → W and g': Z → W four functions such that:

f = {(2,2), (4,4)}, g = {(2,2), (4,4)}, f' = {(2,2), (4,4), (6,6)}, g' = {(2,2), (4,4), (3,3)}.

It easy to see that f' ∘ f = g' ∘ g. Moreover, assume that there exists two functions f'': Y → V and g'': Z → V such that f'' ∘ f = g'' ∘ g. If k: W → V is a function such that k ∘ f' = f'' and k ∘ g' = g'', then the fact that f'' ∘ f = g'' ∘ g leaves no choice for the choice of k, which ensures uniqueness.

Remark that the object W computed by pushout of f and g in Figure 3.6 is just the union of Z and Y.

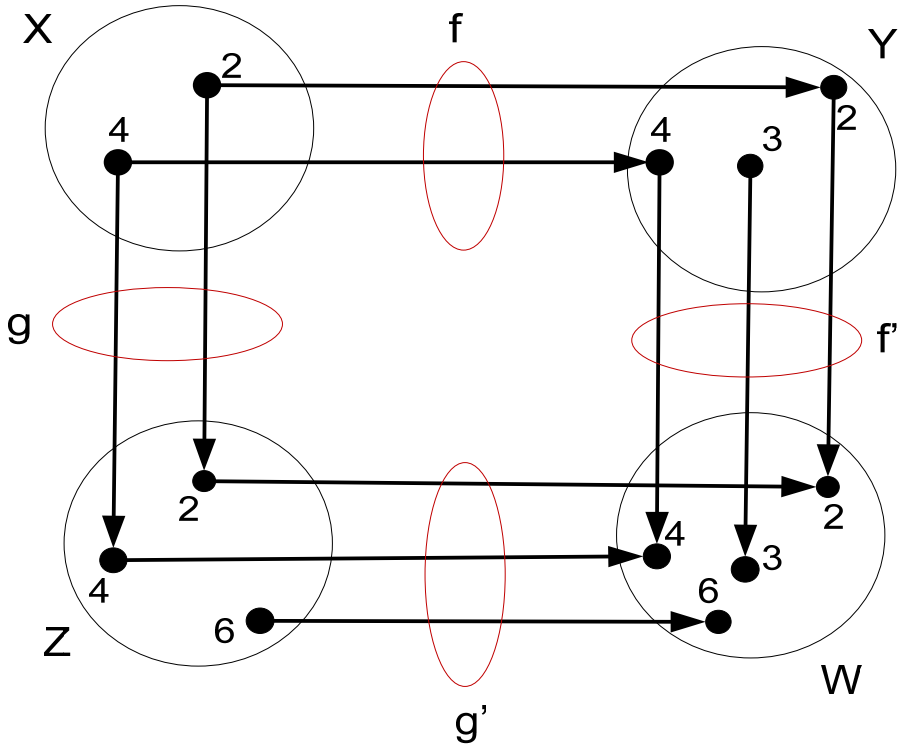

Figure 3.6. Example of Pushout in category SET

In general, the pushout allows us to merge objects based on their relationships without violating the requirements that are imposed on their structure and adding any unnecessary duplication of elements. In fact, as pointed out by Goguen [46], pushouts represent a construction to interconnect systems to form a larger systems.

3.4 Summary

In this chapter, I introduced some of the important concepts of category theory that are be used in this dissertation. I have introduced the notion of graph and graph homomorphisms. I have also defined what a category is and what construct can be used on categories. In the next chapter,

I represent design models as first as graphs, then as categories. This formal representation of design model is used in a categorical construction in order to establish a composite design model.

I chose a category theory approach because the composition of two objects in a category is constructive. Hence, if the construction of the composition satisfies the conditions to be done as a categorical construction, this construction becomes trivial and can be automatically derived. Alternatively, due to the complexity of the models, giving the concrete details of the same construction in set theory is cumbersome and proving that the construction produces the correct result would be tedious. However, proofs in category theory are often short and elegant as they follow common construction patterns.

CHAPTER 4 - COMPOSITION OF MULTIAGENT ORGANIZATIONS

"A theory is the more impressive the greater is the simplicity of its premises, the more different are the kinds of things it relates and the more extended the range of its applicability." — *Albert Einstein*

"I know what you're thinking about, but it isn't so, nohow. Contrarywise, if it was so, it might be, and if it were so, it would be. But, as it isn't, it ain't. That's logic." —*Lewis Carroll, Alice in Wonderland*

Design models are often created by different teams and need to be merged into one main design. Similarly, complex designs can be decomposed into smaller more manageable design models and then integrated later. Moreover, several projects might require the reuse of some previous designs. Unfortunately, most of the current Agent-Oriented Software Engineering (AOSE) methodologies simply suggest the decomposition of organization designs and fail to provide a rigorous process to recombine them. In most cases, the designer informally uses the agent interaction mechanisms to integrate organization designs.

In this chapter, I define a general framework for the formal composition of OMAS design models. The composition is done solely at the design level, resulting in a single composite organization design that can then be used at runtime. The main organizational models used in this work are the goal and role models, which are key models that provide OMAS their adaptability. These models (in various forms) have been used in several OMAS framework. My work is based on the Organization Model for Computational Adaptive Systems (OMACS) [26]. There are many other organizational models for OMAS [36] and the approach proposed in this

research could well be adapted for any of them. OMACS proposes a formal framework for describing organizational models for MAS and is supported by a rigorous methodology tailored from the O-MaSE process framework [44]. OMACS defines an organization as a set of goals (G) that the organization is attempting to accomplish, a set of roles (R) that must be played to achieve those goals, a set of capabilities (C) required to play those roles and a set of agents (A) who are assigned to roles in order to achieve organizational goals. There are more entities defined in OMACS that are not relevant for this dissertation. The reader is referred to [28] for the complete model.

The goal models and role models represent the persistent part of a multiagent organization, the organization structure [39], which can then be populated later with heterogonous agents to produce a dynamic organization. Hence, this work does not deal with the actual agents that will participate in the organization.

I propose a framework allowing the composition design models, by treating composition as an algebraic operator over models and their relationships. Using a category theoretic approach, I formalize the goal model, the role model, and finally the entire organization. I define each model as a category and then show that these models can be composed by computing their pushout in the appropriate category. By using this mathematical framework, I obtain a formal construction of the composition of organizations that is guaranteed to result in a correct organizational design.

4.1 Organizational Models

OMACS defines an organization as a set of goals (G) that the organization is attempting to accomplish, a set of roles (R) that must be played to achieve those goals, a set of capabilities (C) required to play those roles and a set of agents (A) who are assigned to roles in order to achieve organizational goals. The complete OMACS model is defined in [28]. In this dissertation, I use a generalization of the OMACS model and only consider the goals, roles and the relationship that exists between them. These entities represent the persistent part of the organization that can be populated with heterogonous agents to produce a dynamic organization. In the following subsections, I formally define the models that are used throughout this work.

4.1.1 Goal Model

In a typical multiagent organization, organizational goals and roles are organized in a goal tree [58, 74, 113, 119] and in a role model [69, 119, 125] respectively. For this dissertation, I chose to organize our goals using a Goal Model for Dynamic Systems (GMoDS) [27]. In a GMoDS goal model, goals are organized in a goal tree such that subgoals of a goal are either an OR-decomposition or an AND-decomposition of that goal. A goal represents a desirable state of a system and is represented by the tuple $g = \langle name, type \rangle$ where *name* is the name of the goal, and *type* represents the decomposition of the goal, which can be *OR, AND, or LEAF* (leaf goals are of type LEAF and cannot be decomposed). In addition, the GMoDS goal model contains two time-based relationships between goals: the *precedes* and *triggers* relations. We say *goal* g_1 *precedes goal* g_2, if g_1 must be satisfied before g_2 can be pursued by the organization. Moreover, during the pursuit of specific goals, *events* may occur that cause the instantiation of new goals. Instantiated goals may be parameterized to allow a context sensitive meaning. If an event e can occur during the pursuit of goal g_1 that instantiates goal g_2, we say g_1 *triggers* g_2 *based on e*.

I extend GMoDs [27] to incorporate the notion of external goals and internal goals. Internal goals (G_i) are the actual goals that the organizations try to achieve. They are organized in a tree. External goals (G_x) are just placeholders for goals from other organizations and they do not impact the satisfiability of a goal model as they are never be assigned to an agent. External goals are not part of the decomposition tree. They can only trigger internal goals and be triggered by internal goals. In addition, without loss of generality, I assume that all goal models have the same root. This root is represented by an empty AND goal called *generic root* (*g_root*). Formally, the goal model can be represented as mathematical structure composed of a rooted tree and a graph. The tree correspond to the AND-OR decompositions between goals. Its edges represent the *parent* relationship. The graph represents the *time-based* relationships between goals.

Definition 4.1: Goal Model

A *goal model* is a tuple GM = ⟨ G, ET, EG, g_root ⟩ where:

- G : set of *organizational goals* such that $G = G_x \cup G_i$, where G_x represents the set of *external goals* and G_i the set of *internal goals*. We have $G_x \cap G_i = \varnothing$;

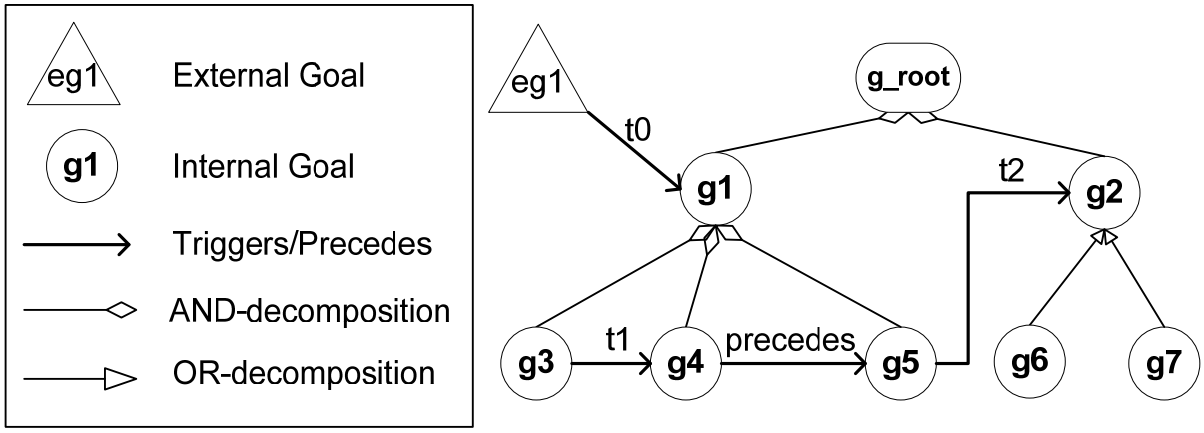

Figure 4.1. Goal Model Example

- ET $\in G_i \times G_i$: set of *parent* edges such that $\langle G_i, ET, g\_root\rangle$ is a rooted tree
- EG $\in G \times G$: set of *time-based* edges such that $\langle G, EG\rangle$ is a directed graph
- g_root $\in G_i$: the *root* of the goal model.

Given a goal model GM, the set $G_L \subset G_i$ represents the leaf goals. The rooted tree is called *induced tree* and the graph is called *induced graph*.

Moreover, I define three functions over the nodes goal model: parent, precedes and triggers.

Definition 4.2: Functions on goals

Given a goal model GM = $\langle G, ET, EG, g\_root\rangle$ and a set of events Ev, we have:

- parent: $G_i \rightarrow G_i$; defines the parent of a given goal
- precedes: $G_i \rightarrow 2^{Gi}$; indicates all the goals preceded by a given goal
- triggers: $Ev \rightarrow 2^{G \times G}$; $\langle g_1,g_2\rangle \in$triggers(e) iff g_1 triggers g_2 based on e.

Following this definition, we can characterize the internal and external goals as follows:

$G_i = \{g \in G \mid g\_root \in parent^*(g)\}$;

$G_x = \{g \in G - \{g\_root\} \mid parent(g) = \varnothing\}$.

Moreover, we have:

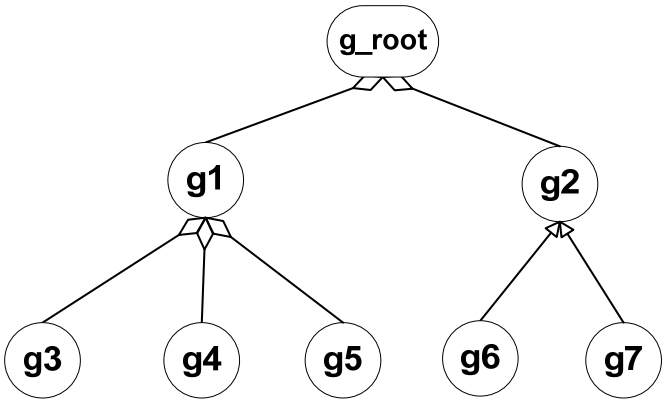

Figure 4.2. Example of Induced Tree

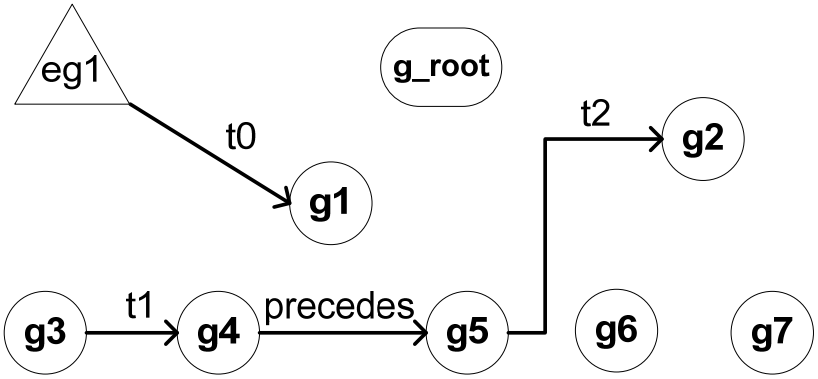

Figure 4.3. Example of Induced Graph

$ET = \{\langle g_1,g_2\rangle \in G_i \times G_i \mid g_1 = parent(g_2)\}$.

$EG = \{\langle g_1,g_2\rangle \in G \times G \mid (g_2 \in precedes(g_1)) \vee (\exists e \in Ev \mid \langle g_1,g_2\rangle \in triggers(e))\}$

Example 4.1

Figure 4.1 represents the goal model GM_example = ⟨G, ET, EG, g_root⟩, where:

- $G = \{g\_root, g_1, g_2, g_3, g_4, g_5, g_6, g_7, eg_1\}$ with $G_i = \{g\_root, g_1, g_2, g_3, g_4, g_5, g_6, g_7\}$ and $G_x = eg_1\}$
- $ET = \{ \langle g_1,g\_root\rangle, \langle g_2,g\_root\rangle, \langle g_3,g_1\rangle, \langle g_4,g_1\rangle, \langle g_5,g_1\rangle , \langle g_6,g_2\rangle , \langle g_7,g_2\rangle\}$

- EG = {⟨eg_1,g_1⟩, ⟨g_3,g_4⟩, ⟨g_4,g_5⟩, ⟨g_5,g_2⟩}
- root = {g_root}

Moreover, for each goal g in the goal model in Figure 4.1, the type of the goal (g.type) is defined based on the decomposition arrow. Hence, g_root.type=g_1.type=AND; g_2.type=OR; g_3.type = g_4.type = g_5.type = g_6.type = g_7.type = LEAF.

Figure 4.2 and Figure 4.3 respectively show the induced tree and the induced graph for this goal model. Note that an edge ⟨g_1,g_2⟩ represents a directed edge from g_1 to g_2.

4.1.2 Role Model

I also organize the roles using a role model that is essentially a set of roles connected by protocols. There are two types of roles: internal roles and external roles. *Internal roles* are roles that are defined inside the organization. *External roles* represent placeholder for roles from external organizations. They represent an interface to the outside world, which allows organizations to cater for interactions with unknown roles at design time. Eventually, either later on in the design or at runtime, external roles will be replaced by concrete roles (internal roles) from other organizations. Formally, a role model can be viewed as a directed graph having roles as nodes and protocol as edges such that an edge p from role r_1 to role r_2 would indicate a protocol p for which r_1 is the initiator and r_2 the responder. I assume that in a role model, protocols names are unique. This can be enforced by having a protocol naming scheme that takes into account the participants of that protocol. In addition, I assume that given two roles, there is at most one protocol between them. This assumption is valid as if there is more than one protocol between two roles, those protocols can be combined into one protocol having several alternate cases [59].

Definition 4.3: Role Model

A *role model* is a tuple RM = ⟨R, P, participant⟩ where:

- R: set of *roles*
- P: set of *protocols*
- participants: P→R × R ; indicates the pair of roles connected by a given protocol

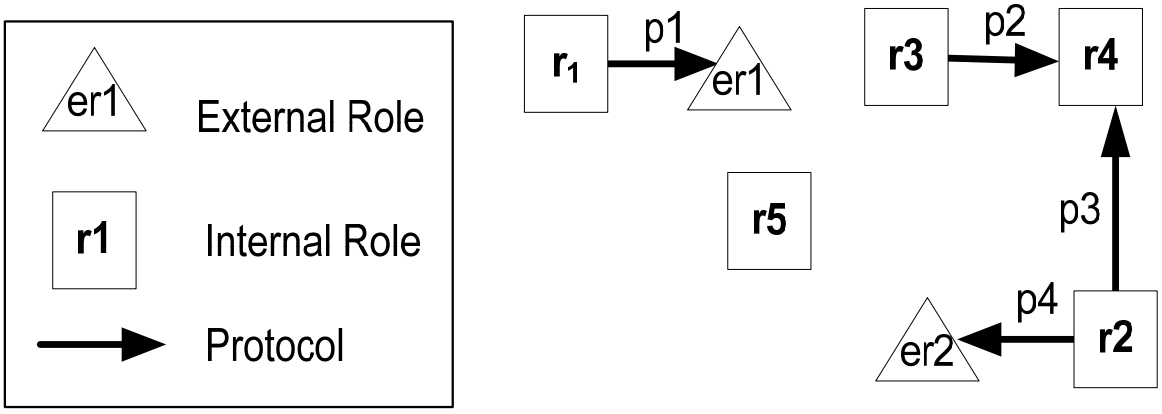

Figure 4.4. Example of Role Model

In addition, we have $R = R_i \cup R_x$ ($R_i \cap R_x = \varnothing$), where R_x represents the set of *external* roles and R_i the set of *internal* roles.

This role model corresponds to a directed graph having roles as nodes and protocols as edges.

Example 4.2

Figure 4.4 represents the role model RM_example = ⟨R, P, participant⟩, where:

- $R = \{ r_1, r_2, r_3, r_4, r_5, er_1, er_2\}$
- $P = \{ p_1, p_2, p_3, p_4\}$
- participant = $\{(p_1, \langle r_1, er_1\rangle), (p_2, \langle r_3, r_4\rangle), (p_3, \langle r_2, r_4\rangle), (p_4, \langle r_2, er_2\rangle)\}$

4.1.3 Organization Structure

Finally, I define an organization as a goal model and a role model such that each leaf goal is achieved by a role.

Definition 4.4: Organization

An *organization* is a tuple O = ⟨GM, RM, achieves⟩ where:

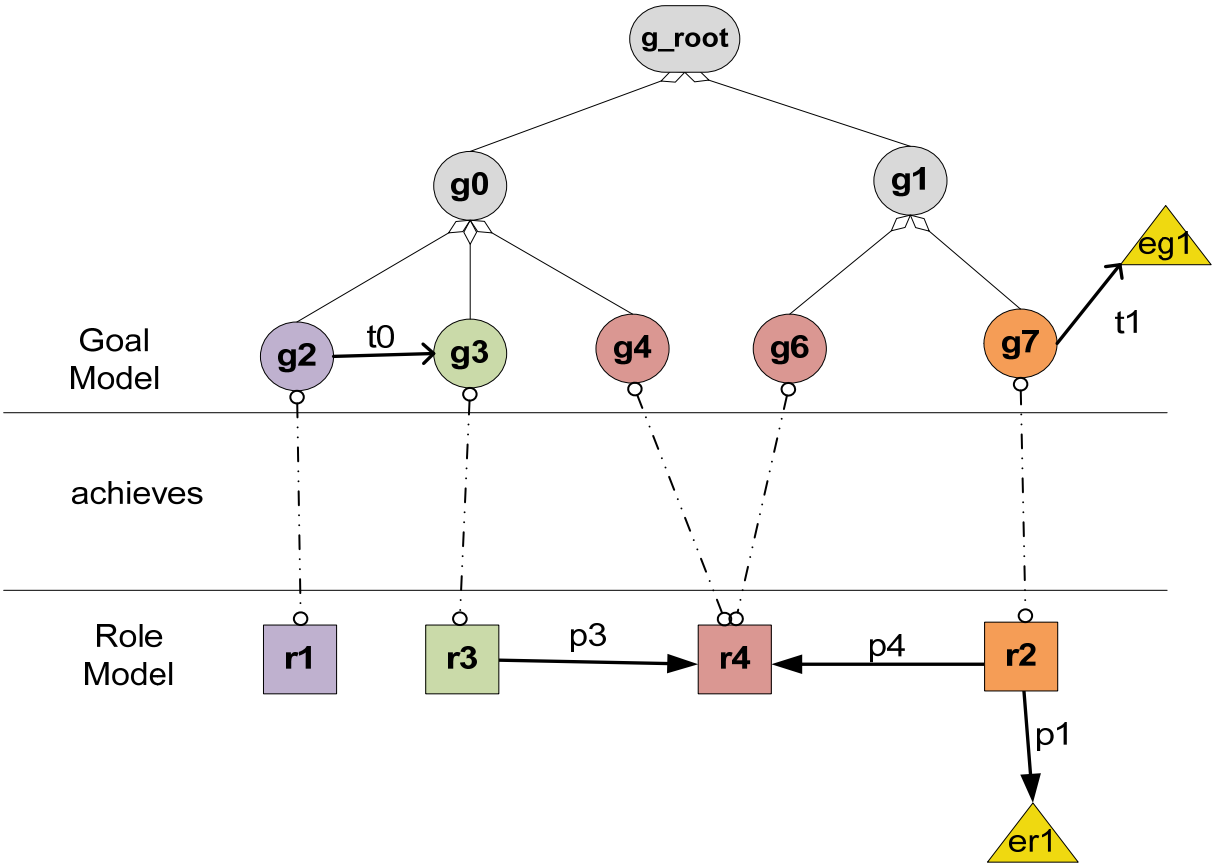

Figure 4.5. Example of Organization

- GM: Goal Model
- RM: Role Model
- achieves $\subseteq R \times G_l$: set of role-goal pairs such that the role achieves the goal.

Essentially, we can view an organization design as a directed graph with multiple node types and multiple edge types following the structure imposed by the goal and role model. The type of nodes and edges matches the corresponding organizational notions. Hence, the nodes can be of type *goal* or *role* while the edges can be of type *achieve*, *protocol*, *parent* or *time-based*.

Example 4.3

Figure 4.5 represents the organization ORG_example = ⟨GM, RM, achieves⟩, where:

- GM = ⟨G, ET, EG, g_root⟩ as depicted in the top part of Figure 4.5

- RM = ⟨R, P, participant⟩ as depicted in the bottom part of Figure 4.5
- achieves = $\{(r_1, g_2), (r_3, g_3), (r_4,g_4), (r_4,g_6), (r_2,g_7)\}$

4.2 Category of Goal Models

In this section, I define the category of Goal Model. I then introduce the key notions that allow the composition of goal model via pushout.

Definition 4.5: Goal Model Homomorphism

Given two goal models $GM_1 = \langle G_1, ET_1, EG_1, root_1 \rangle$ and $GM_2 = \langle G_2, ET_2, EG_2, root_2 \rangle$, a *goal model homomorphism* from GM_1 to GM_2 is a function $\mathbf{\Gamma = \langle f, g, h \rangle}$ where:

- $f : G_1 \rightarrow G_2$, such that $f(root_1) = root_2$
- $g : ET_1 \rightarrow ET_2$, such that if $|a,b| \in ET_1$ then $g(|a,b|) = |f(a),f(b)| \in ET_2$
- $h : EG_1 \rightarrow EG_2$, such that if $|a,b| \in EG_1$ then $h(|a,b|) = |f(a), f(b)| \in EG_2$.

Note that |a,b| denotes an edge. This distinct notation allows edges to be easily differentiated from any other tuples.

Proposition 4.6: Category of goal models

Goal models along with goal model homomorphisms define the **category GOAL_MODEL**.

Proof:

Let us prove that goal models along with goal model homomorphisms form a category. According to Definition 3.5, we need to identify the objects, morphisms and identity morphisms and verify that the composition of morphism exists and is associative.

Objects**:** The objects are goal models.

Morphisms**:** The morphisms are goal model homomorphisms.

Identity**:** The identity morphism is a function $id_{GM} = \langle id_G, id_{ET}, id_{EG} \rangle$ such that id_G is an identity function that maps each goal to itself, id_{ET} is an identity function that maps each induced tree edge to itself and id_{EG} is an identity function that maps each induced graph edge to itself.

Composition**:** Let X,Y,Z be three goal models and $\Gamma_1 = \langle f_1,g_1,h_1 \rangle$, $\Gamma_2 = \langle f_2,g_2,h_2 \rangle$ be two goal model homomorphisms such that: $\Gamma_1 : X \rightarrow Y$ and $\Gamma_2 : Y \rightarrow Z$.

The composition is defined as follows: $\Gamma_2 \circ \Gamma_1 = \langle f_2 \circ f_1, g_2 \circ g_1, h_2 \circ h_1 \rangle$.

Associativity**:** The goal model homomorphism consists of functions between sets. Hence, the assiociativity property is derived from the corresponding property of functions between sets [72].

□

Definition 4.7: Configurations of Goal Models

A *configuration of goal models* specifies all the mappings that are used to merge two goal models. Given two goal models GM_1 and GM_2, a *configuration of goal models* GM_1 and GM_2 is a triplet $\mathbf{config_{goal} = \langle GM_0, \Gamma_1, \Gamma_2 \rangle}$ where:

- GM_0 is a goal model
- Γ_1 is a goal model homomorphism from GM_0 to GM_1
- Γ_2 is a goal model homomorphism from GM_0 to GM_2

Definition 4.8: Goal model composition

The *composition* of two goal models GM_1, GM_2 over a goal model configuration $\langle GM_0, \Gamma_1, \Gamma_2 \rangle$ is the goal model resulting from the pushout of Γ_1 and Γ_2 in category GOAL_MODEL.

Example 4.4

The composition of goal models GM_1, GM_2 over the configuration $\langle GM_0, \Gamma_1, \Gamma_2 \rangle$ is depicted in Figure 4.6. Goal model GM_3 represents the composed model and it is obtained by pushout. The mappings for the functions comprised in the goal model homomorphisms are shown in Figure 4.7.

The elements for the pushout of goal models shown in Figure 4.7 are defined as follows:

Goal Model GM_0:

$GM_0 = \langle G_0, ET_0, EG_0, g\_root \rangle$,

$G_0 = \{ g\_root , g_2, g_4, g_5, g_6, g_6', g_7, g_8 \}$,

$ET_0 = \{ |g\_root,g_4|, |g_4,g_5|, |g_4,g_6| , |g_6,g_7| , |g_6,g_8| \}$

$EG_0 = \{ |g_2,g_6'|, |g_7,g_8| \}$

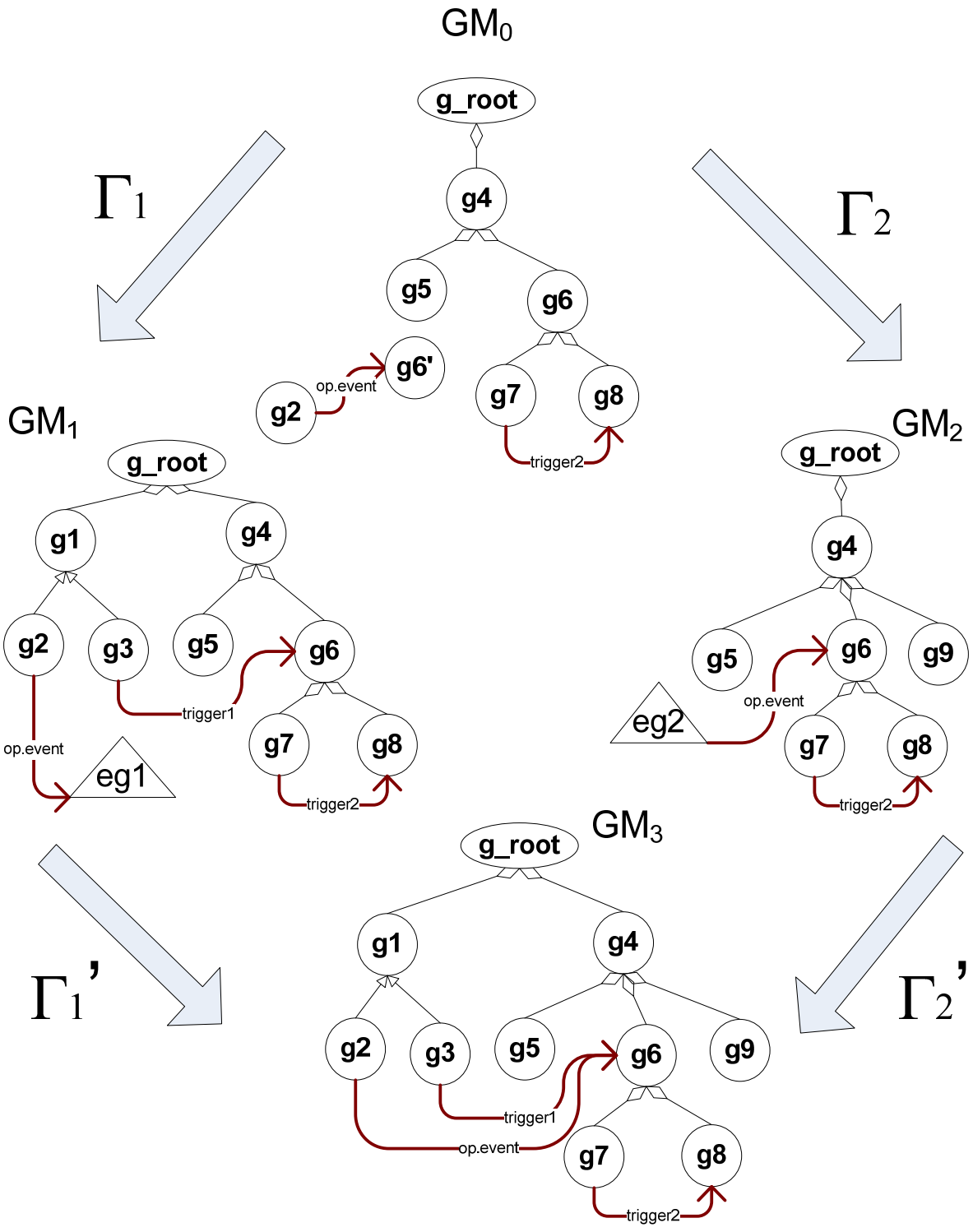

Figure 4.6. Overview of Pushout of Goal Models

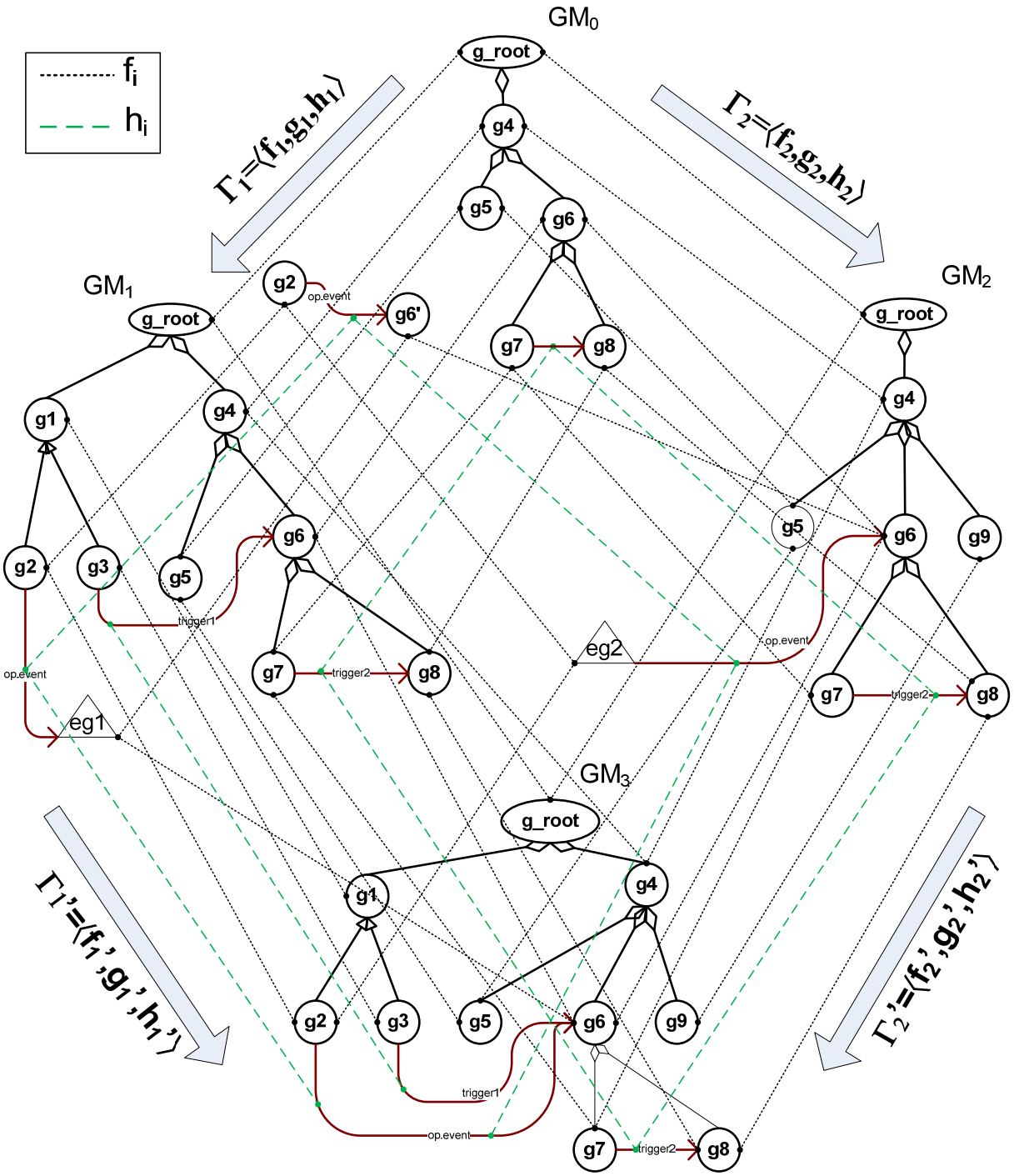

Figure 4.7. Pushout of Goal Models with detailed functions. Only functions mapping goals (f_i) and induced graph edges (h_i) are shown. Functions mapping tree edges (g_i) are not shown.

Goal Model GM_1:

$GM_1 = \langle G_1, ET_1, EG_1, g\_root \rangle$,

$G_1 = \{ g\_root , g_1, g_2, g_3, g_4, g_5, g_6, g_7, g_8, eg_1\}$,

$ET_1 = \{|g\_root, g_1|, |g\_root,g_4|, |g_1,g_2|, |g_1,g_3|, |g_4,g_5|, |g_4,g_6| , |g_6,g_7| , |g_6,g_8|\}$

$EG_1 = \{ |g_2,eg_1|, |g_3,g_6|, |g_7,g_8|\}$

Goal Model GM_2:

$GM_2 = \langle G_2, ET_2, EG_2, g\_root \rangle$,

$G_2 = \{ g\_root , g_4, g_5, g_6, g_7, g_8, g_9, eg_2\}$,

$ET_2 = \{ |g\_root,g_4|, |g_4,g_5|, |g_4,g_6| , |g_4,g_9| , |g_6,g_7|, |g_6,g_8|\}$

$EG_2 = \{ |eg_2, g_6|, |g_7,g_8|\}$

Goal Model GM_3:

$GM_3 = \langle G_3, ET_3, EG_3, g\_root \rangle$,

$G_3 = \{ g\_root , g_4, g_5, g_6, g_7, g_8, g_9, eg_2\}$,

$ET_3 = \{ \langle g\_root,g_4 \rangle, \langle g_4,g_5 \rangle, \langle g_4,g_6 \rangle , \langle g_4,g_9 \rangle , \langle g_6,g_7 \rangle, \langle g_6,g_8 \rangle\}$

$EG_3 = \{ \langle eg_2, g_6 \rangle, \langle g_7,g_8 \rangle\}$

Homomorphism Γ_1: (mappings f_1 and h_1 for Γ_1 are shown in Figure 4.7)

$\Gamma_1 = \langle f_1, g_1, h_1 \rangle$ such that: $f_1: G_0 \rightarrow G_1$, $g_1: ET_0 \rightarrow ET_1$, $h_1: EG_0 \rightarrow EG_1$. We have:

- $f_1 = \{\langle g\_root, g\_root \rangle, \langle g_4, g_4 \rangle, \langle g_5, g_5 \rangle, \langle g_6, g_6 \rangle, \langle g_7, g_7 \rangle, \langle g_8, g_8 \rangle, \langle g_2, g_2 \rangle, \langle g_6', eg_1 \rangle\}$;
- $g_1 = \{ \langle |g\_root,g_4|, |g\_root,g_4| \rangle, \langle |g_4,g_5|, |g_4,g_5| \rangle, \langle |g_4,g_6|, |g_4,g_6| \rangle, \langle |g_6,g_7|, |g_6,g_7| \rangle, \langle |g_6,g_8|, |g_6,g_8| \rangle \}$;
- $h_1 = \{\langle |g_2,g_6'|, |g_2, eg_1| \rangle, \langle |g_7,g_8|, |g_7,g_8| \rangle\}$;

Homomorphism Γ_2: (mappings f_2 and h_2 for Γ_2 are shown in Figure 4.7)

$\Gamma_2 = \langle f_2, g_2, h_2 \rangle$ such that $f_2 : G_0 \rightarrow G_2$; $g_2: ET_0 \rightarrow ET_2$; $h_2: EG_0 \rightarrow EG_2$. We have:

- $f_2 = \{\langle g\_root, g\_root \rangle, \langle g_4, g_4 \rangle, \langle g_5, g_5 \rangle, \langle g_6, g_6 \rangle, \langle g_7, g_7 \rangle, \langle g_8, g_8 \rangle, \langle g_2, eg_2 \rangle, \langle g_6', g_6 \rangle\}$;
- $g_2 = \{ \langle |g\_root,g_4|, |g\_root,g_4| \rangle, \langle |g_4,g_5|, |g_4,g_5| \rangle, \langle |g_4,g_6|, |g_4,g_6| \rangle, \langle |g_6,g_7|, |g_6,g_7| \rangle, \langle |g_6,g_8|, |g_6,g_8| \rangle \}$;

- $h_2 = \{\langle |g_2,g_6'|, |eg_2, g_6|\rangle, \langle |g_7,g_8|, |g_7,g_8|\rangle\}$;

Homomorphism Γ_1': (mappings f_1' and h_1' for Γ_1' are shown in Figure 4.7)

$\Gamma_1' = \langle f_1', g_1', h_1'\rangle$ such that f_1': $G_1 \rightarrow G_3$, g_1': $ET_1 \rightarrow ET_3$, h_1': $EG_1 \rightarrow EG_3$. We have:

- $f_1' = \{\langle g\_root, g\_root\rangle, \langle g_1, g_1\rangle, \langle g_2, g_2\rangle, \langle g_3, g_3\rangle, \langle g_4, g_4\rangle, \langle g_5, g_5\rangle, \langle g_6, g_6\rangle, \langle g_7, g_7\rangle, \langle g_8, g_8\rangle, \langle eg_1, g_6\rangle\}$;
- $g_1' = \{\langle |g\_root, g_1|, |g\_root, g_1|\rangle, \langle |g_1, g_2|, |g_1, g_2|\rangle, \langle |g_1, g_3|, |g_1, g_3|\rangle, \langle |g\_root,g_4|, |g\_root,g_4|\rangle, \langle |g_4,g_5|, |g_4,g_5|\rangle, \langle |g_4,g_6|, |g_4,g_6|\rangle, \langle |g_6,g_7|, |g_6,g_7|\rangle, \langle |g_6,g_8|, |g_6,g_8|\rangle\}$;
- $h_1' = \{\langle |g_2, eg_1|, |g_2,g_6|\rangle, \langle |g_7,g_8|, |g_7,g_8|\rangle, \langle |g_3,g_6|, |g_3,g_6|\rangle\}$;

Homomorphism Γ_2': (mappings f_2' and h_2' for Γ_2' are shown in Figure 4.7)

$\Gamma_2' = \langle f_2', g_2', h_2'\rangle$ such that f_2': $G_2 \rightarrow G_3$, g_2': $ET_2 \rightarrow ET_3$, h_2': $EG_2 \rightarrow EG_3$. We have:

- $f_2' = \{\langle g\_root, g\_root\rangle, \langle g_4, g_4\rangle, \langle g_5, g_5\rangle, \langle g_6, g_6\rangle, \langle g_7, g_7\rangle, \langle g_8, g_8\rangle, \langle g_9, g_9\rangle, \langle eg_2, g_2\rangle\}$;
- $g_2' = \{\langle |g\_root,g_4|, |g\_root,g_4|\rangle, \langle |g_4,g_5|, |g_4,g_5|\rangle, \langle |g_4,g_6|, |g_4,g_6|\rangle, \langle |g_6,g_7|, |g_6,g_7|\rangle, \langle |g_6,g_8|, |g_6,g_8|\rangle, \langle |g_4, g_9|, |g_4, g_9|\rangle\}$;
- $h_2' = \{\langle |eg_2, g_6|, |g_2,g_6|\rangle, \langle |g_7,g_8|, |g_7,g_8|\rangle\}$;

GM_3 along with homomorphism Γ_1' and Γ_2' represent the pushout of GM_0 with homomorphism Γ_1 and Γ_2 . In fact, we have:

- $f_1' \circ f_1 = \{\langle g\_root, g\_root\rangle, \langle g_4, g_4\rangle, \langle g_5, g_5\rangle, \langle g_6, g_6\rangle, \langle g_7, g_7\rangle, \langle g_8, g_8\rangle, \langle g_2, g_2\rangle, \langle g_6', g_6\rangle\}$
- $f_2' \circ f_2 = \{\langle g\_root, g\_root\rangle, \langle g_4, g_4\rangle, \langle g_5, g_5\rangle, \langle g_6, g_6\rangle, \langle g_7, g_7\rangle, \langle g_8, g_8\rangle, \langle g_2, g_2\rangle, \langle g_6', g_6\rangle\}$
- $g_1' \circ g_1 = \{\langle |g\_root,g_4|, |g\_root,g_4|\rangle, \langle |g_4,g_5|, |g_4,g_5|\rangle, \langle |g_4,g_6|, |g_4,g_6|\rangle, \langle |g_6,g_7|, |g_6,g_7|\rangle, \langle |g_6,g_8|, |g_6,g_8|\rangle\}$;
- $g_2' \circ g_2 = \{\langle |g\_root,g_4|, |g\_root,g_4|\rangle, \langle |g_4,g_5|, |g_4,g_5|\rangle, \langle |g_4,g_6|, |g_4,g_6|\rangle, \langle |g_6,g_7|, |g_6,g_7|\rangle, \langle |g_6,g_8|, |g_6,g_8|\rangle\}$;
- $h_1' \circ h_1 = \{\langle |g_2,g_6'|, |g_2, g_6|\rangle, \langle |g_7,g_8|, |g_7,g_8|\rangle\}$;
- $h_2' \circ h_2 = \{\langle |g_2,g_6'|, |g_2, g_6|\rangle, \langle |g_7,g_8|, |g_7,g_8|\rangle\}$;

As $\Gamma_1' \circ \Gamma_1 = \langle f_1' \circ f_1, g_1' \circ g_1, h_1' \circ h_1\rangle$ and $\Gamma_2' \circ \Gamma_2 = \langle f_2' \circ f_2, g_2' \circ g_2, h_2' \circ h_2\rangle$, we have $\mathbf{\Gamma_1' \circ \Gamma_1 = \Gamma_2' \circ \Gamma_2}$. □

4.3 Category of Role Models

In this section, I define the category of Role Model. I then introduce the concepts that will allow the composition of role model via pushout.

Definition 4.9: Role models Homomorphism

Given two role models $RM_1 = \langle R_1, P_1, participant_1 \rangle$ and $RM_2 = \langle R_2, P_2, participant_2 \rangle$, a *role model homomorphism* from RM_1 to RM_2 is a function $\Delta = \langle i, j \rangle$ with $i : R_1 \rightarrow R_2$, $j : P_1 \rightarrow P_2$ such that:

- $\forall p \in P_1 |\ participant_1(p) = (r_1, r_2),\ j(p) = (i(r_1), i(r_2))$.

Proposition 4.10: Category of Role Models

Role models along with role model homomorphisms define the **category ROLE_MODEL**.

Proof:

Let us prove that role models along with role model homomorphisms form a category. According to Definition 3.5, we need to identify the objects, morphisms and identity morphisms and verify that the composition of morphism exists and is associative.

Objects**:** The objects are role models.

Morphisms**:** The morphisms are role model homomorphisms.

Identity**:** The identity morphism is a function $id_{RM} = \langle id_R, id_P \rangle$ such that id_R is an identity function that maps each role to itself, id_P is an identity function that maps each protocol to itself.

Composition**:** Let RM_1, RM_2, RM_3 be three role models and $\Delta_1 = \langle i_1, j_1 \rangle$, $\Delta_2 = \langle i_2, j_2 \rangle$ be two role model homomorphisms such that: Δ_1: $RM_1 \rightarrow RM_2$ and Δ_2: $RM_2 \rightarrow RM_3$. The composition is defined as follows: $\Delta_2 \circ \Delta_1 = \langle i_2 \circ i_1, j_2 \circ j_1 \rangle$.

Associativity**:** The role model homomorphism consists of functions between sets. Hence, the assiociativity property is derived from the corresponding property of functions between sets [72].

□

Definition 4.11: Configurations of Role Models

A *configuration of role models* specifies all the mappings that are used to merge two role models. Given two role models RM_1 and RM_2, a *configuration of role models* RM_1 and RM_2 is a triplet $\mathbf{config_{role} = \langle RM_0, \Delta_1, \Delta_2 \rangle}$ where:

- RM_0 is a role model
- Δ_1 corresponds to a role model homomorphism from RM_0 to RM_1
- Δ_2 corresponds to a role model homomorphism from RM_0 to RM_2

Definition 4.12: Role model composition

The *composition of two role models* RM_1, RM_2 over a role model configuration $\langle RM_0, \Delta_1, \Delta_2 \rangle$ is the role model resulting from the pushout of Δ_1 and Δ_2 in category ROLE_MODEL.

Example 4.5

The composition of role models RM_1, RM_2 over the configuration $\langle RM_0, \Delta_1, \Delta_2 \rangle$ is depicted in Figure 4.8. Role model RM_3 represents the composed model and it is obtained by pushout. The mappings for the functions comprised in the role model homomorphisms are shown in Figure 4.9.

The elements for the pushout of role models shown in Figure 4.9 are defined as follows:

Role Model RM_0:

$RM_0 = \langle R_0, P_0, participant_0 \rangle$,

$R_0 = \{ r_1, r_2, r_4, r_6, r_6' \}$

$P_0 = \{ p_1, p_2 \}$

$participant_0 = \{\langle p_1, \langle r_1, r_6' \rangle\rangle, \langle p_2, \langle r_2, r_6 \rangle\rangle\}$

Role Model RM_1:

$RM_1 = \langle R_1, P_1, participant_1 \rangle$,

$R_1 = \{ r_1, r_2, r_3, r_4, r_6, er_1, er_2 \}$

$P_1 = \{ p_1, p_2, p_3, p_4 \}$

$participant_1 = \{\langle p_1, \langle r_1, er_1 \rangle\rangle, \langle p_2, \langle r_2, r_6 \rangle\rangle, \langle p_3, \langle r_3, r_4 \rangle\rangle, \langle p_4, \langle r_3, er_2 \rangle\rangle\}$

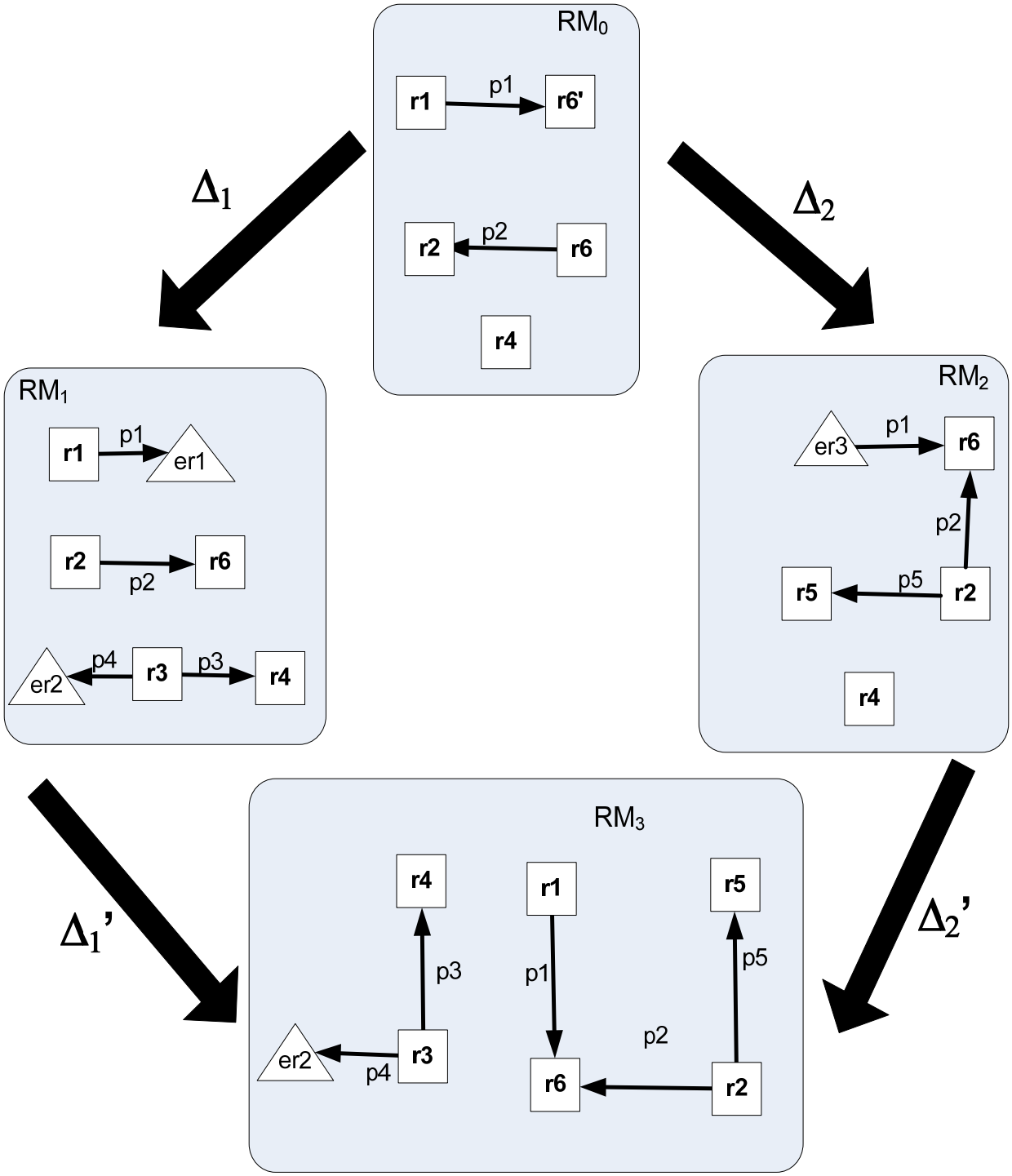

Figure 4.8. Overview of Pushout of Role Models

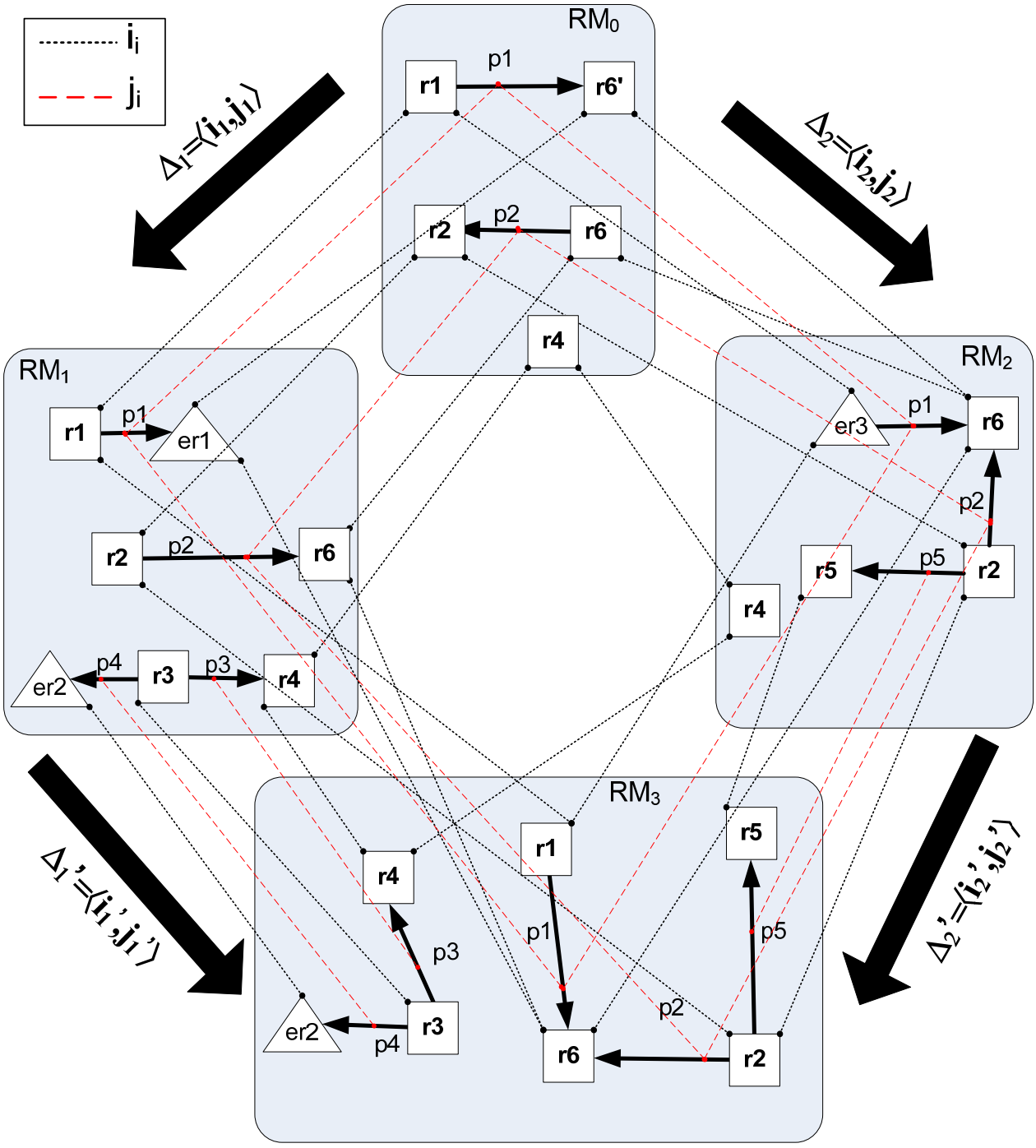

Figure 4.9. Pushout of Role Models with detailed functions. Functions mapping roles (i_i) and protocols (j_i) are shown.

Role Model RM_2:

$RM_2 = \langle R_2, P_2, participant_2 \rangle$,

$R_2 = \{ r_2, r_4, r_5, r_6, er_3\}$

$P_2 = \{ p_1, p_2, p_5\}$

$participant_1 = \{\langle p_1, \langle er_3, r_6\rangle\rangle, \langle p_2, \langle r_2, r_6\rangle\rangle, \langle p_5, \langle r_2, r_5\rangle\rangle\}$

Role Homomorphism Δ_1: (mappings i_1 and j_1 for Δ_1 are shown in Figure 4.9)

$\Delta_1 = \langle i_1, j_1\rangle$ such that $i_1: R_0 \rightarrow R_1$, $j_1: P_0 \rightarrow P_1$. We have:

- $i_1 = \{\langle r_1, r_1\rangle, \langle r_2, r_2\rangle, \langle r_4, r_4\rangle, \langle r_6, r_6\rangle, \langle r_6', er_1\rangle\}$;
- $j_1 = \{\langle p_1, p_1\rangle, \langle p_2, p_2\rangle\}$;

Role Homomorphism Δ_2: (mappings i_2 and j_2 for Δ_2 are shown in Figure 4.9)

$\Delta_2 = \langle i_2, j_2 \rangle$ such that $i_2: R_0 \rightarrow R_2$, $j_2: P_0 \rightarrow P_2$. We have:

- $i_2 = \{\langle r_1, er_3\rangle, \langle r_2, r_2\rangle, \langle r_4, r_4\rangle, \langle r_6, r_6\rangle, \langle r_6', r_6\rangle\}$;
- $j_2 = \{\langle p_1, p_1\rangle, \langle p_2, p_2\rangle\}$;

Role Homomorphism Δ_1': (mappings i_1' and j_1' for Δ_1' are shown in Figure 4.9)

$\Delta_1' = \langle i_1', j_1'\rangle$ such that $i_1': R_1 \rightarrow R_3$, $j_1': P_1 \rightarrow P_3$. We have:

- $i_1' = \{\langle r_1, r_1\rangle, \langle r_2, r_2\rangle, \langle r_3, r_3\rangle, \langle r_4, r_4\rangle, \langle r_6, r_6\rangle, \langle er_1, r_6\rangle, \langle er_2, er_2\rangle\}$;
- $j_1' = \{\langle p_1, p_1\rangle, \langle p_2, p_2\rangle, \langle p_3, p_3\rangle, \langle p_4, p_4\rangle\}$;

Role Homomorphism Δ_2': (mappings i_2' and j_2' for Δ_2' are shown in Figure 4.9)

$\Delta_2' = \langle i_2', j_2' \rangle$ such that $i_2': R_2 \rightarrow R_3$, $j_2': P_2 \rightarrow P_3$. We have:

- $i_2' = \{\langle r_2, r_2\rangle, \langle r_4, r_4\rangle, \langle r_5, r_5\rangle, \langle r_6, r_6\rangle, , \langle er_3, r_1\rangle\}$;
- $j_2' = \{\langle p_1, p_1\rangle, \langle p_2, p_2\rangle, \langle p_5, p_5\rangle\}$;

RM_3 along with homomorphism Δ_1' and Δ_2' represent the pushout of RM_0 with homomorphism Δ_1 and Δ_2. In fact, we have:

- $i_1' \circ i_1 = \{\langle r_1, r_1\rangle, \langle r_2, r_2\rangle, \langle r_4, r_4\rangle, \langle r_6, r_6\rangle, \langle r_6', r_6\rangle\}$
- $i_2' \circ i_2 = \{\langle r_1, r_1\rangle, \langle r_2, r_2\rangle, \langle r_4, r_4\rangle, \langle r_6, r_6\rangle, \langle r_6', r_6\rangle\}$

- $j_1' \circ j_1 = \{\langle p_1, p_1\rangle, \langle p_2, p_2\rangle\}$
- $j_2' \circ j_2 = \{\langle p_1, p_1\rangle, \langle p_2, p_2\rangle\}$

As $\Delta_1' \circ \Delta_1 = \langle i_1' \circ i_1, j_1' \circ j_1\rangle$ and $\Delta_2' \circ \Delta_2 = \langle i_2' \circ i_2, j_2' \circ j_2\rangle$, we have $\boldsymbol{\Delta_1' \circ \Delta_1 = \Delta_2' \circ \Delta_2}$. □

4.4 Category of Organization Models

In this section, I define the category of Organizations. I then introduce the concepts that allow the composition of organizations via pushout.

Definition 4.13: Organizations Homomorphism

Given two organizations $ORG_1 = \langle GM_1, RM1, achieves_1\rangle$, $ORG_2 = \langle GM_2, RM2, achieves_2\rangle$, an *organization homomorphism* $\Phi = \langle \Gamma, \Delta, k\rangle$ from ORG_1 to ORG_2 consists of:

- $\Gamma = \langle f, g, h\rangle$: goal model homomorphism from GM_1 to GM_2
- $\Delta = \langle i, j\rangle$: role model homomorphism from RM_1 to RM_2
- k : $achieves_1 \rightarrow achieves_2$, such that if $|r,g| \in achieves_1$, then $k(|r,g|) \in achieves_2$ and $k(|r,g|) = |\, i(r), f(g)|$

Proposition 4.14: Category of Organizations

Organizations along with organization homomorphisms define the **category ORG_MODEL**.

Proof:

Let us prove that organizations along with organization homomorphisms form a category. According to Definition 3.5, we need to identify the objects, morphisms and identity morphisms and verify that the composition of morphism exists and is associative.

Objects**:** The objects are organizations.

Morphisms**:** The morphisms are organization homomorphisms.

Identity**:** The identity morphism is a function $id_{ORG} = \langle id_{GM}, id_{RM}, id_k\rangle$ such that id_{GM} is an identity function that maps each goal model to itself (as defined in Section 4.2), id_{RM} is an identity function that maps each role model to itself (as defined in Section 4.3) and id_k is an identity function that maps each *achieves* edge to itself.

Composition**:** Let ORG_1, ORG_2, ORG_3 be three organizations and $\Phi_1 = \langle \Gamma_1, \Delta_1, k_1 \rangle$, $\Phi_2 = \langle \Gamma_2, \Delta_2, k_2 \rangle$ be two organization homomorphisms such that: $\Phi_1: ORG_1 \rightarrow ORG_2$ and $\Phi_2: ORG_2 \rightarrow ORG_3$. The composition is defined as follows:

$\Phi_2 \circ \Phi_1 = \langle \Gamma_2 \circ \Gamma_1, \Delta_2 \circ \Delta_1, k_2 \circ k_1 \rangle$.

Associativity**:** An organization homomorphism consists of functions between sets (Goal model homomorphisms and role model homomorphisms are set functions). Hence, the assiociativity property is derived from the corresponding property of functions between sets [72]. □

Definition 4.15: Configuration of Organizations

A *configuration of organizations* specifies all the mappings that are used to merge two organizations. Given two organizations ORG_1 and ORG_2, a *configuration of organizations* ORG_1 and ORG_2 is a triplet **config= $\langle ORG_0, \Phi_1, \Phi_2 \rangle$** where:

- ORG_0 is an organization
- Φ_1 corresponds to an organization homomorphism from ORG_0 to ORG_1
- Φ_2 corresponds to an organization homomorphism from ORG_0 to ORG_2

Definition 4.16: Composition of Organizations

The *composition* of two organizations ORG_1, ORG_2 over a configuration of organization config = $\langle ORG_0, \Phi_1, \Phi_2 \rangle$ is the organization resulting from the pushout of Φ_1 and Φ_2 in category ORG_MODEL.

Notation:

This composition is noted $\vdash (ORG_1, ORG_2, config) = ORG_1 \vdash^{config} ORG_2$.

The general intuition behind the pushout construction is that it aggregates the unrelated organization structures together without adding anything new and merges the shared structure defined in the configuration. It results in a composite organization that has all elements of both organizations while eliminating duplicates identified in the shared part. In fact, we are interested in composing two organizations that have some elements in common. Actually, composing two completely unrelated organizations is possible but uninteresting.

Example 4.6

Figure 4.10 shows an example of composition of organization ORG_1 and ORG_2 over the configuration $\langle ORG_0, \Phi_1, \Phi_2 \rangle$. This composition results in the pushout organization ORG_3 as depicted in Figure 4.10. The goal models and role models from the organizations shown here have been studied in Example 4.4 and Example 4.5. Therefore, we will not go into the details of the mapping for the goal models and roles models. I will just give the details for the *achieves* mappings.

Organization ORG_0:

GM_0: defined in Example 4.4.

RM_0: defined in Example 4.5.

$achieves_0 = \{|r_4, g_5|, |r_6, g_7|, |r_2, g_8|\}$

Organization ORG_1:

GM_1: defined in Example 4.4.

RM_1: defined in Example 4.5.

$achieves_1 = \{|r_1, g_2|, |r_3, g_3|, |r_4, g_5|, |r_6, g_7|, |r_2, g_8|\}$

Organization ORG_2:

GM_2: defined in Example 4.4.

RM_2: defined in Example 4.5.

$achieves_2 = \{|r_4, g_5|, |r_6, g_7|, |r_2, g_8|, |r_5, g_9|\}$

Organization Homomorphism Φ_1: (mappings k_1 is shown in Figure 4.9)

$\Phi_1 = \langle \Gamma_1, \Delta_1, k_1 \rangle$ where Γ_1 and Δ_1 have been defined in Example 4.4 and Example 4.5 and k_1: $achieves_0 \rightarrow achieves_1$. We have:

- $k_1 = \{\langle |r_4, g_5|, |r_4, g_5| \rangle, \langle |r_6, g_7|, |r_6, g_7| \rangle, \langle |r_2, g_8|, |r_2, g_8| \rangle\}$;

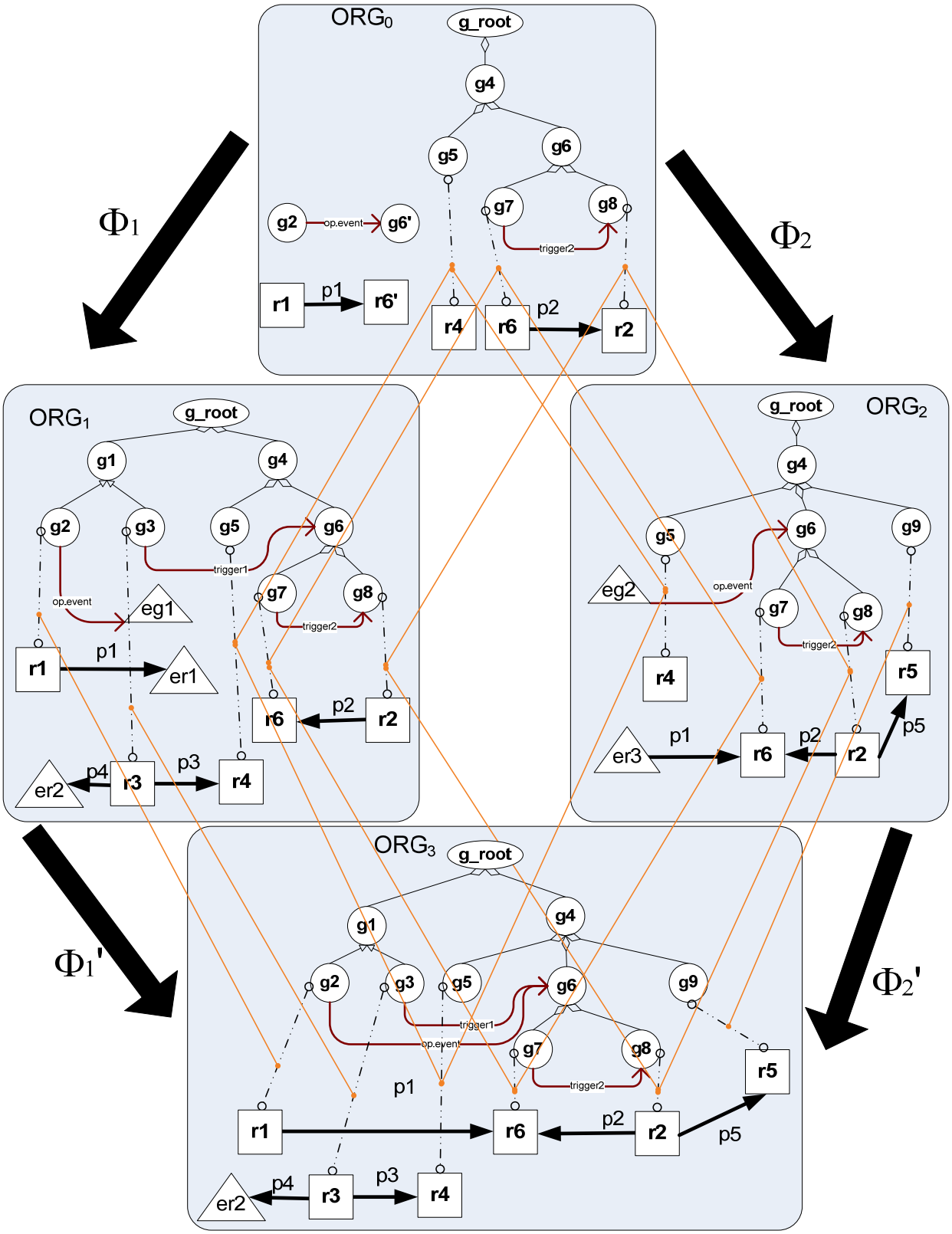

Figure 4.10. Pushout of Organizations. Only *achieves edges* mappings (k_i) are shown.

Organization Homomorphism Φ_2: (mappings of k_2 is shown in Figure 4.9)

$\Phi_2 = \langle \Gamma_2, \Delta_2, k_2 \rangle$ where Γ_2 and Δ_2 have been defined in Example 4.4 and Example 4.5 and k_2: achieves$_0 \rightarrow$ achieves$_2$. We have:

- $k_2 = \{\langle |r_4, g_5|, |r_4, g_5| \rangle, \langle |r_6, g_7|, |r_6, g_7| \rangle, \langle |r_2, g_8|, |r_2, g_8| \rangle\}$;

Organization Homomorphism Φ_1': (mappings k_1' is shown in Figure 4.9)

$\Phi_1' = \langle \Gamma_1', \Delta_1', k_1' \rangle$ where Γ_1' and Δ_1' have been defined in Example 4.4 and Example 4.5 and k_1': achieves$_1 \rightarrow$ achieves$_3$. We have:

$k_1' = \{\langle |r_1, g_2|, |r_1, g_2| \rangle, \langle |r_3, g_3|, |r_3, g_3| \rangle, \langle |r_4, g_5|, |r_4, g_5| \rangle, \langle |r_6, g_7|, |r_6, g_7| \rangle, \langle |r_2, g_8|, |r_2, g_8| \rangle\}$;

Organization Homomorphism Φ_2': (mappings k_2' is shown in Figure 4.9)

$\Phi_2' = \langle \Gamma_2', \Delta_2', k_2' \rangle$ where Γ_2' and Δ_2' have been defined in Example 4.4 and Example 4.5 and k_2': achieves$_2 \rightarrow$ achieves$_3$. We have:

$k_2' = \{\langle |r_5, g_9|, |r_5, g_9| \rangle, \langle |r_4, g_5|, |r_4, g_5| \rangle, \langle |r_6, g_7|, |r_6, g_7| \rangle, \langle |r_2, g_8|, |r_2, g_8| \rangle\}$;

ORG_3 along with homomorphism Φ_1' and Φ_2' represent the pushout of ORG_0 with homomorphism Φ_1 and Φ_2 . In fact, we have:

- $k_1' \circ k_1 = \{\langle |r_4, g_5|, |r_4, g_5| \rangle, \langle |r_6, g_7|, |r_6, g_7| \rangle, \langle |r_2, g_8|, |r_2, g_8| \rangle\}$;
- $k_2' \circ k_2 = \{\langle |r_4, g_5|, |r_4, g_5| \rangle, \langle |r_6, g_7|, |r_6, g_7| \rangle, \langle |r_2, g_8|, |r_2, g_8| \rangle\}$;

Hence, we have $k_1' \circ k_1 = k_2' \circ k_2$. Moreover, we have shown that $\Gamma_1' \circ \Gamma_1 = \Gamma_2' \circ \Gamma_2$ (Example 4.4) and $\Delta_1' \circ \Delta_1 = \Delta_2' \circ \Delta_2$ (Example 4.5). As $\Phi_1' \circ \Phi_1 = \langle \Gamma_1' \circ \Gamma_1, \Delta_1' \circ \Delta_1, k_1' \circ k_1 \rangle$ and $\Phi_2' \circ \Phi_2 = \langle \Gamma_2' \circ \Gamma_2, \Delta_2' \circ \Delta_2, k_2' \circ k_2 \rangle$, we have $\boldsymbol{\Phi_1' \circ \Phi_1 = \Phi_2' \circ \Phi_2}$. □

The pushout construction always results in a valid organization (as defined in Definition 4.4). This is mainly due to the facts that the pushout is constructed based on organization homomorphisms, which preserve the organization structure (Definition 4.13). However, depending on the configuration chosen, the composite organization can have a semantic that does not necessarily represent the semantic of the initial organizations.

Proposition 4.17: Correctness of the Composition of Organizations

Let ORG_1, ORG_2, ORG_3 be three organizations and config be a configuration of organizations. If ORG_1 and ORG_2 are valid organizations and $ORG_3 = ORG_1 \overset{config}{\vdash\!\!\!\vdash} ORG_2$ then ORG_3 is also a valid organization.

Proof:

An organization is valid if its structure corresponds to the one defined in Definition 4.4. As ORG_1 and ORG_2 are valid, proving that ORG_3 preserves the structure of both ORG_1 and ORG_2 will suffice to prove that ORG_3 is also valid. Hence, we only need to verify that ORG3 preserves the *parent*, *time-based*, *protocol* and *achieve* edges of ORG_1 and ORG_2. We prove this fact by contradiction.

The composed organization ORG_3 is obtained by pushout, which also results in the creation of two organization homomorphisms Φ_1: $ORG_1 \rightarrow ORG_3$ and Φ_2: $ORG_2 \rightarrow ORG_3$. We define $\Phi_1 = \langle f_1, g_1, h_1, i_1, j_1, k_1 \rangle$, $\Phi_2 = \langle f_2, g_2, h_2, i_2, j_2, k_2 \rangle$ where f_i, g_i, h_i, i_i, j_i, k_i $(1 \leq i \leq 2)$, are the functions defined in Definition 4.5, Definition 4.9 and Definition 4.13. Basically, f_i are functions mapping *goals*, g_i are functions mapping *parent* edges, h_i are functions mapping *time-based* edges, i_i are functions mapping *roles*, j_i are functions mapping *protocol* edges and finally k_i are functions mapping *achieves* edges.

Case 1: Assume that ORG_3 does not preserve the *parent* edges from ORG_1.

Let a_1, a_2, a_3, a_4, a_5 be goals. Hence, with this assumption, we have:

$\forall\ a_1,a_2: ORG_1\ ,\ \forall\ a_3,\ a_4: ORG_3,$

$(a_1 = parent(a_2) \wedge f_1(a_1) = a_3 \wedge f_1(a_2) = a_4\ ) \Rightarrow\ \exists a_5{:}ORG_3,\ a_5 \neq a_4 \mid g_1(|a_1,a_2|) = |a_3,\ a_5|$

$\Rightarrow g_1(|a_1,a_2|) \neq |f_1(a_1),\ f_1(a_2)|$

As Φ_1 is an organization homomormisphm from ORG_1 to ORG_3, we have:

$g_1(|a_1,a_2|) = |a_3, a_4| = |f_1(a_1), f_1(a_2)|$ (by Definition 4.5 and Definition 4.13).

However, the assumption led to $g_1(|a_1,a_2|) \neq |f_1(a_1), f_1(a_2)|$. Therefore, there is a contradiction and ORG_3 preserves the *parent* edges from ORG_1.

Case 2: Assume that ORG_3 does not preserve the *parent* edges from ORG_2.

Hence, with this assumption, we have:

$\forall\ a_1,a_2$: ORG_2 , $\forall\ a_3, a_4$: ORG_3,

$(a_1=\text{parent}(a_2) \wedge f_2(a_1)= a_3 \wedge f_2(a_2)= a_4) \Rightarrow\ \exists a_5{:}ORG_3,\ a_5 \neq a_4 \mid g_2(|a_1,a_2|) = |a_3, a_5|$

$\Rightarrow g_2(|a_1,a_2|) \neq |f_2(a_1), f_2(a_2)|$

As Φ_2 is an organization homomormisphm from ORG_2 to ORG_3, we have:

$g_2(|a_1,a_2|) = |a_3, a_4| = |f_2(a_1), f_2(a_2)|$ (by Definition 4.5 and Definition 4.13).

However, the assumption led to $g_2(|a_1,a_2|) \neq |f_2(a_1), f_2(a_2)|$. Therefore, there is a contradiction and ORG_3 preserves the *parent* edges from ORG_2.

Case 3: Assume that ORG_3 does not preserve the *time-based* edges from ORG_1.

Let a_1, a_2, a_3, a_4, a_5 be goals. Hence, with this assumption, we have:

$\forall\ a_1,a_2$: ORG_1 , $\forall\ a_3, a_4$: ORG_3,

($|a_1,a_2|$ *time-based* edge $\wedge\ f_1(a_1)= a_3 \wedge f_1(a_2)= a_4) \Rightarrow\ \exists a_5{:}ORG_3,\ a_5 \neq a_4 \mid h_1(|a_1,a_2|) = |a_3, a_5|$

$\Rightarrow h_1(|a_1,a_2|) \neq |f_1(a_1), f_1(a_2)|$

As Φ_1 is an organization homomormisphm from ORG_1 to ORG_3, we have:

$h_1(|a_1,a_2|) = |a_3, a_4| = |f_1(a_1), f_1(a_2)|$ (by Definition 4.5 and Definition 4.13).

However, the assumption led to $h_1(|a_1,a_2|) \neq |f_1(a_1), f_1(a_2)|$. Therefore, there is a contradiction and ORG_3 preserves the *time-based* edges from ORG_1.

Case 4: Assume that ORG_3 does not preserve the *time-based* edges from ORG_2.

Let a_1, a_2, a_3, a_4, a_5 be goals. Hence, with this assumption, we have:

$\forall\ a_1,a_2$: ORG_2 , $\forall\ a_3, a_4$: ORG_3,

($|a_1,a_2|$ *time-based* edge $\wedge\ f_2(a_1)= a_3 \wedge f_2(a_2)= a_4) \Rightarrow\ \exists a_5{:}ORG_3,\ a_5 \neq a_4 \mid h_2(|a_1,a_2|) = |a_3, a_5|$

$\Rightarrow h_2(|a_1,a_2|) \neq |f_2(a_1), f_2(a_2)|$

As Φ_2 is an organization homomormisphm from ORG_2 to ORG_3, we have:

$h_2(|a_1,a_2|) = |a_3, a_4| = |f_2(a_1), f_2(a_2)|$ (by Definition 4.5 and Definition 4.13).

However, the assumption led to $h_2(|a_1,a_2|) \neq |f_2(a_1), f_2(a_2)|$. Therefore, there is a contradiction and ORG_3 preserves the *time-based* edges from ORG_2.

Case 5: Assume that ORG_3 does not preserve the *protocol* edges from ORG_1.

Let a_1, a_2, a_3, a_4, a_5 be roles. Hence, with this assumption, we have:

$\forall\ a_1,a_2$: ORG_1 , $\forall\ a_3, a_4$: ORG_3,

$(|a_1,a_2|$ *protocol* edge $\wedge\ i_1(a_1)= a_3 \wedge i_1(a_2)= a_4\ ) \Rightarrow\ \exists a_5$:$ORG_3$, $a_5 \neq a_4 \mid\ j_1(|a_1,a_2|) = |a_3, a_5|$

$\Rightarrow j_1(|a_1,a_2|) \neq |i_1(a_1), i_1(a_2)|$

As Φ_1 is an organization homomormisphm from ORG_1 to ORG_3, we have:

$j_1(|a_1,a_2|) = |a_3, a_4| = |i_1(a_1), i_1(a_2)|$ (by Definition 4.9 and Definition 4.13).

However, the assumption led to $j_1(|a_1,a_2|) \neq |i_1(a_1), i_1(a_2)|$. Therefore, there is a contradiction and ORG_3 preserves the *protocol* edges from ORG_1.

Case 6: Assume that ORG_3 does not preserve the *protocol* edges from ORG_2.

Let a_1, a_2, a_3, a_4, a_5 be goals. Hence, with this assumption, we have:

$\forall\ a_1,a_2$: ORG_2 , $\forall\ a_3, a_4$: ORG_3,

$(|a_1,a_2|$ *protocol* edge $\wedge\ i_2(a_1)= a_3 \wedge i_2(a_2)= a_4\ ) \Rightarrow\ \exists a_5$:$ORG_3$, $a_5 \neq a_4 \mid j_2(|a_1,a_2|) = |a_3, a_5|$

$\Rightarrow j_2(|a_1,a_2|) \neq |i_2(a_1), i_2(a_2)|$

As Φ_2 is an organization homomormisphm from ORG_2 to ORG_3, we have:

$j_2(|a_1,a_2|) = |a_3, a_4| = |i_2(a_1), i_2(a_2)|$ (by Definition 4.9 and Definition 4.13).

However, the assumption led to $j_2(|a_1,a_2|) \neq |i_2(a_1), i_2(a_2)|$. Therefore, there is a contradiction and ORG_3 preserves the *protocol* edges from ORG_2.

Case 7: Assume that ORG_3 does not preserve the *achieve* edges from ORG_1.

Let a_1, a_3, be goals and a_2, a_4, a_5 be roles. Hence, with this assumption, we have:

$\forall\ a_1,a_2$: ORG_1 , $\forall\ a_3, a_4$: ORG_3,

$(|a_1,a_2|$ *achieve* edge $\wedge\ f_1(a_1)= a_3 \wedge i_1(a_2)= a_4\ ) \Rightarrow\ \exists a_5$:$ORG_3$, $a_5 \neq a_4 \mid\ k_1(|a_1,a_2|) = |a_3, a_5|$

$\Rightarrow k_1(|a_1,a_2|) \neq |f_1(a_1), i_1(a_2)|$

As Φ_1 is an organization homomormisphm from ORG_1 to ORG_3, we have:

$k_1(|a_1,a_2|) = |a_3, a_4| = |f_1(a_1), i_1(a_2)|$ (by Definition 4.13).

However, the assumption led to $k_1(|a_1,a_2|) \neq |f_1(a_1), i_1(a_2)|$. Therefore, there is a contradiction and ORG_3 preserves the *achieve* edges from ORG_1.

Case 8: Assume that ORG_3 does not preserve the *achieve* edges from ORG_2.

Let a_1, a_3, be goals and a_2, a_4, a_5 be roles. Hence, with this assumption, we have:

$\forall\ a_1,a_2: ORG_2$, $\forall\ a_3, a_4: ORG_3$,

$(|a_1,a_2|$ *achieve* edge $\wedge\ f_2(a_1)= a_3 \wedge i_2(a_2)= a_4\ ) \Rightarrow\ \exists a_5{:}ORG_3, a_5 \neq a_4 \mid k_2(|a_1,a_2|) = |a_3, a_5|$

$\Rightarrow k_2(|a_1,a_2|) \neq |f_2(a_1), i_2(a_2)|$

As Φ_2 is an organization homomormisphm from ORG_2 to ORG_3, we have:

$k_2(|a_1,a_2|) = |a_3, a_4| = |f_2(a_1), i_2(a_2)|$ (by Definition 4.13).

However, the assumption led to $k_2(|a_1,a_2|) \neq |f_2(a_1), i_2(a_2)|$. Therefore, there is a contradiction and ORG_3 preserves the *achieve* edges from ORG_2.

Therefore, ORG3 preserves the *parent*, *time-based*, *protocol* and *achieve* edges from ORG_1 and ORG_2. Hence, ORG_3 is valid.

□

4.5 Related Work

The problem of composing models has been studied in various domains [15] and many approaches have proposed the use of the notion of colimit in category theory as a formalism to compose various types of models. For instance, some works have been done to compose UML models [9, 52], requirement models [97, 102], statechart models [85], database schemas [16, 95], ontologies [17, 54] and programs [86].

In the multiagent systems community, there are very few works unifying category theory and multiagent systems. Most of those types of research are done at the implementation level. For instance, Johnson et al. [68] use category theory to formalize the composition multiagent dialogue protocols while Soboll [108] proposes to model multiagent cooperation patterns as

categories. However, none of those approaches explicitly considers organizational designs. In this chapter, I proposed a formal approach to compose a set of interrelated models that compose a multiagent organization design.

4.6 Summary

The main contribution of this chapter is providing an abstract mechanism for merging OMAS designs. I have shown that the composition of multiagent organizations can be formulated using the pushout notion in category theory. I defined three main categories, GOAL_MODEL, ROLE_MODEL and ORG_MODEL, as the category of goal models, role models and organization models respectively. Then, I have defined the notion of *organization homomorphisms* and specified the composition of organization as the pushout object of organization homomorphisms. Nevertheless, finding suitable organization homomorphisms is not an easy task. Moreover, arbitrary homomorphisms could potentially lead to semantically incorrect composite organizations that cannot be implemented into a coherent system. In the next chapter, I provide a specific approach that guides designers to decide what organizations to compose. Moreover, this approach guarantees the construction of correct homomorphisms that can be used in the composition by pushout.

CHAPTER 5 - A SERVICE-ORIENTED FRAMEWORK FOR DESIGNING MULTIAGENT ORGANIZATIONS

"Out of intense complexities intense simplicities emerge." — Winston Churchill

""What is the use of a book", thought Alice, "without pictures or conversations?" "
—*Lewis Carroll, Alice in Wonderland*

In the previous chapter, I have shown how to compose two arbitrary multiagent organizations in the context of the OMACS model. This composition is general and allows the organizations to be composed in several ways, resulting in an organization that might not behave in a predictable way. In this chapter, I provide a specific approach that will guide designers to decide when to compose and what organizations to compose. I propose employing reusable multiagent organizations designed using both component-oriented and service-oriented principles. These two well-established software engineering approaches allow us to decompose large multiagent organizations into smaller organizations that can be developed separately and composed later, thus allowing designers to design large and complex OMAS more easily. This framework uses the composition process defined in the previous chapter and proposes some guidelines that allow designers to know what to expect when composing organizations. In my approach, I view services as basic elements used to develop large multiagent organizations. They represent independent organizations encapsulating common functionalities and they can be *composed* with other organizations to obtain larger systems. Hence, in my approach, organizations are viewed as reusable components that use and provide services. In general, services represent cooperative tasks that cannot be achieved by a single agent but rather require the cooperation of several agents in order to provide the service.

In this chapter, I define and formalize all the entities required to develop those so-called *reusable organizations* and compose them into a single composite organization. Each organization exposes generic interfaces called *connection points*. Connection points allow organizations to *provide* and/or *use* services. They can then be interconnected in a suitable way such that the required services match the provided services. This composition is made through *connectors* that link connection points together. The composition process ensures the consistency of all the bindings and results in a valid composite organization.

5.1 Running Example

As the running example for this chapter, I consider two simple organizations that need to be composed. These organizations are presented in Figure 5.1 and Figure 5.2. In theses figures, we represent *internal goals* as ovals, *internal roles* as rectangles, *external goals and roles* as triangles, *precedes* and *triggers* relations as open-head arrows, *protocols* as full-head arrows, and *achieves* relations as dash lines. *Conjunctive goals* are connected to their subgoals by diamond-shaped links and *disjunctive goals* by triangle-shaped links. Roles are decorated by the «requires» stereotype that indicates the agent capabilities that would be required to play that role.

The first is a *Search Organization* (Figure 5.1), in which a team of heterogeneous robots searches for victims at a disaster area. The main goal is *Search*. Note that the top-level goal named *Root* is an empty goal added to simplify the composition, as indicated in Section 4.1. The *Search* goal has two conjunctive subgoals: *Divide Area* and *Explore*, which is parameterized with the subarea to explore. The *Divide Area* goal divides the area into smaller subareas that can be assigned to individual robots. Once the main area has been divided, a number of *Explore* goals are triggered. For each leaf goal, there is role designed to achieve it as shown.

The second organization is the *Rescue Organization* (Figure 5.2), in which a team of robots coordinate to rescue victims at given locations. The main goal, *Rescue*, is decomposed into two conjunctive goals: *ID Victim*, whose objective is to confirm the presence of a victim, and *Pickup Victim*, which aims at bringing the victim to safety.

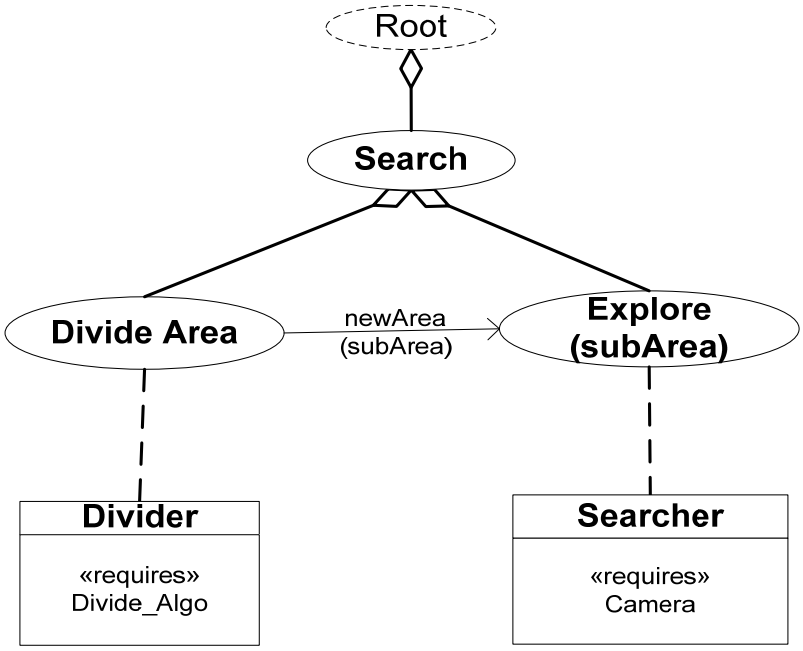

Figure 5.1. Search Organization

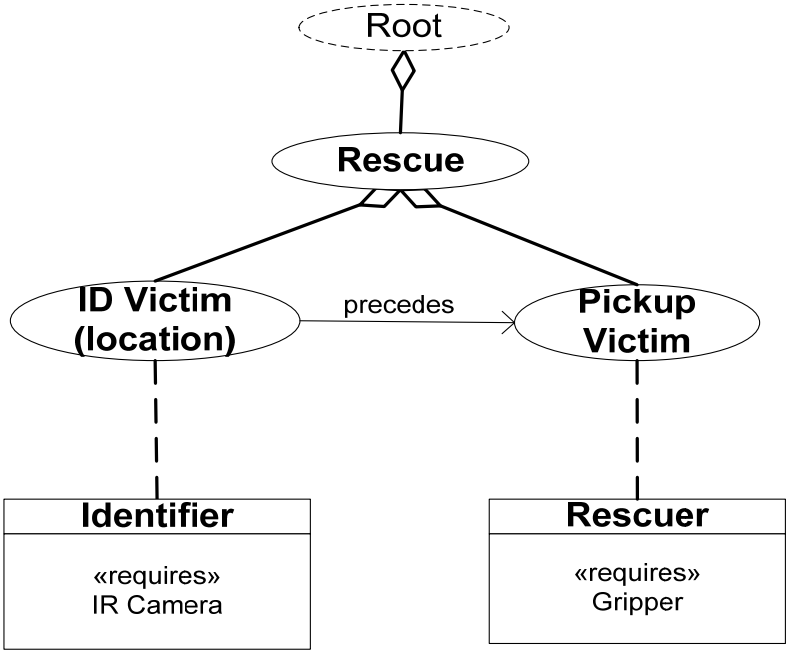

Figure 5.2. Rescue Organization

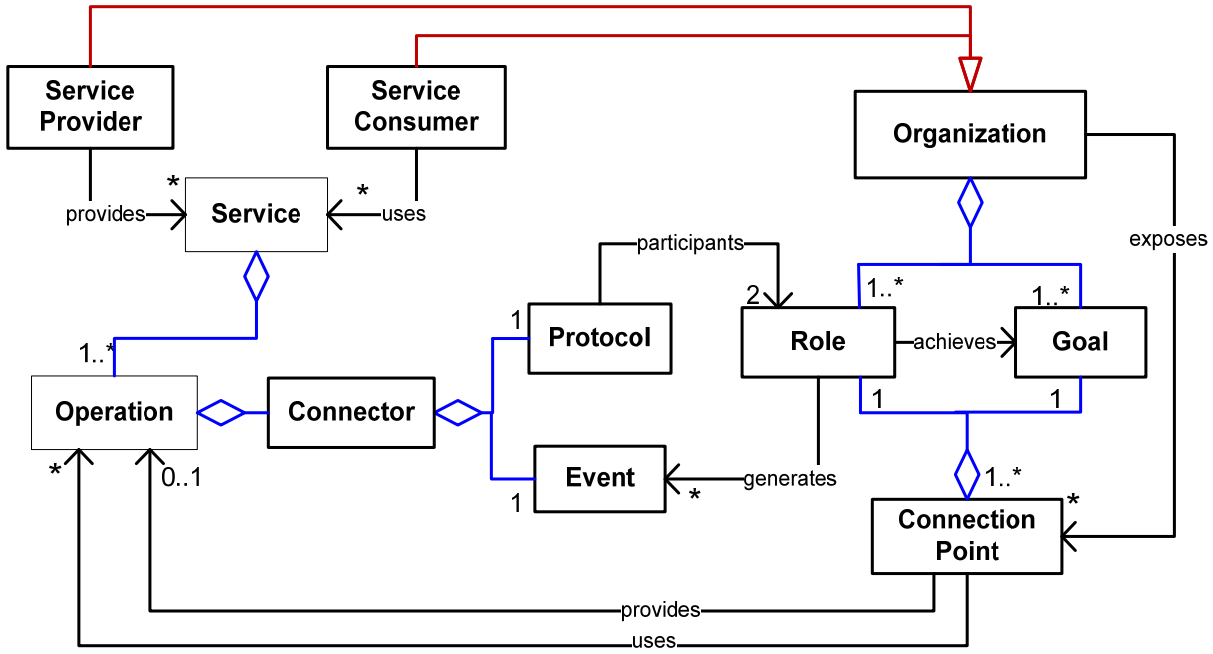

Figure 5.3. Organizational Service Metamodel

5.2 Service Model

In my framework, I view services as multiagent organizations encapsulating functionalities commonly used in MAS. Once services are designed, they can be used by other organizations in order to build larger systems. In order to include our service-oriented approach into the design of OMAS, I incorporate some of the core service-oriented concepts into a generalized OMACS metamodel. Figure 5.3 presents the organizational service metamodel, comprising of the service and organizational entities and their relationships.

The central concept is that of a *Service*. Services offer one or more *operations*. Each operation possesses a *connector* that is used to connect *connection points* exposed by organizations. *Connection points* are pairs of *goals* and *roles* that can be connected by *events* and *protocols* from *connectors*. *Service providers* and *service consumers* are both autonomous organizations who respectively provide and use operations from services. These entities are discussed in detail in the following subsections.

5.2.1 Services

A Service is a logical entity that represents a coarse-grained multiagent functionality. This coarse-grained functionality contains a set of fine-grained functionalities, called *operations*, which can be used during the design of multiagent organizations. Each service has an XML-based specification that contains a specification of each operation provided. We denote by SVC the set of all services.

Example 5.1

The XML excerpt in Figure 5.4 shows the specification of the rescuing service whose main operation is to rescue a victim at a given area. It shows the service name and the operation name along with other parameters. Theses parameters are discussed in the next sections.

To be purposeful, a service must be implemented by at least one provider. Services facilitate reuse in the sense that they allow consumers to request operations solely based on the service specification, without knowing anything about the implementation proposed by providers.

```
<service name= Rescuing>
  <operation name= rescue>
    <connector>
            <protocol>rescue_protocol</protocol>
            <event>rescue_event(location)</event>
    </connector>
    <conditions>
      <pre> The location is accessible </pre>
      <post> The victim has been rescued </post>
    </conditions>
  </operation>
</service>
```

Figure 5.4. Rescuing Service specification

5.2.2 Operations

An *operation* can be viewed as a set of goals that an organization has to achieve in order to guarantee the postcondition of that operation. Operations can result in some computations being made (e.g. computing an optimal path for a swarm of UAVs) or some actions being performed (e.g. neutralizing an enemy). The actual goals that need to be achieved for an operation depend on specific organizations. Each operation consists of a set of *preconditions* and *postconditions,* and a *connector.*

Definition 5.1: Operation

An *operation* is a tuple op = ⟨ connector, precondition, postcondition ⟩ where:

- connector: interface used to interconnect goals and roles
- precondition: A condition prior the execution of the operation
- postconditions: A condition after the execution of the operation

We denote by OP the set of all operations.

Preconditions represent constraints that have to be respected by any consumers in order for the operation to be executed properly. *Postconditions* indicate what to expect upon completion of that operation. Finally, connectors provide the "glue" allowing consumers and providers to be connected based on an operation.

Definition 5.2: Connector

A *connector* is a tuple con = ⟨ event, protocol ⟩ where:

- event: event used to connect goals
- protocol: protocol used to interconnect roles

Interaction protocols specify how the interaction happens between consumers and providers. R*equest events* are events that trigger the instantiation of the operations. An interaction protocol and a request event form a *connector*. Hence, a connector defines the event that needs to occur in order for the operation to start and provides a way for providers and consumers to exchange information regarding the execution of the operation.

Example 5.2

The *rescue operation* specified in Example 5.1 requires that the victim's location be accessible and guarantees that the victim, if any, is rescued upon successful termination. The operation starts based on the event *rescue_event*, which provides the location of the victim found. In addition, information regarding the state of the victim, the urgency of the rescue and the outcome of the rescue can be exchanged via the protocol *rescue_protocol*.

5.2.3 Connection points

In my framework, I model provided and required operations of organizations through the notion of a *connection point*. A connection point of an organization is a logical construct that is associated with a particular operation and concretely represented by a goal-role pair from that organization.

Definition 5.3: Connection Point

A *connection point* of an organization O is a tuple cp = ⟨ goal, role ⟩ where:

- goal is a goal from O
- role is a role from O.

We denote by CP the set of all connection points.

However, not all connection points are valid. There are two types of *valid connection points*: entry and exit connection points. An *entry connection point* of an organization guarantees a proper instantiation of the operations it provides. Its goal and role components are called the *entry goal* and *entry role* respectively.

Definition 5.4: Entry Connection Points of an Organization

The set EntryCP of entry connection points of an organization O is defined as follows:

$$\mathbf{O.EntryCP} = \{cp{:}CP \mid \exists g_L \in G_L, \langle cp.role, g_L \rangle \in achieves \wedge cp.goal \in parent^*(g_L) \quad (1)$$

$$\wedge \forall g'' \in parent^*(g_L), g''.type \neq OR)\} \quad (2)$$

where $parent^*: G_i \rightarrow 2^{Gi}$ is the reflexive transitive closure of parent relationship in the induced tree. It returns all the ancestors of a goal, including the goal itself.

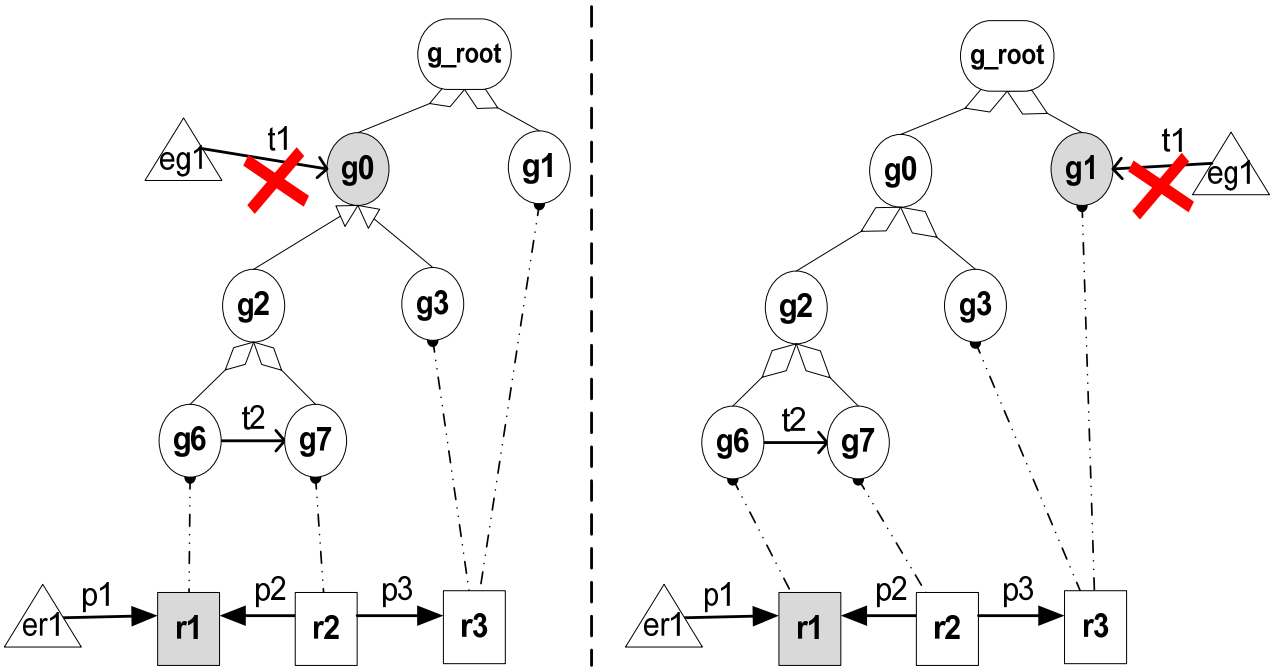

Figure 5.5. Invalid Entry Connection Points

Essentially, to be valid, an entry connection point needs to guarantee that once its entry goal is triggered, its entry role can be assigned to an agent. For that, given a connection point from an organization, we require that the leaf goal achieved by its entry role (called g_L) be a descendant of its entry goal (1) and that no alternate path should exist from the root to g_L (2). These two conditions guarantee that the entry role can be assigned to an agent (if available) once the entry goal is triggered.

Example 5.3

Figure 5.5 shows two cases of invalid entry connection points. In these figures, the connection points are represented by the gray nodes. In fact, in the organization on the left, $\langle g_0, r_1 \rangle$ is a connection point. In addition, g_6 is achieved by the entry role r_1. However, g_0, an ancestor of g_6, is an OR-goal, which violates condition (2) of Definition 5.4. In this scenario, when g_0 is triggered, the organization can choose to pursue g_3 and r_1 will never get assigned to an agent, preventing the operation from executing properly.

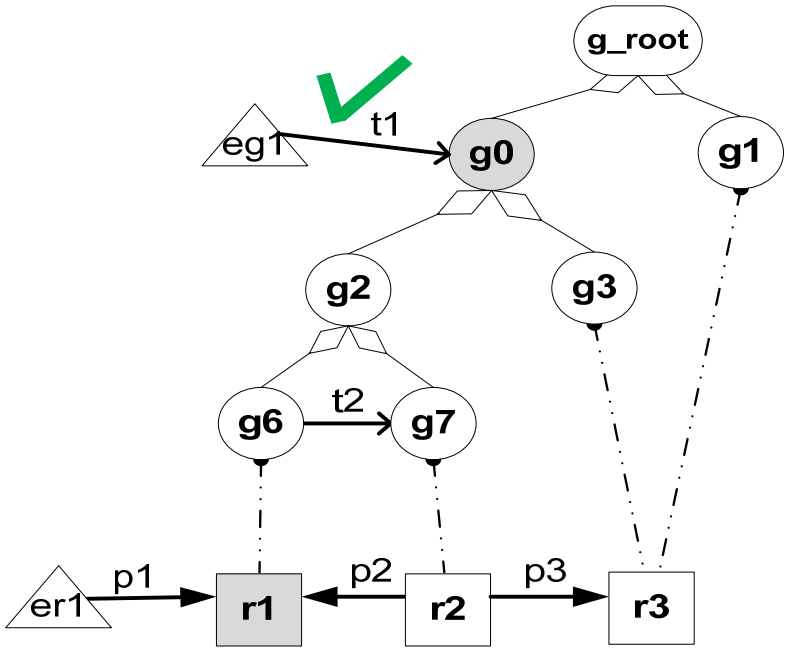

Figure 5.6. Valid Entry Connection Points

Similarly, in the organization on the right, $\langle g_1, r_1 \rangle$ is the connection point. Goal g_6 is achieved by the entry role r_1. However, g_6 is not a descendant of g_1, which violates condition (1) of Definition 5.4. In this scenario, r_1 can get assigned without g_1 being triggered.

Figure 5.6 shows an example of a good connection point. In this organization, $\langle g_0, r_1 \rangle$ is a connection point and both conditions of Definition 5.4 are met.

An exit connection point of an organization guarantees a proper request of the operation it uses. Its goal and role components are called the *exit goal* and *exit role* respectively. Given a connection point from an organization, its exit goal should be a leaf goal that is achieved by its exit role.

Definition 5.5: Exit Connection Points of an Organization

The set ExitCP of exit connection points of an organization O is defined as follows:

O.ExitCP = $\{cp:CP \mid cp.goal \in G_L \wedge \langle cp.role, cp.goal \rangle \in achieves\}$

Basically, in an exit connection point, the exit role needs to achieve the exit goal.

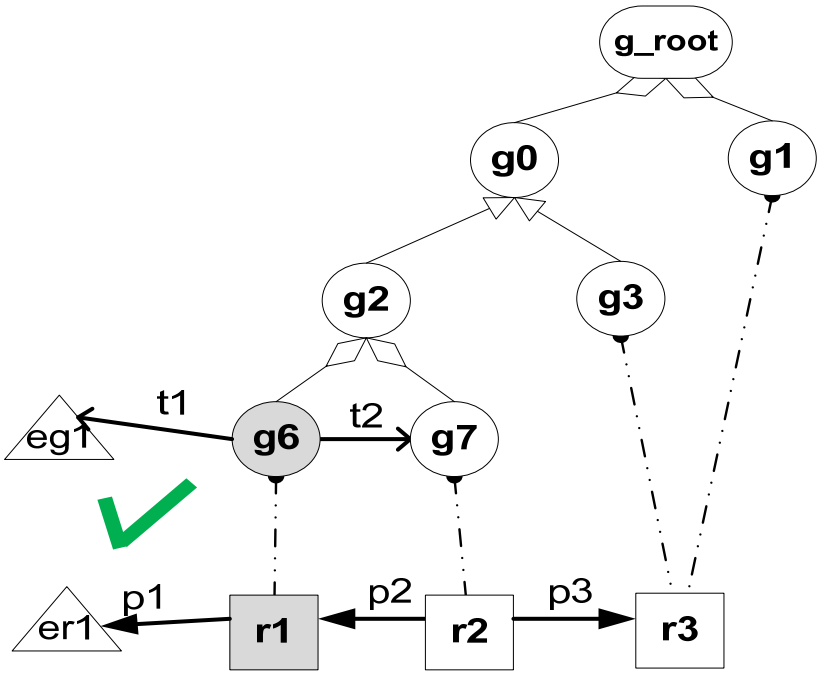

Figure 5.7. Exit Connection Points

Example 5.4

Figure 5.7 shows a valid exit connection point. Any other goal, role pair can be an exit connection point as long as the exit role needs to achieve the exit goal.

Connection points are the key interfaces that allow organizations to provide and use operations.

Definition 5.6: Operations provided by a connection point

Given an organization O and a connection point cp, the function *provides:* $CP \times ORG \rightarrow 2^{OP}$ defines the set of operations provided by cp. It is defined as:

provides(cp) = { op:OP | $\exists\, gx_2 \in G_x \mid (gx_2, \text{cp.goal}) \in \text{triggers(op.event)}$ (1)

$\wedge\, \exists\, rx_2 \in R_x \mid (rx_2, \text{cp.role}) \in \text{participants(op.protocol)}\ \}$ (2)

The roles and goals mentioned in Definition 5.6 are part of the same organization as the connection point. We say that an entry connection point *provides an operation* if its *entry goal* is

instantiated based on the occurrence of the *request event* of that operation (1) and its entry role engages with an external role in the *interaction protocol* defined for that operation (2).

Definition 5.7: Operations used by a connection point

Given an organization O, a connection point cp, the function *uses: CP* $\times$ *ORG* $\rightarrow 2^{OP}$ defines the set of operations used by cp as:

uses(cp) = { op:OP | $\exists$ $gx_1 \in G_x$ | (cp.goal, gx_1) $\in$ triggers(op.event) (1)

$\wedge \exists$ $rx_1 \in R_x$ | (cp.role, rx_1) $\in$ participants(op.protocol) } (2)

In Definition 5.7, we say that an exit connection point *requires an operation* if its *exit goal* generates a trigger based on the *request event* of that operation (1) and if its *exit role* engages with an external role in the *interaction protocol* defined for that operation (2).

5.2.4 Service Providers

A *service provider* is an autonomous organization that *provides all the operations* of a particular service. We say that an organization provides a service if, for all operations of the service, it exposes a *unique entry connection point* providing that operation.

Definition 5.8: Services providers

The function *provides: ORG*$\rightarrow 2^{SVC}$ defines the set of services provided by an organization. Given an organization O, we have:

provides(O) = {s:SVC | ($\forall$op$\in$ s.operations, $\exists$ cp$\in$O.EntryCP | op $\in$ provides(cp))}

We say that organization O is a service provider for service s if s $\in$ provides(O).

In addition, a service provider needs to be designed such that whenever the preconditions of an operation are met, it pursues a subset of its goals whose achievement leads to a state satisfying the postconditions of that operation. In this framework, providers can also require operations from other services.

5.2.5 Service Consumer

A *service consumer* is an autonomous organization that *uses one or more operations* from various services. To use an operation, an organization needs to expose at least one *exit connection point* using that operation.

Definition 5.9: Services consumers

The function uses: ORG $\rightarrow 2^{SVC}$ defines the set of services used by an organization. Given an organization O, we have:

uses(O) = { s:SVC | ($\exists$ op $\in$ s.operations, $\exists$ cp$\in$ O.ExitCP | op $\in$ uses(cp))}

We say that organization O is a service consumer for service s if s $\in$ uses(O)).

For each operation used, a service consumer can expose multiple connection points. Designers of service consumers choose the operations they need based on the service specification, which describes what each operation is supposed to do. Designers are responsible to ensure that the service consumers respect the preconditions of the operations they use.

Example 5.5

In this example, we modify the organizations presented in Figure 5.1 and Figure 5.2 such that the *Rescue Organization* provides the *Rescuing* service specified in Example 5.1 (see Figure 5.8) and the *Search Organization* uses the *rescue* operation of that service (see Figure 5.9). For both organizations, we need to select valid connections points and connect them to the *rescue_event* and *rescue_protocol* declared in the operation's connector. The *Searcher* role from the *Search Organization* can use the *Rescuing Service* to achieve its goal. Thus, the *Search Organization* contains the exit connection point ⟨*Explore*, *Searcher*⟩ such that the exit goal *Explore* triggers the external goal gx_1 based on *rescue_event* while the exit role *Searcher* participates in protocol *rescue_protocol* with external role rx_1. Similarly, the *Rescue Organization* provides the *rescue* operation. Hence, we identify the entry connector point ⟨*ID Victim*, *Identifier*⟩ with its entry goal and role connected to external entities gx_2 and rx_2 via the *rescue* operation connector. We indicate the services used by an exit role via the «uses» keyword and the operation provided by an entry role by the «provides» keyword.

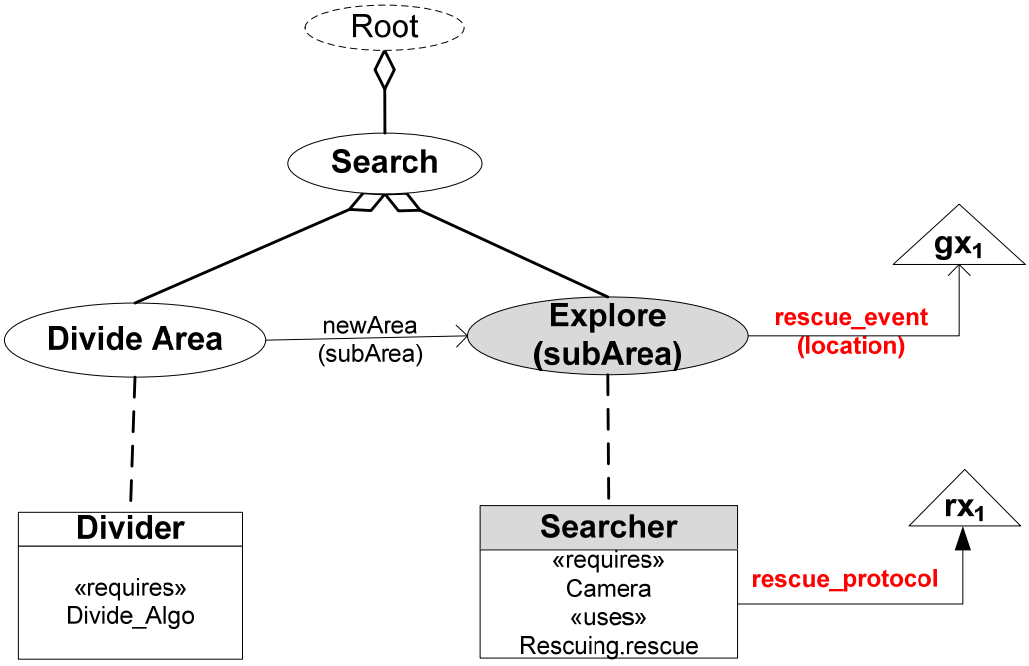

Figure 5.8. Search Organization - Consumer

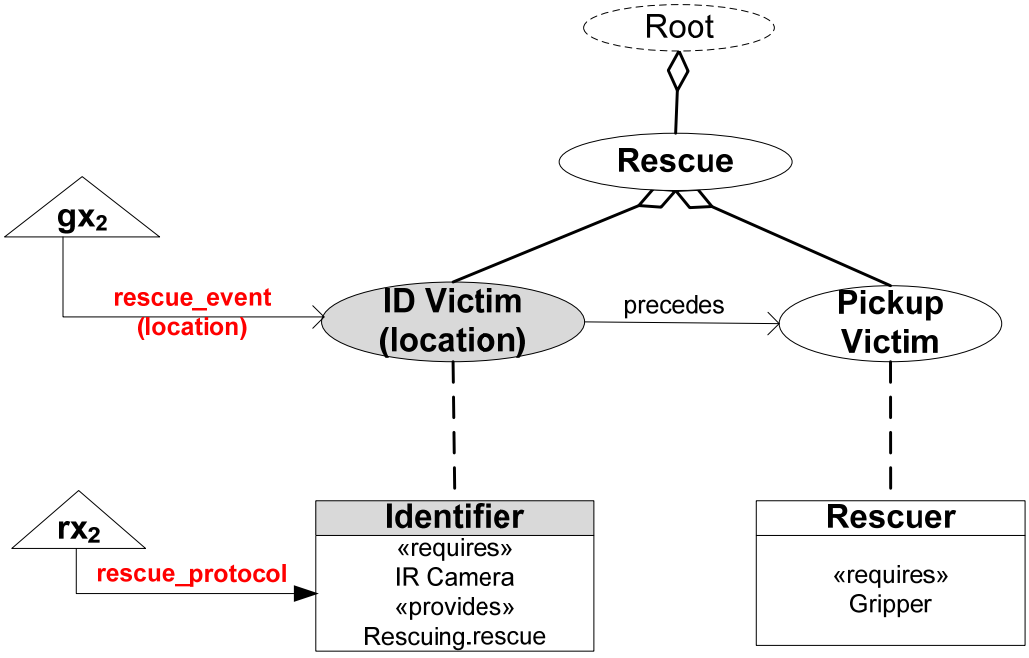

Figure 5.9. Rescue Organization - Provider

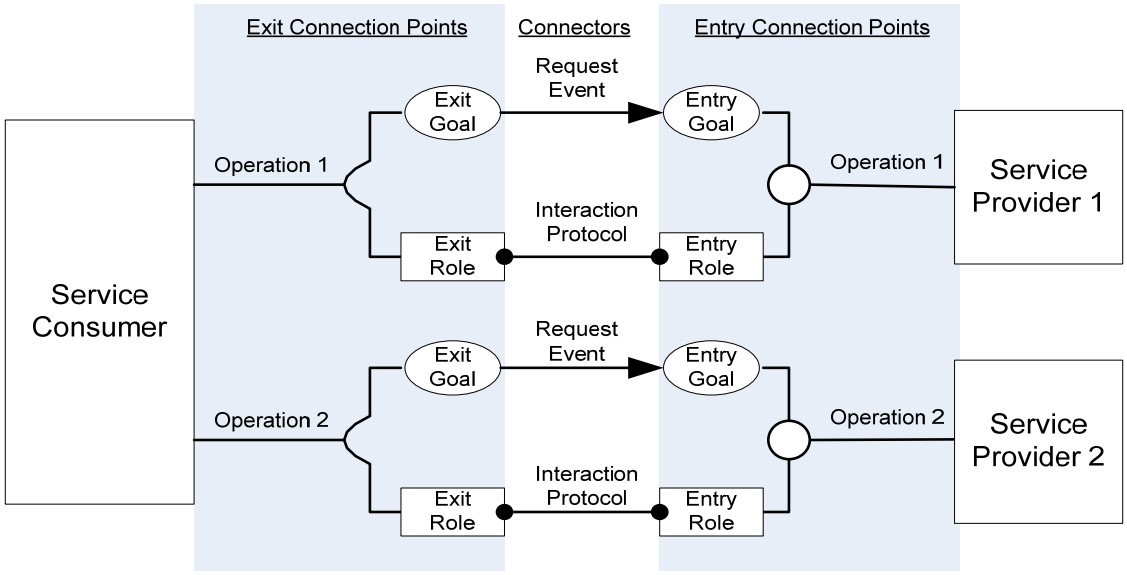

Figure 5.10. Interconnection of organization connection points using connectors.

5.3 Composition of services

So far, we have introduced the entities essential for the design of organization-based multiagent services. We now present our composition process, which is a mechanism through which organizations can be assembled in order to produce larger ones.

Composition is a design-time process that binds a consumer application with a provider in order to create a single composite organization. The composition process is illustrated in Figure 5.10. Given an operation, the composition process connects the exit connection point of a consumer with the entry connection point of a provider using the operation's connector. This interconnection ensures that the *exit goal* from the consumer organization can trigger (via the *request event*) the initialization of the *entry goal* from the provider, thus triggering the execution of the operation. Once the operation is initialized, *exit roles* and *entry roles* can interact via the *interaction protocol*. The composition parameters are captured through a *configuration*.

Definition 5.10: Configuration of connection points

A *configuration of connection points* is a tuple cfg = ⟨ exit, op, entry ⟩ where:

- op is an operation

- exit is an exit connection point using op
- entry is an entry connection point providing op

We denote by CFG the set all configurations.

Finally, I can now formalize the result of the composition of two organizations over a configuration. This composition is the pushout object of two organization homomorphism that can be constructed based on the relationships between the two initial organizations.

Proposition 5.11: Composition of organizations over a configuration

Given two organizations O_1 and O_2 and a configuration of connection points cfg = ⟨ exit, op, entry⟩ such that exit ∈ O_1.ExitCP and entry ∈ O_2.EntryCP, the composition of O_1 and O_2 over cfg is their composition over a configuration of organizations config = $\langle O_0, \Phi_1, \Phi_2 \rangle$. This configuration of organizations is constructed based on the configuration of connection points.

Notation:

The composition of two organizations over a configuration is the function ⊢ : ORG × ORG × CFG → ORG. Given two organizations O_1 and O_2 and a configuration cfg = ⟨ exit, op, entry ⟩, this composition is denoted $\vdash(O_1, O_2, cfg) = O_1 \vdash^{cfg} O_2$.

Proof:

The configuration of organizations O_1 and O_2 has been defined in Definition 4.15 as a triplet $\langle O_0, \Phi_1, \Phi_2 \rangle$ where:

- O_0 is an organization (the shared organization)
- Φ_1 corresponds to an organization homomorphism from O_0 to O_1
- Φ_2 corresponds to an organization homomorphism from O_0 to O_2

I first describe the construction of O_0, Φ_1 and Φ_2. Then I prove that the functions Φ_1 and Φ_2 obtained from this construction are organization homomorphisms.

Construction of organization homomorphism

Given two organizations O_1 and O_2 , let *cfg* = ⟨*exit, op, entry*⟩ where :

- op is an operation with connector ⟨op.event, op.protocol⟩,
- exit = ⟨exit.goal, exit.role⟩,
- entry = ⟨ entry.goal, entry.role ⟩.

From Definition 5.6 and Definition 5.7, exit.goal is required to trigger an external goal gx_1 based on op.event, and exit.role and rx_1 should participate in op.protocol. Similarly, entry.goal should be triggered by goal gx_2 and entry.role and rx_2 should participate in op.protocol. To construct organization O_0 and organization homomorphisms Φ_1: $O_0 \rightarrow O_1$ and Φ_2: $O_0 \rightarrow O_2$, we use the following steps:

Step 1: Equivalence between external entities

Using the connection points and the external entities, we establish equivalences by associating connection points from one organization to external entities of the other and vice versa. Therefore, for any given configuration, we always have:

- exit.goal ≡ gx_2, exit.role ≡ rx_2 and entry.goal ≡ gx_1, entry.role≡ rx_1

Step 2: Equivalence between nodes

We include equivalences between all "identical" nodes (goals and roles). Without loss of generality, we can assume that two roles are identical if they have the same name. Therefore, we assume a unique namespace for all organization roles. Moreover, we define two goals as identical if they are both empty or if they have the same name and type and their parents and children are also identical. Note that as all goal models have the same root, we always to include the equivalence about the roots. More formally, we have:

- if g_1 is a goal in O_1 and g_2 is a goal in O_2 then

  $(g_1 = g_2 = g\_root)$

  $\vee$ (g_1.name = g_2.name $\wedge$ g_1.type = g_2.type $\wedge$ parent(g_1) ≡ parent(g_2)) $\Rightarrow$ $g_1 \equiv g_2$
- if r_1 is a role in O_1 and r_2 is a role in O_2 then

  r_1.name = r_2.name $\Rightarrow$ $r_1 \equiv r_2$

Step 3: Equivalence between edges

We include equivalences between all "identical" edges. Given an edge $|a_1,b_1|$ from O_1 and an edge $|a_2,b_2|$ from O_2, if , in step 1 or 2 we had $a_1 \equiv a_2$ and $b_1 \equiv b_2$, then we set $|a_1,b_1| \equiv |a_2,b_2|$. More formally, we have:

- if $|a_1,b_1|$ is an edge in O_1 and $|a_2,b_2|$ is an edge in O_2 then
  $(a_1 \equiv b_1 \ \wedge a_2 \equiv b_2) \Rightarrow |a_1,b_1| \equiv |a_2,b_2|$

Step 4: Creation configuration of organizations

For each equivalence of nodes $a_1 \equiv a_2$ from step 1 and 2 such that a_1 is in O_1 and a_2 is in O_2, we create a node a_0 of the same type in O_0 and we establish $N_1(a_0) = a_1$ and $N_2(a_0) = a_2$, where N_i is the appropriate node mapping function of Φ_i. For instance, if a_1 and a_2 are goals, then a_0 is also a goal and $N_1(a_0)$ refers to the goal mapping function of Φ_1. Similarly, for each equivalence of edges $|a_1,b_1| \equiv |a_2,b_2|$ from step 3 such that $|a_1b_1|$ is in O_1 and $|a_2,b_2|$ is in O_2, we create an edge $|a_0,b_0|$ in O_0, such that $N_1(a_0) = a_1$, $N_1(b_0) = b_1$, $N_2(a_0) = a_2$ and $N_2(b_0) = b_2$. In addition, we establish $E_1(|a_0,b_0|) = |a_1,b_1|$ and $E_2(|a_0,b_0|) = |a_2,b_2|$, where E_i is the appropriate edge mapping function of Φ_i. For instance, if $|a_1,b_1|$ and $|a_2,b_2|$ are *achieves* edges, then $|a_0,b_0|$ is also an *achieves* edge and $F_1(|a_0,b_0|)$ refers to the *achieves* edge mapping function of Φ_1.

Formally, we have:

- For *goals* a_1 and a_2, $a_1 \equiv a_2 \Rightarrow \exists a_0 \in O_0 \mid f_1(a_0) = a_1 \wedge f_2(a_0) = a_2$,
- For *roles* a_1 and a_2, $a_1 \equiv a_2 \Rightarrow \exists a_0 \in O_0 \mid i_1(a_0) = a_1 \wedge i_2(a_0) = a_2$,
- For *tree edges* $|a_1,b_1|$, $|a_2,b_2|$, $|a_1,b_1| \equiv |a_2,b_2| \Rightarrow \exists\ |a_0,b_0| \in O_0$, $g_1(|a_0,b_0|) = |a_1,b_1|$
  $\wedge\ g_2(|a_0,b_0|) = |a_2,b_2| \wedge f_1(a_0) = a_1 \wedge f_2(a_0) = a_2 \wedge f_1(b_0) = b_1 \wedge f_2(b_0) = b_2$,
- For *graph edges* $|a_1,b_1|$, $|a_2,b_2|$, $|a_1,b_1| \equiv |a_2,b_2| \Rightarrow \exists\ |a_0,b_0| \in O_0$, $h_1(|a_0,b_0|) = |a_1,b_1|$
  $\wedge\ h_2(|a_0,b_0|) = |a_2,b_2| \wedge f_1(a_0) = a_1 \wedge f_2(a_0) = a_2 \wedge f_1(b_0) = b_1 \wedge f_2(b_0) = b_2$,
- For *protocols* $|a_1,b_1|$ and $|a_2,b_2|$, $|a_1,b_1| \equiv |a_2,b_2| \Rightarrow \exists\ |a_0,b_0| \in O_0$, $j_1(|a_0,b_0|) = |a_1,b_1|$
  $\wedge\ j_2(|a_0,b_0|) = |a_2,b_2| \wedge i_1(a_0) = a_1 \wedge i_2(a_0) = a_2 \wedge i_1(b_0) = b_1 \wedge i_2(b_0) = b_2$,
- For *achieves* edges $|a_1,b_1|$ and $|a_2,b_2|$, $|a_1,b_1| \equiv |a_2,b_2| \Rightarrow \exists\ |a_0,b_0| \in O_0$, $k_1(|a_0,b_0|) =$
  $= |a_1,b_1| \wedge k_2(|a_0,b_0|) = |a_2,b_2| \wedge i_1(a_0) = a_1 \wedge i_2(a_0) = a_2 \wedge f_1(b_0) = b_1 \wedge f_2(b_0) = b_2$.

Proof that the construction leads to organization homomorphisms

Let $\Phi_1 = \langle f_1, g_1, h_1, i_1, j_1, k_1 \rangle$, $\Phi_2 = \langle f_2, g_2, h_2, i_2, j_2, k_2 \rangle$ where f_i, g_i, h_i, i_i, j_i, k_i $(1 \leq i \leq 2)$, are the functions defined in Definition 4.5, Definition 4.9 and Definition 4.13. Basically, f_i are functions mapping *goals*, g_i are functions mapping edges from the induced tree of the goal models (*parent edges*), h_i are functions mapping edges from the induced graph of the goal

models (*time-based edges*), i_i are functions mapping *roles*, j_i are functions mapping *protocol edges* and finally k_i are functions mapping *achieves edges*. First, let us show that f_i, g_i, h_i are functions preserving the structure of the goal model.

Let $root_0$, $root_1$, $root_2$ be the root of the goal model in O_0, O_1 and O_2 respectively. Let us show that the root is preserved by the functions f_1 and f_2. Therefore, we need to **show that $f_1(root_0) = root_1$ and $f_2(root_0) = root_2$**, where $root_0$, $root_1$, $root_2$ are the roots in O_0, O_1, and O_2 respectively. From step 2, we have :

$root_1 \equiv root_2$.

From step 4, we create a goal g_0 such that:

$f_1(g_0) = root_1$ and $f_2(g_0) = root_2$.

By setting g_0 as the root of O_0 (i.e. $g_0 = root_0$), we have:

$f_1(root_0) = root_1$ and $f_2(root_0) = root_2$.

Next**,** let $|a_0,b_0|$ be a tree edge in O_0. Let us show that tree edges are preserved by the functions g_1 and g_2. Hence, we need to **show that $g_1(|a_0,b_0|) = |f_1(a_0),f_1(b_0)|$ and $g_2(|a_0,b_0|) = |f_2(a_0),f_2(b_0)|$**. We have:

$|a_0,b_0|$ tree edge in $O_0 \Rightarrow \exists\ |a_1,b_1| \in O_1$ and $\exists\ |a_2,b_2| \in O_2$ such that $|a_1,b_1| \equiv |a_2,b_2|$ (from step 3)

$\Rightarrow f_1(a_0) = a_1 \wedge f_1(b_0) = b_1 \wedge f_2(a_0) = a_2 \wedge f_2(b_0) = b_2 \wedge g_1(|a_0,b_0|) = |a_1,b_1|$

$\wedge\ g_2(|a_0,b_0|) = |a_1,b_1|$ (from step 4)

$\Rightarrow g_1(|a_0,b_0|) = |f_1(a_0),f_1(b_0)| \wedge g_2(|a_0,b_0|) = |f_2(a_0),f_2(b_0)|$

Finally, let $|a_0,b_0|$ be a induced graph edge in O_0. Let us show that graph edges are preserved by the functions g_1 and g_2. For that, we need to **show that $h_1(|a_0,b_0|) = |f_1(a_0),f_1(b_0)|$ and $h_2(|a_0,b_0|) = |f_2(a_0),f_2(b_0)|$**. We have:

$|a_0,b_0|$ graph edge in $O_0 \Rightarrow \exists\ |a_1,b_1| \in O_1$ and $\exists\ |a_2,b_2| \in O_2$ such that $|a_1,b_1| \equiv |a_2,b_2|$ (from step 3)

$\Rightarrow f_1(a_0) = a_1 \wedge f_1(b_0) = b_1 \wedge f_2(a_0) = a_2 \wedge f_2(b_0) = b_2 \wedge h_1(|a_0,b_0|) = |a_1,b_1|$

$\wedge\ h_2(|a_0,b_0|) = |a_1,b_1|$ (from step 4)

$\Rightarrow h_1(|a_0,b_0|) = |f_1(a_0),f_1(b_0)| \wedge h_2(|a_0,b_0|) = |f_2(a_0),f_2(b_0)|$

Next, let us show that i_i, j_i are functions preserving the structure of the role model. Let $|a_0,b_0|$ be a protocol in O_0. Let us show that protocols are preserved by the function j_1. Hence, we need to **show that $j_1(|a_0,b_0|) = |i_1(a_0),i_1(b_0)|$ and $j_2(|a_0,b_0|) = |i_2(a_0),i_2(b_0)|$**. We have:

$|a_0,b_0|$ protocol in $O_0 \Rightarrow \exists\ |a_1,b_1| \in O_1$ and $\exists\ |a_2,b_2| \in O_2$ such that $|a_1,b_1| \equiv |a_2,b_2|$ (from step 3)

$\Rightarrow i_1(a_0) = a_1 \wedge i_1(b_0) = b_1 \wedge i_2(a_0) = a_2 \wedge i_2(b_0) = b_2 \wedge j_1(|a_0,b_0|) = |a_1,b_1|$

$\wedge\ j_2(|a_0,b_0|) = |a_1,b_1|$ (from step 4)

$\Rightarrow j_1(|a_0,b_0|) = |i_1(a_0),i_1(b_0)| \wedge j_2(|a_0,b_0|) = |i_2(a_0),i_2(b_0)|$

Finally, let's show that the function k_i preserves the *achieves* relationship. For that, we need to show that given an *achieves* edge $|a_0,b_0|$, we have **$k_1(|a_0,b_0|) = |i_1(a_0),f_1(b_0)|$ and $k_2(|a_0,b_0|) = |i_2(a_0),f_2(b_0)|$**. We have:

$|a_0,b_0|$ achieves edge in $O_0 \Rightarrow \exists\ |a_1,b_1| \in O_1$ and $\exists\ |a_2,b_2| \in O_2$ such that $|a_1,b_1| \equiv |a_2,b_2|$ (step 3)

$\Rightarrow i_1(a_0) = a_1 \wedge f_1(b_0) = b_1 \wedge i_2(a_0) = a_2 \wedge f_2(b_0) = b_2 \wedge k_1(|a_0,b_0|) = |a_1,b_1|$

$\wedge\ k_2(|a_0,b_0|) = |a_1,b_1|$ (from step 4)

$\Rightarrow k_1(|a_0,b_0|) = |i_1(a_0),f_1(b_0)| \wedge k_2(|a_0,b_0|) = |i_2(a_0),f_2(b_0)|$

□

Example 5.6

To demonstrate how this construction works in practice, I use our Search and Rescue organizations whose designs are shown in Figure 5.8 and Figure 5.9. We design the organization called S&R as the composition of Search and Rescue over cfg where cfg= { ⟨Explore, Searcher⟩, rescue, ⟨ID Victim, Identifier⟩}. As described in Example 5.5, *⟨Explore, Searcher⟩* is an exit connection point in the Search Organization and *⟨ID Victim, Identifier⟩* an entry connection point in the Rescue Organization. The rescue operation is specified in Example 5.2. Hence, we have:

S&R = Search $\vdash\!\!\!\!-^{cfg}$ Rescue.

Let us follow the construction steps proposed in Proposition 5.11 to build organization org_0 and organization homomorphisms Φ_1, Φ_2 that are used in the pushout operation leading to organization S&R. Figure 5.11 shows org_0, Φ_1, Φ_2 derived from this construction. We have:

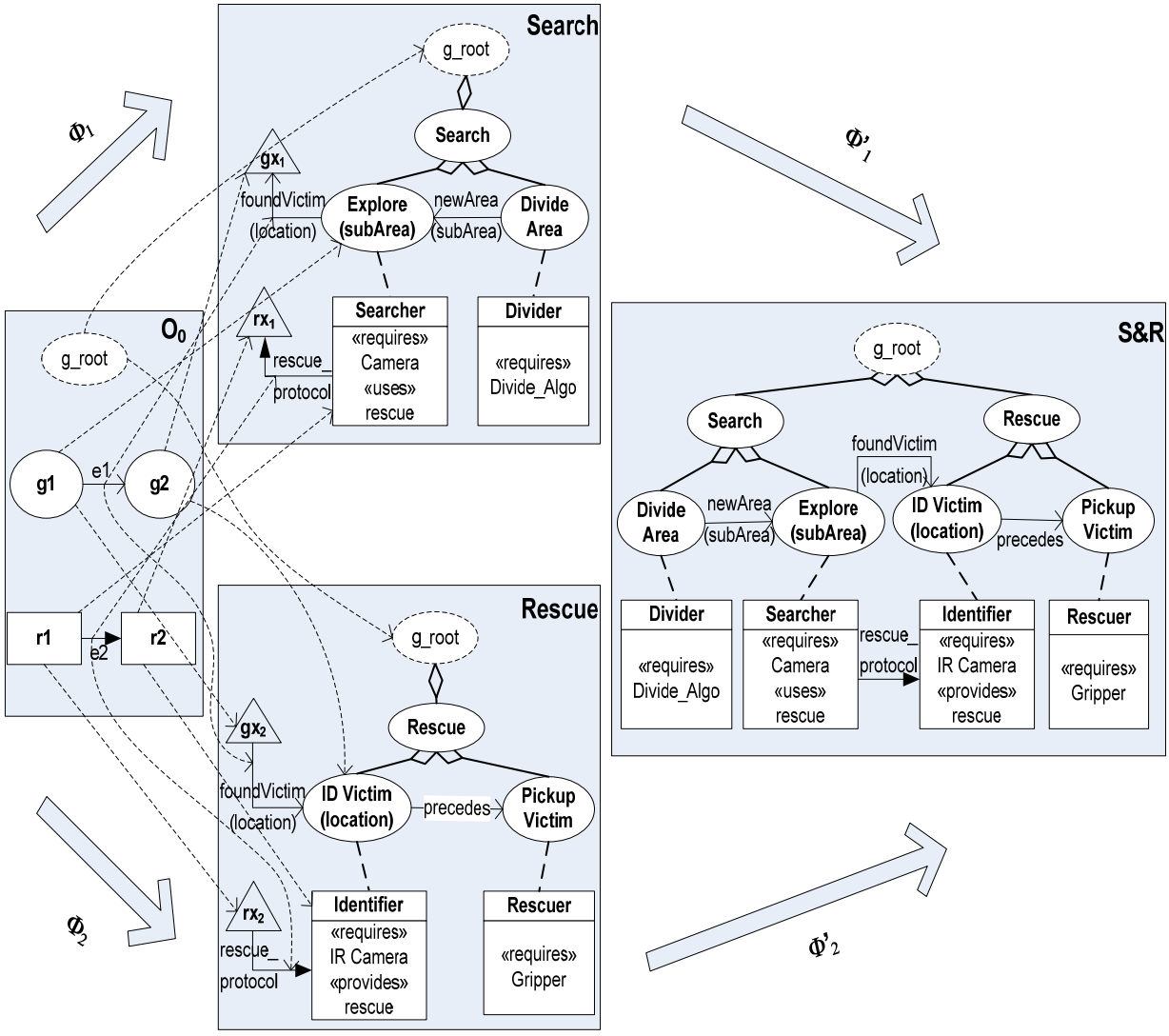

Figure 5.11. Composition of Search and Rescue

Step 1:

Explore ≡ gx_2

gx_1 ≡ ID Victim

Searcher ≡ rx_2,

rx_1 ≡ Identifier.

Step 2:

g_root ≡ g_root,

Step 3:

| Explore, gx_1 | ≡ | gx_2, ID Victim |

| Explorer, rx_1| ≡ | rx_2, Identifier |

Step 4:

From the equivalences created in steps 1,2 and 3, we create organization O_0 with goals g_root, g_1, g_2, roles r_1, r_2, and edges e_1, e_2. Moreover, we create organization homomorphisms $\Phi_1 = \langle f_1, g_1, h_1, i_1, j_1, k_1 \rangle$ and $\Phi_2 = \langle f_2, g_2, h_2, i_2, j_2, k_2 \rangle$ such that Φ_1: $O_0 \rightarrow$ Search, Φ_2: $O_0 \rightarrow$ Rescue. We have:

Explore ≡ gx_2 $\Rightarrow$ $f_1(g_1)$ = Explore $\wedge$ $f_2(g_1) = gx_2$;

gx_1 ≡ ID Victim $\Rightarrow$ $f_1(g_2) = gx_1 \wedge f_2(g_2)$= ID Victim;

Searcher ≡ rx_2 $\Rightarrow$ $i_1(r_1)$ = Searcher $\wedge$ $i_2(r_1) = rx_2$;

rx_1 ≡ Identifier $\Rightarrow$ $i_1(r_2) = rx_1 \wedge i_2(r_2)$ = Identifier;

g_root ≡ g_root $\Rightarrow$ f_1(g_root) = g_root and f_2(g_root) = g_root ;

|Explore, gx_1| ≡ |gx_2, ID Victim| $\Rightarrow$ $h_1(|g_1,g_2|) = |$Explore, $gx_1|$

$\wedge$ $h_1(|g_1,g_2|) = |gx_2$, ID Victim| $\wedge$ $f_1(g_1)$ = Explore $\wedge$ $f_2(g_1) = gx_2$

$\wedge$ $f_1(g_2) = gx_1 \wedge f_2(g_2)$= ID Victim

|Explorer, rx_1| ≡ |rx_2, Identifier| $\Rightarrow$ $j_1(|r_1,r_2|) = |$Explorer, $rx_1|$

$\wedge$ $j_1(|g_1,g_2|) = |rx_2$, Identifier| $\wedge$ $i_1(r_1)$ = Searcher $\wedge$ $i_2(r_1) = rx_2$

$\wedge$ $i_1(r_2) = rx_1 \wedge i_2(r_2)$ = Identifier

□

Finally, we can compute the composed organization *S&R = Search* $\vdash\!\!\!\!-^{cfg}$ *Rescue* as the pushout object of Φ_1 and Φ_2 in the category ORG. This organization is also shown on Figure 5.11. For clarity of the diagram, we do not show the mappings for Φ'_1 and Φ'_2 leading to S&R. The details of this composition are discussed in Appendix A.1.

The composition of two compatible organizations results in a single organization in which, depending on the assignment process, agents can end up playing roles in both the consumer and provider organizations. When designing an application, we need to iterate the composition process over adequate providers as long as the resulting composite organization still requires some operations. This iteration is possible as unused connection points are still available after composition. Essentially, after the composition, all exit and entry connection points that are not part of the configuration belong to the composite organization.

Proposition 5.12: Connection points of a composite organization

Let O_1, O_2, O_3 be three organizations and cfg = ⟨exit, op, entry⟩ be a configuration of connection points such that exit ∈ O_1.ExitCP and entry ∈ O_2.EntryCP.

If $O_3 = O_1 \vdash\!\!\!\!-^{cfg} O_2$, then :

O_3.ExitCP = O_1.ExitCP ∪ O_2.ExitCP – {exit}

O_3.EntryCP = O_1.EntryCP ∪ O_2.EntryCP – {entry}

Proof:

As stated in Section 3.3, the pushout construction operated during computation of the composition aggregates the unrelated organization components together without adding any new components and merges the shared components defined in the configuration. As a result, the composite organization has all components of both organizations while eliminating duplicates identified in the shared part. Let *cp* be a connection point from either O_1 or O_2. From Definition 5.6 and Definition 5.7, *cp* should be connected to external entities. We have two cases:

Case 1: cp is not part of the configuration i.e. cp ≠ exit and cp ≠ entry

Based on the construction proposed in Proposition 5.11**,** cp and its external entities are not part of the shared organization. Therefore, they appear in the pushout organization (the pushout operation only merges element from the shared organization). We have:

$\{cp\} \in O_3.EntryCP$ or $\{cp\} \in O_3.ExitCP$ (depending if cp was an entry or exit connection point).

Case 2: cp is part of the configuration i.e. cp = exit or cp = entry

Based on the construction proposed in Proposition 5.11 (step 1), external entities connected to connection point cp are mapped to internal entities. Hence, in the pushout object, cp is connected to internal entities. By Definition 5.6 and Definition 5.7, cp is no longer a valid connection point. Hence, we have:

$\{entry\} \notin O_3.EntryCP$ if cp = entry

$\{exit\} \notin O_3.ExitCP$ if cp = exit.

Therefore, from case 1 and 2, we have:

$O_3.ExitCP = O_1.ExitCP \cup O_2.ExitCP - \{exit\}$

$O_3.EntryCP = O_1.EntryCP \cup O_2.EntryCP - \{entry\}$ □

5.4 Related Work

Several organizational frameworks for multiagent systems have been proposed to deal with the complexity of large software systems [39, 44, 48, 74, 90, 125]. Moreover, it has been long suggested that decomposition in OMAS would help cope with the complexity of systems [65]. However, most of the current methodologies suggest the decomposition of large organizations into smaller ones without providing a rigorous process to recombine them. For instance, Ferber et al. proposes partitioning a multiagent system into groups [39]. Those groups can interact with each other by having a gatekeeper agent participating in multiple groups. However, there is no formal description on the way those groups are aggregated into one consistent system. Similarly, Zambonelli et al. propose a methodology based on organizational abstraction for multiagent systems [125]. They recognize the importance of reusability in OMAS and propose dividing the system into loosely-coupled sub-organizations to reduce design complexity. Nonetheless, reconnection of those sub-organizations is left to the designer who needs to know the internal behavior of each sub-organization in order to assemble them appropriately. Hence, in most cases, the designer informally uses the agent interaction

mechanisms to integrate multiagent organization designs. My approach proposes a rigorous specification of the interfaces necessary to adequately compose various multiagent organizations, thus allowing a better reusability and maintainability of complex OMAS. Organizations can then be developed by different designers and later combined via their specified interfaces.

Besides organizational approaches, other frameworks, which have been derived from component-based software engineering, propose a compositional approach for building multiagent systems (MAS). In DESIRE [11], Brazier et al. propose a compositional design of MAS in which agents are represented by components that can be composed of subcomponents. They claim that a multiagent system can be built incrementally by composition of several components at different levels of abstraction. However, DESIRE, like other frameworks based on components, is agent-centric and does not consider OMAS, which provides better abstraction for complex systems.

Dastani et al. define a coordination model for agents and MAS called Reo [24]. Their model handles coordination only and does not consider the entities (agents or MAS) that need to coordinate. The designer chooses the entities participating in the coordination and Reo handles the coordination. However, Reo does not specify any interfaces that MAS need to provide in order to be composed. In my work, I not only specify the coordination mechanisms necessary for organizations to be composed, but I also specify standard interfaces that allow organizations to be interconnected without knowing the details of their design models.

Few works explicitly consider using service-oriented principles for developing OMAS. Most of the work unifying services and agents concepts concern agent-based service-oriented systems, in which agents are used as a mere wrapper of services [60, 61]. Cao et al. propose a methodology called Organization and Service Oriented Analysis and Design (OSOAD) [18]. OSOAD combines organizational modeling, agent-based design and service-oriented computing in order to build complex open OMAS. Their approach is similar to mine in the sense that complex OMAS are built based on service-oriented principles. However, in their approach, the system is designed using an organizational approach and services are offered at the agent level, whereas my approach considers entire multiagent organizations as service providers. This approach allows us to develop cooperative services, which are services that cannot be provided by an individual agent.

5.5 Summary

I have presented an approach to ease the development of complex OMAS by developing reusable multiagent organizations. Most current approaches only use decomposition to benefit separation of concerns during design. They lack of formalization concerning the composition of the sub-organizations, which makes such sub-organizations challenging to reuse and the resulting applications very difficult to maintain.

Our approach combines service-oriented principles with organizational concepts in order to provide MAS designers with predefined reusable multiagent organizations. I have described how to design such organization-based multiagent services so that they expose the appropriate interfaces in order for potential consumers to request the operations they provide. Moreover, I have described our composition process that merges multiagent organizations into a single composite organization.

A significant advantage of our approach is the ability to compose multiagent organizations to develop a wide variety of complex applications. Indeed, it is easier to design complex organizations by using collections of simpler organization modules wrapped up as services. Moreover, by designing multiagent organizations with a separation of concerns mindset and by reusing identical goals during the composition, my approach ensures that the composite organization will have the minimum number of goals, whereas an ad-hoc design that did not plan for reuse could result in duplicate goals in different subtrees.

Finally, a composite organization can efficiently manage the available resources by avoiding the overhead that would have resulted from coordinating several organizations. In fact, having a single organization simplifies reorganization tasks by allowing us to reuse work already done concerning reorganization of individual organizations [104, 107, 126].

The service-oriented approach proposed in this chapter represents one way of using the theoretical framework proposed in Chapter 4. It allows the construction of organization homomorphisms based on specific definitions of what design elements can be considered identical. However, it is possible to define other approaches that would compose design elements based on other criteria.

CHAPTER 6 - CASE STUDIES

"That some achieve great success, is proof to all that others can achieve it as well." — *Abraham Lincoln*

"One day Alice came to a fork in the road and saw a Cheshire cat in a tree. "Which road do I take? " she asked. His response was a question: "Where do you want to go? " "I don't know, " Alice answered. "Then," said the cat, "it doesn't matter." "
—*Lewis Carroll, Alice in Wonderland*

In this chapter, I demonstrate the validity and usefulness of my approach by developing two applications using several services. For the first application, I design several cooperative robotic services and show they can be composed to create a complex organization design. In addition, I propose other organization designs developed using conventional design approaches and compare them with my approach. The second application is a Wireless Sensor Networks application that uses one service. I provide two different designs of the same service and show how the service-oriented framework allow the permutation of those two designs without any modifications to the main design. Furthermore, I implement this application in order to demonstrate some of its adaptive properties.

6.1 Cooperative Robotic for Airport Management (CRAM)

6.1.1 Description

In order to demonstrate the validity of our service-oriented framework for designing MAS applications, I design an application called *Cooperative Robotic for Airport Management*

(CRAM). In this application, a team of heterogeneous robots is in charge of handling some aspects of the airport management task. Essentially, the team needs to monitor the airport building for possible threats, perform preliminary cargo inspections and clean the building. Threats detected need to be safely removed and cargo that failed the preliminary inspection need to be sent to another location for further inspection.

Services for developing MAS applications can exist in a repository of services and then reused or they can come from the decomposition of a particular problem into smaller ones. For this example, I develop two repository services: the *transportation service* and the *cleaning service* and show how they can be used to develop our CRAM application.

```
<service name= Transportation>
  <operation name= push>
    <connector>
            <protocol> push_protocol </protocol>
            <event> push_event(object) </event>
    </connector>
    <conditions>
            <pre> The object exits </pre>
            <post> The object is at its final destination </post>
    </conditions>
  </operation>
  <operation name= carry>
    <connector>
            <protocol> carry_protocol </protocol>
            <event> carry_event(object) </event>
    </connector>
    <conditions>
            <pre> The object exits </pre>
            <post> The object is at its final destination </post>
    </conditions>
  </operation>
</service>
```

Figure 6.1. Transportation Service specification

6.1.2 The Transportation Service

The Transportation Service is a service that allows objects to be transported at a certain destination. Object transportation is a common problem in cooperative robotics [77, 92, 99, 109, 116] and many applications could definitely benefit from such a service. This service involves moving an object to a certain destination by either pushing it or carrying it. Thus, the transportation service proposes two operations: *push* and *carry* (Figure 6.1). The push operation is specified as follows:

- The operation is initiated by the event *push_event* with the object as a parameter.
- Interaction with the operation is made through the protocol *push_protocol*.
- The *precondition* for the operation requires that the object exists.
- The *postcondition* for the operation requires that the object be at its final destination

The carry operation is specified as follows:

- The operation is initiated by the event *carry_event* with the object as a parameter.
- Interaction with the operation is made through the protocol *carry_protocol*.
- The *precondition* for the operation requires that the object exists.
- The *postcondition* for the operation requires that the object be at its final destination.

To design the Transportation Service, I propose an organization called *Cooperative Transportation*, which uses a two-robots team for pushing and carrying objects to their requested destination. The goals and roles used to carry out those operations are shown in Figure 6.2. In this figure, the top-level goal is *Transport Object*. This goal is further decomposed into two subgoals: *Push* and *Carry*. The *Push* goal is decomposed into two subgoals: *Start Pushing* and *Assist Pushing* that are the two leaf goals that the two-robots team is pursuing while pushing an object. In the role model, we design the roles *Pusher*, *Helper*, *Lifter* and *Carrier* such that they each can achieve one leaf goal. The Cooperative Transportation organization provides the Transportation Service specified in Figure 6.1. The entry connection points are *⟨Push, Pusher⟩* and *⟨Deliver, Carrier⟩*. Entry connection point ⟨Push, Pusher⟩ provides the operation *push* while entry connection point ⟨Deliver, Carrier⟩ provides the operation *carry*.

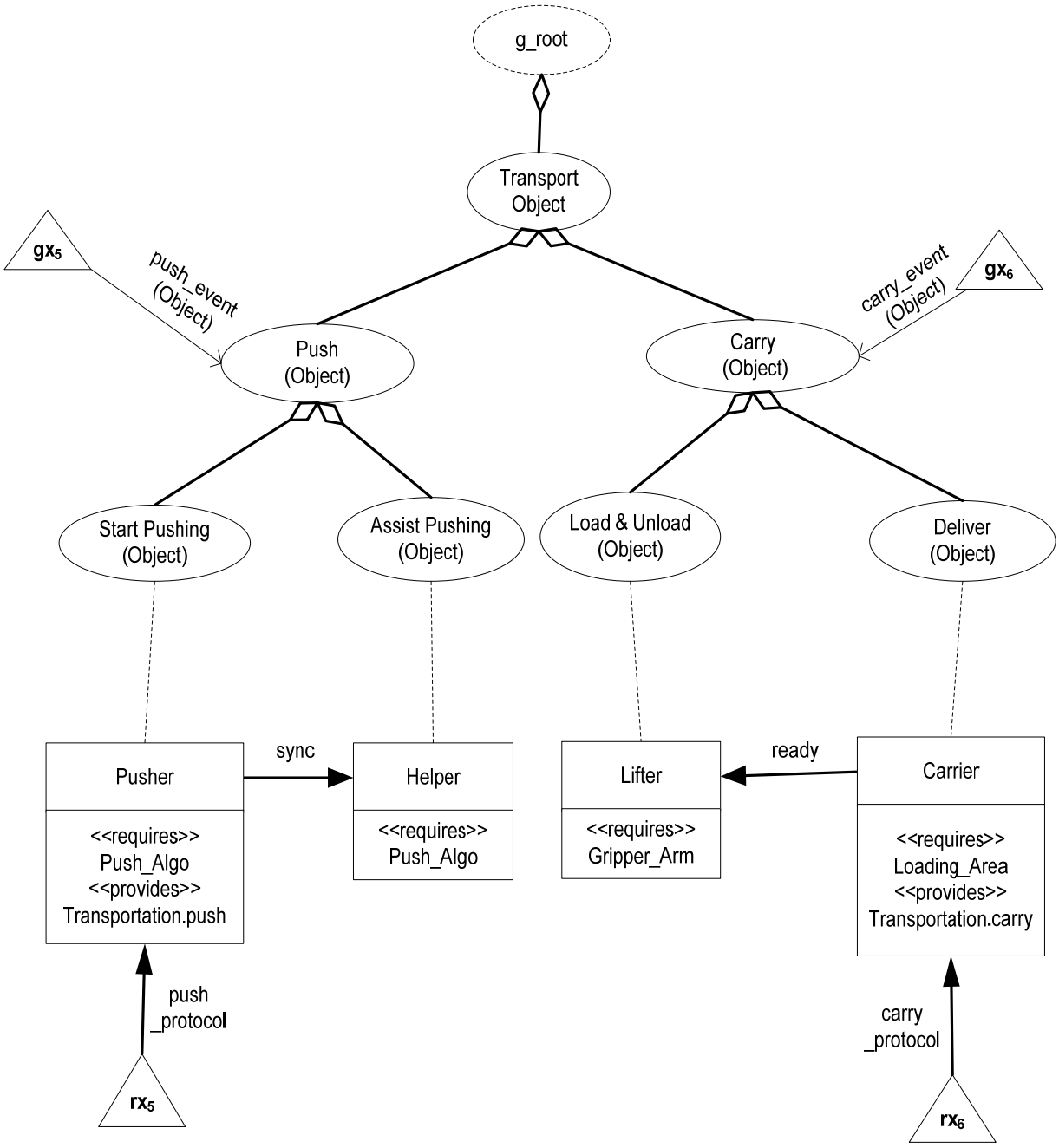

Figure 6.2. Transportation Organization

6.1.3 The Cleaning Service

Cooperative cleaning is a common problem in cooperative robotics and several works have been published regarding the use of robots for cleaning [78, 91, 110, 115]. Here, we propose a cleaning service whose main operation is to clean a given area. This design is similar to the one proposed in [104]. The specification of this service is presented in Figure 6.3.

We design the *Cooperative Cleaning Organization*, shown in Figure 6.4, which involves a team of robots coordinating their actions in order to clean a given area. This organization provides the *Cleaning Service*. The entry connection point providing the *clean* operation is made of the goal *Divide Area* and the role *Leader*. The *Divide Area* goal is in charge of dividing an area into smaller areas that can be handled by individual robots. Once the area to be cleaned has been divided, the *Clean* goal is triggered. The *Clean* goal is decomposed into two disjunctive goals. Hence, it offers two ways of cleaning; the organization can decide to either do a deep clean (*Deep Clean* goal) or just vacuum (*Vacuum* goal). The *Deep Clean* goal is further decomposed into two conjunctive goals: *Sweep* and *Mop*.

```
<service name= Cleaning>
  <operation name= clean>
    <connector>
            <protocol> clean_protocol </protocol>
            <event> clean_event(area) </event>
    </connector>
    <conditions>
            <pre> The area is accessible </pre>
            <post> The area has been cleaned </post>
    </conditions>
  </operation>
</service>
```

Figure 6.3. Cleaning Service specification

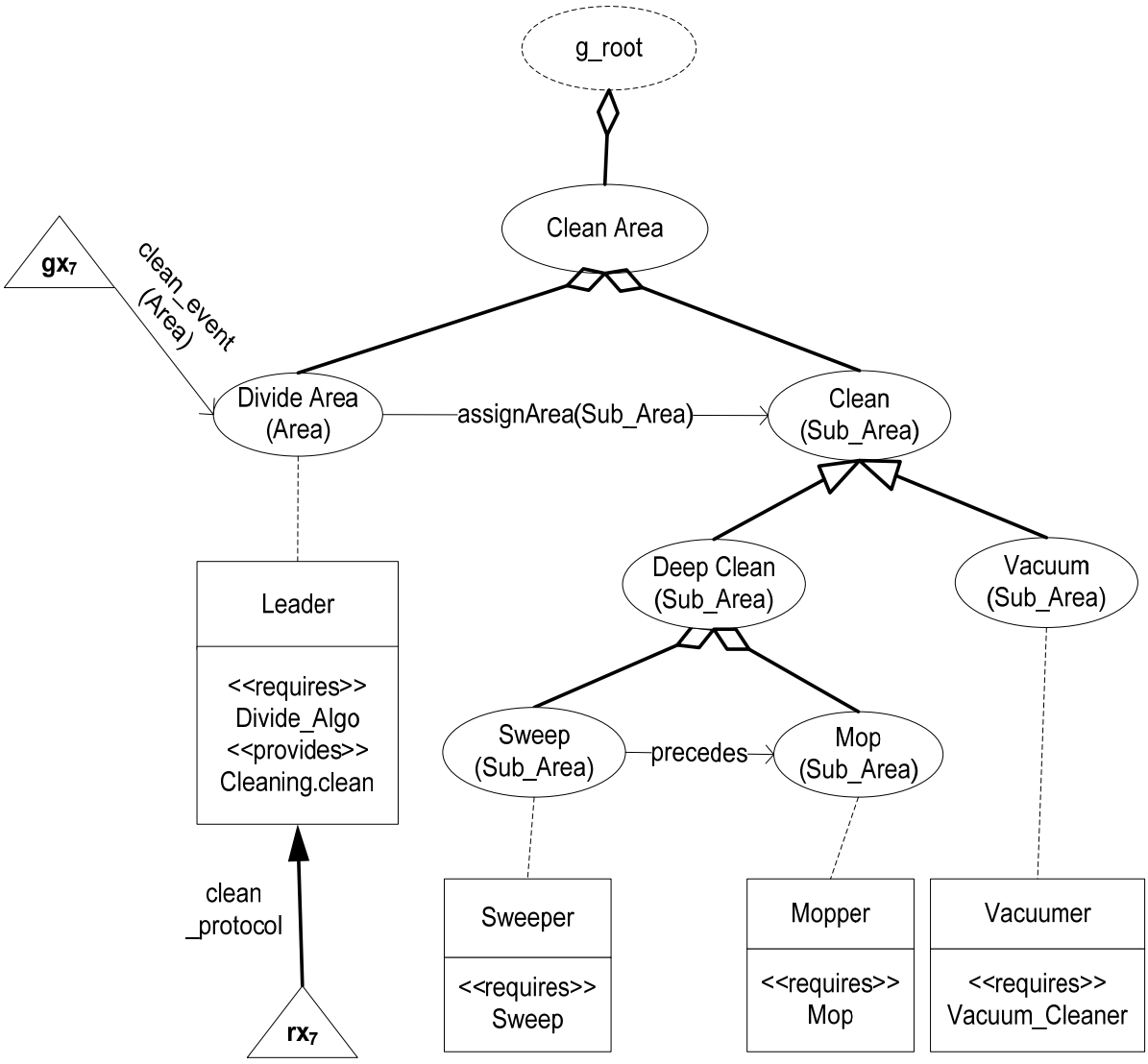

Figure 6.4. Cleaning Organization

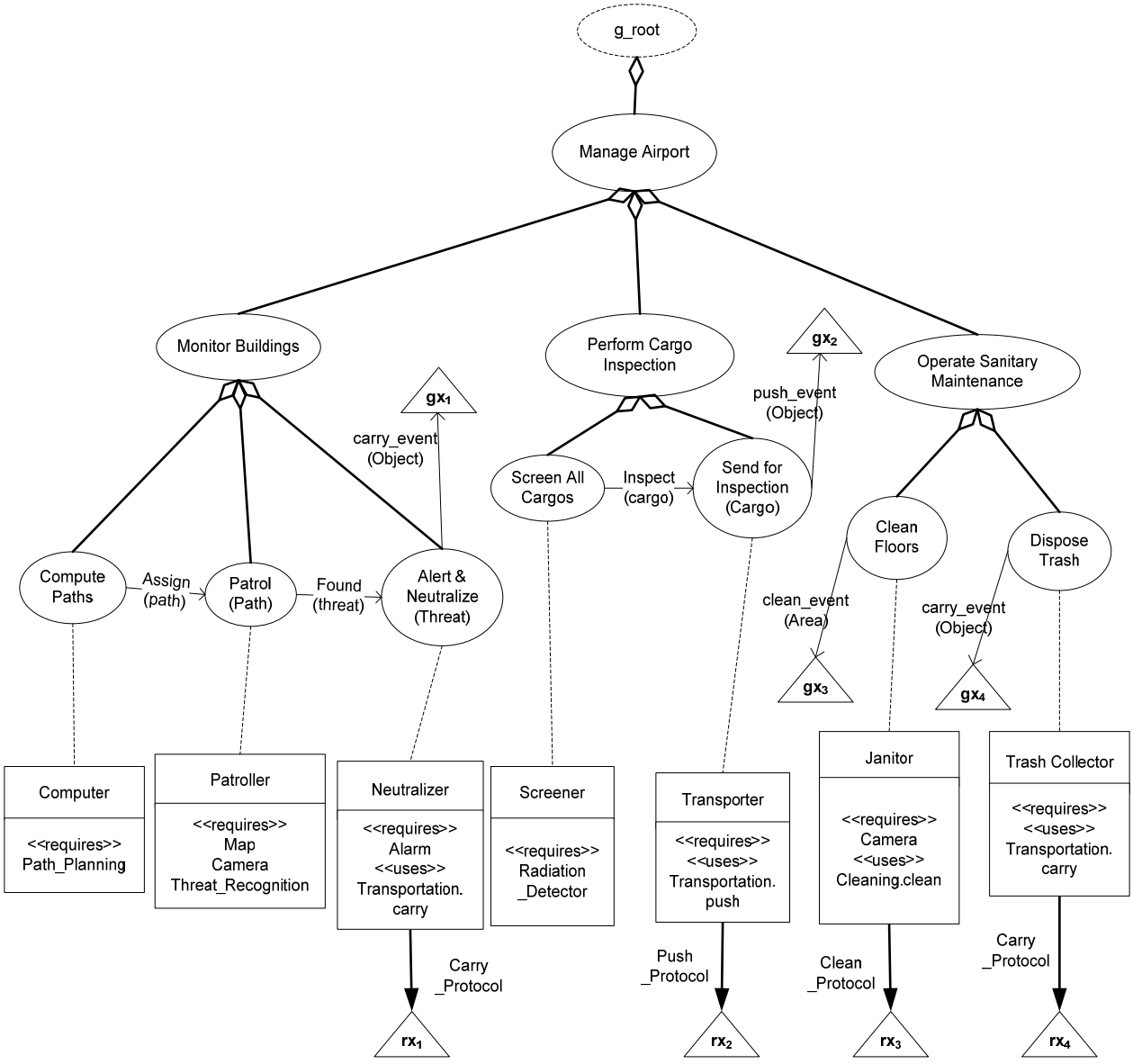

Figure 6.5. CRAM Organization

6.1.4 The Cooperative Robotic for Airport Management organization

We can now design the CRAM organization that uses the services described earlier. The CRAM organization design is presented in Figure 6.5. The main goal of the system, *Manage Airport*, has three conjunctive subgoals that represent the three main tasks of our system: *Monitor Building*, *Perform Cargo Inspection*, *Operate Sanitary Maintenance.* These goals are in turn further decomposed into conjunctive leaf goals. In addition, for each leaf goal in the CRAM organization, we create a role that can achieve it. Moreover, we identify that the *Neutralizer*, *Transporter* and *Trash Collector* roles can use the *Transportation Service* for their achievement. We also decide to use the *Cleaning Service* during the achievement of the *Janitor* role. Thereby, the CRAM organization created contains the following four exit connections points: ⟨*Alert and Neutralize, Neutralizer*⟩, ⟨*Send for Inspection, Transporter*⟩, ⟨*Clean Floors, Janitor*⟩, and ⟨*Dispose Trash, Trash Collector*⟩.

6.1.5 The Composition Process

In this section, I show how to compose the CRAM application with all the required services in order to create a single composite organization. Once the core application has been designed, we must complete the design by composing required services. As mentioned before, the CRAM requires the *push* and *carry* operations from the *Transportation Service* and the *clean* operation from the *Cleaning Service*, which are provided by the *Cooperative Transportation* organization and the *Cooperative Cleaning* organization respectively. Let *Cram, Transportation, and Cleaning* be the CRAM, Cooperative Transportation and Cooperative Cleaning organizations respectively. Moreover, we define the following connection points:

- *cp_Neutralize* = ⟨Alert and Neutralize, Neutralizer⟩, *cp_Trash* = ⟨Dispose Trash, Trash Collector⟩, *cp_Janitor* = ⟨Clean Floors, Janitor⟩, *cp_Transport* = ⟨Send for Inspection, Transporter⟩ from *Cram*,
- *cp_Push* = ⟨Push, Pusher⟩ , and *cp_Carry* = ⟨Carry, Carrier⟩ from *Transportation* such that cp_Push and cp_Carry provide operation *push* and *carry* respectively,
- *cp_Clean* = ⟨Divide Area, Leader⟩ from *Cleaning* such that cp_Clean provides operation *clean*.

We have identified four exit connection points in the CRAM organization. Hence, we need four composition operations in order to compose these exit connection points with appropriate entry connection points from provider organizations. As explained in section 5.3, during the composition, entry and exit goals are connected via a trigger and entry and exit roles are connected via a protocol. Let us first compose *Cram* with *Transportation* providing the *carry* operation. As there are two connection points requiring the carry operation (cp_Neutralize and cp_Trash), we need to have two configurations of connection points $carry_1$ and $carry_2$ such that:

$carry_1$= ⟨cp_Neutralize, carry, cp_Carry⟩ and

$carry_2$ = ⟨cp_Trash, carry, cp_Carry⟩.

Hence, we define organizations $cram\_carry_1$ and $cram\_carry_2$ as:

$$\textbf{\textit{Cram\_carry}}_1 = \textbf{\textit{Cram}} \vdash\!\!\!-^{\textbf{carry1}} \textbf{\textit{Transportation}} \quad \text{(C1)}$$

$$\textbf{\textit{Cram\_carry}}_2 = \textbf{\textit{Cram\_carry}}_1 \vdash\!\!\!-^{\textbf{carry2}} \textbf{\textit{Transportation}} \quad \text{(C2)}$$

Organizations $Cram\_carry_1$ and $Cram\_carry_2$ are shown in Figure 6.6 and Figure 6.7 respectively.

As connection point cp_Transport has not been connected yet, it is still available in organization $Cram\_carry_2$. Thereby, as cp_Transport uses the push operation, we define a configuration linking cp_Transport and cp_Push, which is a connection point from the Transportation organization providing the push operation. Hence, we define the configuration *push* as:

push = ⟨cp_Transport, push, cp_Push⟩;

We can now compose the $Cram\_carry_2$ organization obtained previously with Transportation over the push configuration. We have:

$$\textbf{\textit{Cram\_push}} = \textbf{\textit{Cram\_carry}}_2 \vdash\!\!\!-^{\textbf{push}} \textbf{\textit{Transportation}} \quad \text{(C3)}$$

The organization Cram_push is shown in Figure 6.8.

Finally, we define the configuration *clean* linking connection point cp_Janitor, which uses the *clean* operation, with cp_Clean, which provides the *clean* operation. We have:

clean = ⟨cp_Janitor, clean, cp_clean⟩.

Composing the Cram_push with Transportation over clean, we obtain:

$$\textbf{\textit{Cram\_clean}} = \textbf{\textit{Cram\_push}} \vdash\!\!\!-^{\textbf{clean}} \textbf{\textit{Cleanning}} \qquad \text{(C4)}$$

The organization Cram_push is shown in Figure 6.9.

Cram_clean represents a complete application, i.e. it does not use any services. For clarity, only the shared and composite organizations are shown in Figure 6.6, Figure 6.7, Figure 6.8 and Figure 6.9. The details of the organizations and organization homomorphisms used in composition C1, C2, C3 and C4 above are discussed in Appendix A.2.

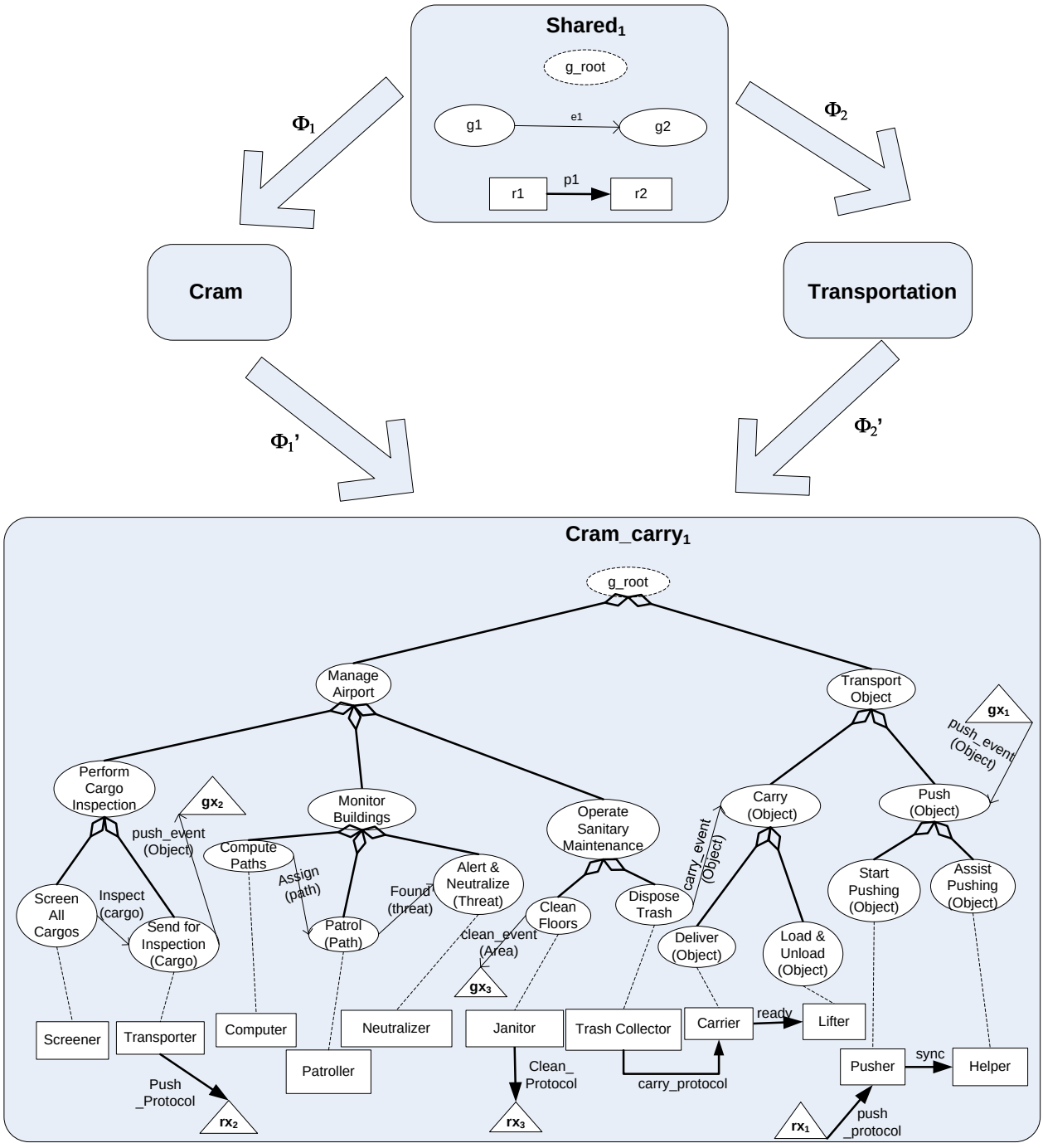

Figure 6.6. Cram_carry$_1$ as the composition of Cram with Transportation over configuration carry$_1$ for operation carry.

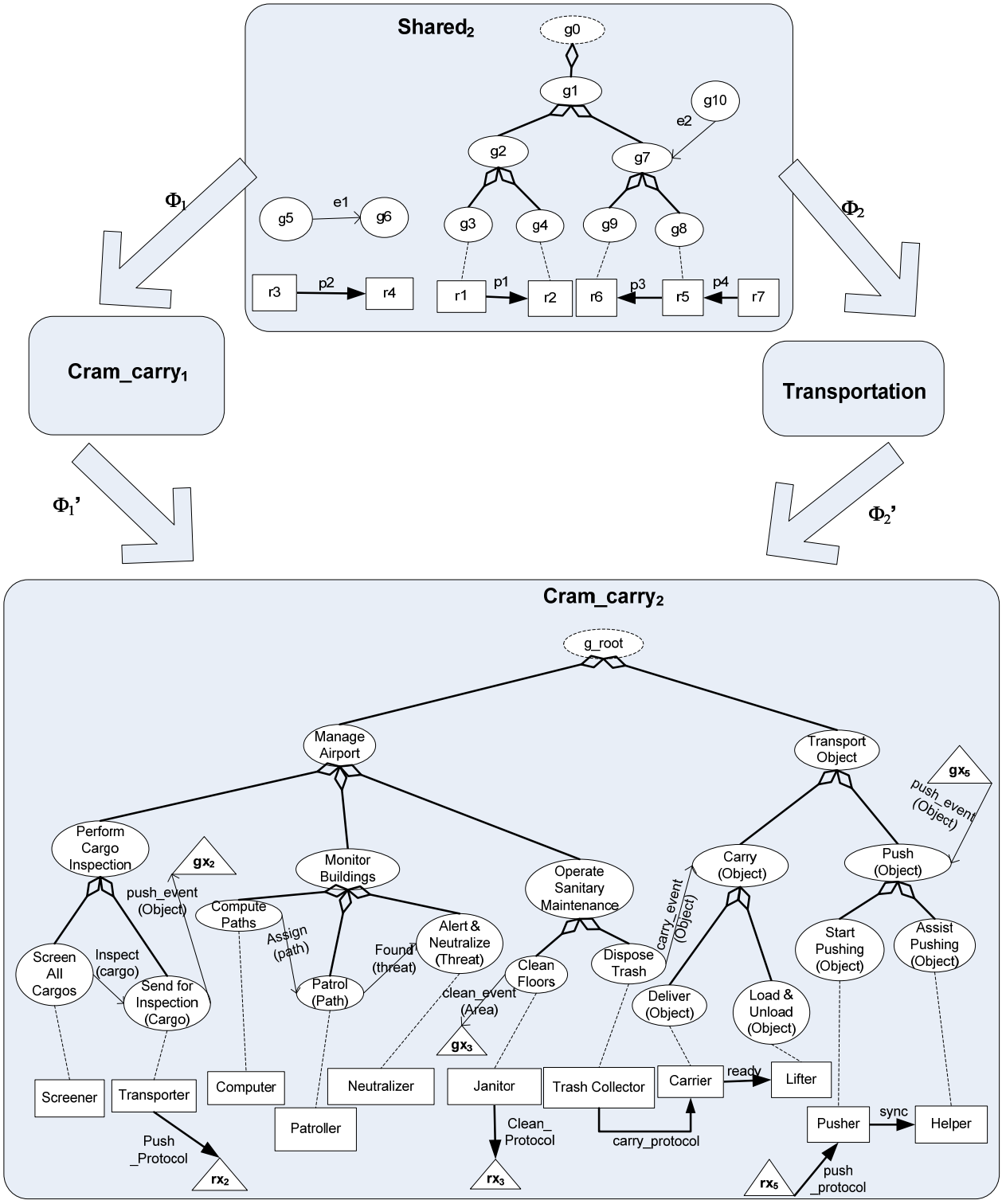

Figure 6.7. Cram_carry$_2$ as the composition of Cram_carry$_1$ with Transportation over configuration carry$_2$ for operation carry

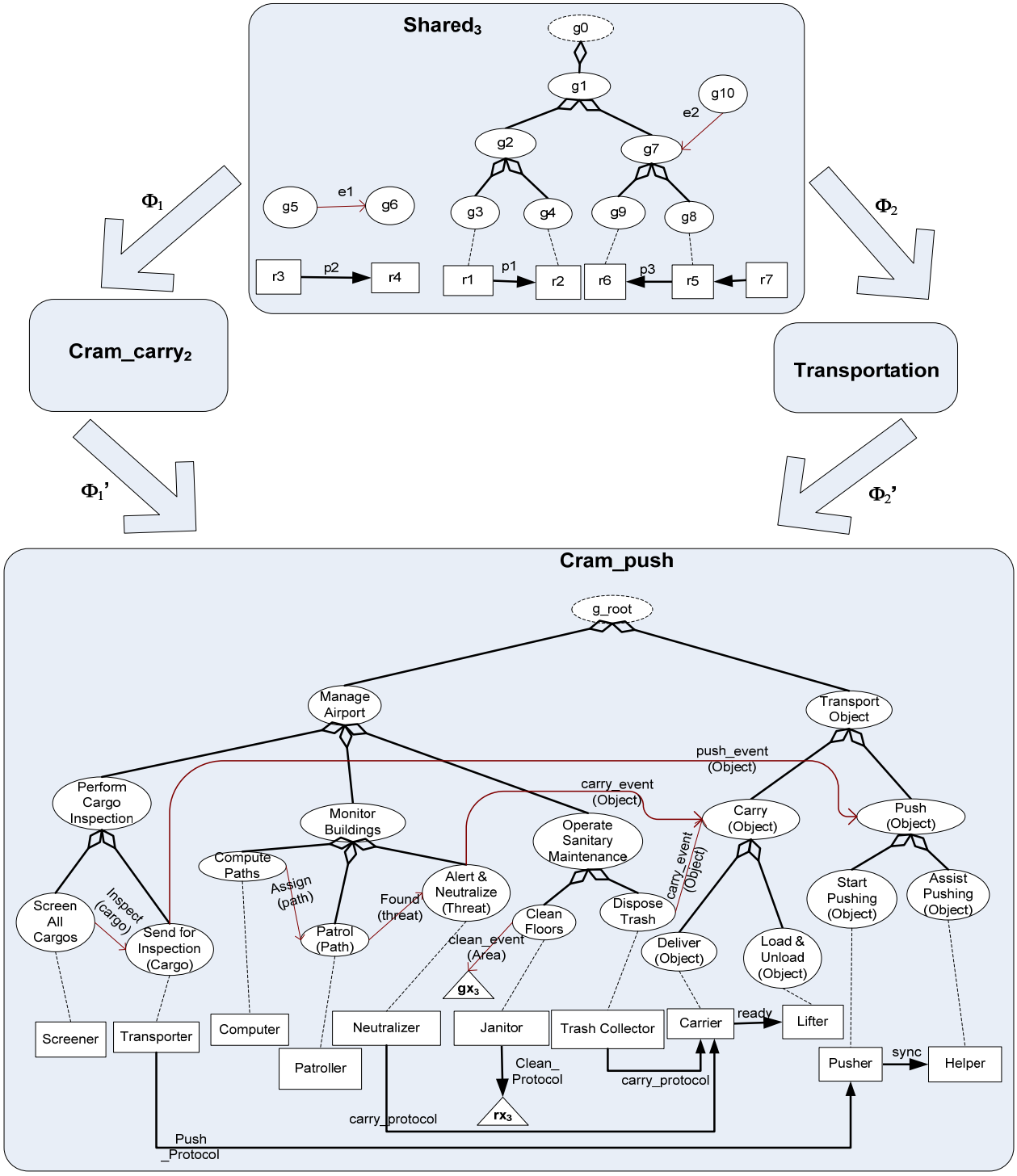

Figure 6.8. Cram_push as the composition of Cram_carry$_2$ with Transportation over configuration carry$_2$ for operation push

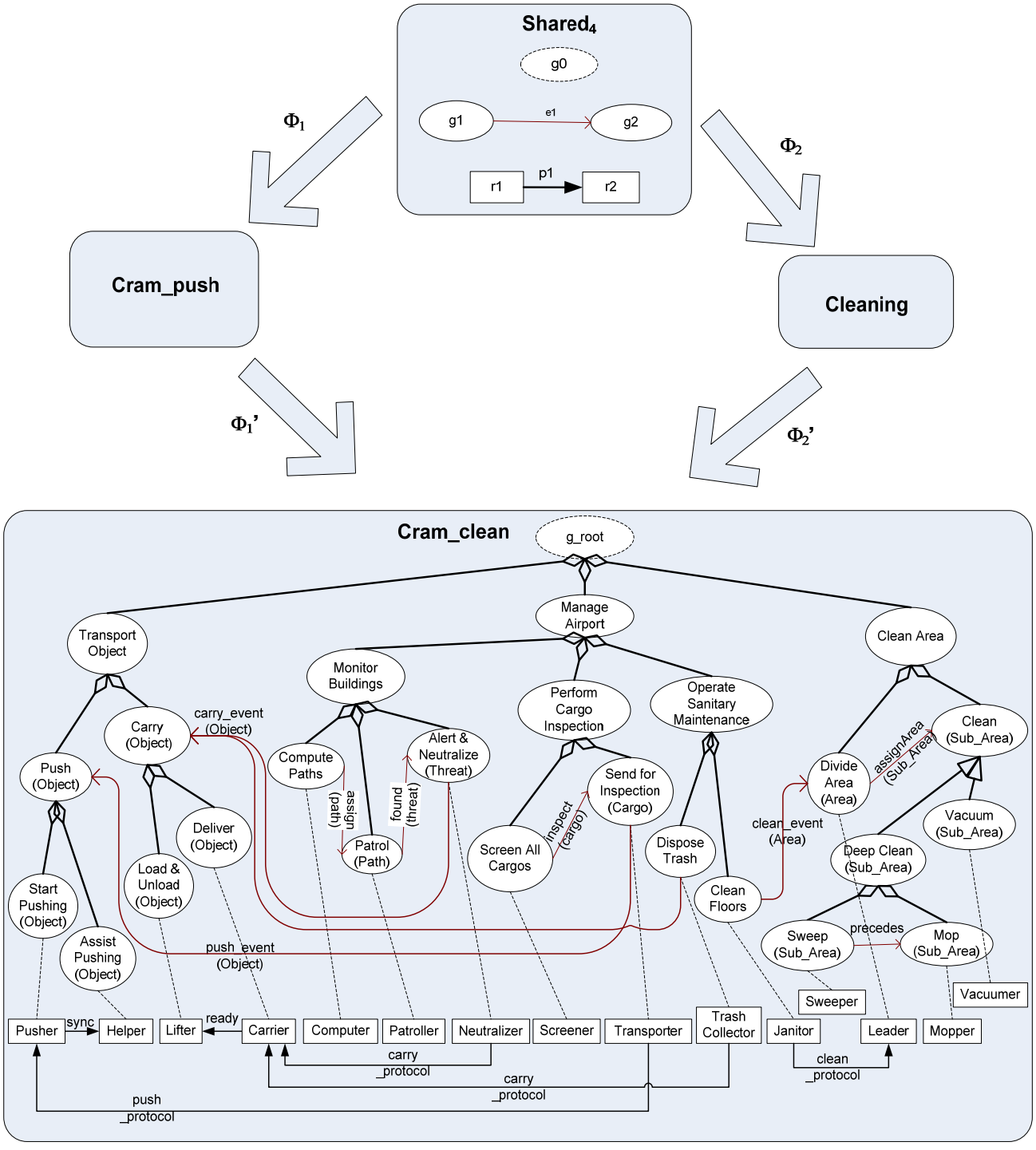

Figure 6.9. Cram_clean as the composition of Cram_push with Cleaning over configuration clean for operation clean

6.1.6 Comparison with an ad-hoc design

In order to compare my compositional design with another design, I propose an alternate design of the CRAM application based on the notion of sub-organizations advocated by numerous methodologies [39, 44, 125]. The *Transportation* and *Cleaning* organization requirements are developed as sub-organizations and then integrated into the main CRAM design. I am interested in evaluating this ad-hoc design and my proposed compositional design with regards to reusability, redundancy and flexibility of the organization design models.

Following the definition of *reusability* suggested by Frakes and Terry [42] and Bansiya and Davis [4], I define reusability of a design model DM as the number of design components from DM reused from previous projects. Hence, for the ad-hoc design, I assume that at least one project was developed previously and resulted in the design of the Transportation and Cleaning sub-organizations. For the compositional design, I assume the prior existence of the Transportation and Cleaning services as designed in Section 6.1.5. For a given organization design, its design components are its goals, roles, triggers and protocols. I compare the number of design components reused in each approach. For the ad-hoc design, only protocols and triggers between reused roles and goals can be counted as reused. This is due to the fact that, in the ad-hoc approach, triggers and protocols involving application goals and roles cannot be taken into consideration as they are application-specific. In fact, most of the time, integration of triggers and protocol require some modifications (like some renamings or parameter modifications) of both the application and the reused sub-organizations in order to have them work properly together. On the contrary, in the compositional approach, triggers and protocols linked to application components are counted as reused are they are generic and developed as part of the service and consequently do not depend on the application. All applications have to conform to these predefined entities and no modification is necessary. The organization design model for the ad-hoc approach is shown in Figure 6.10. Reused goals and roles are shown with a shade of gray. The design model for the composite approach is the one obtained from the previous section (Cram_clean organization) and is shown in Figure 6.11. For this design, reused components are all design components originally coming from the services used. The reused goals and roles in this design are also shown with a shade of gray.

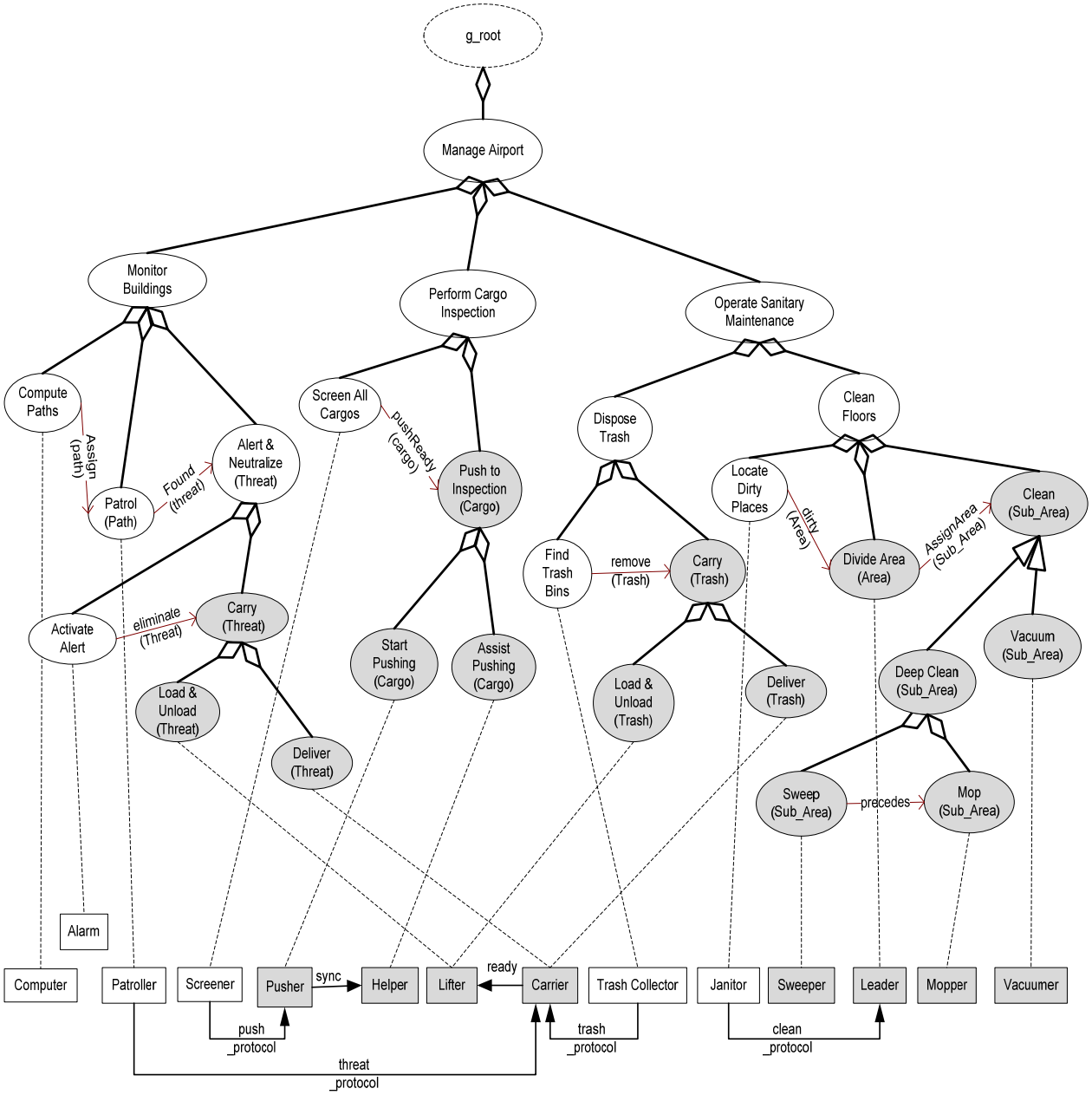

Figure 6.10. Ad-hoc design of the complete CRAM application. Reused goals and roles are in gray.

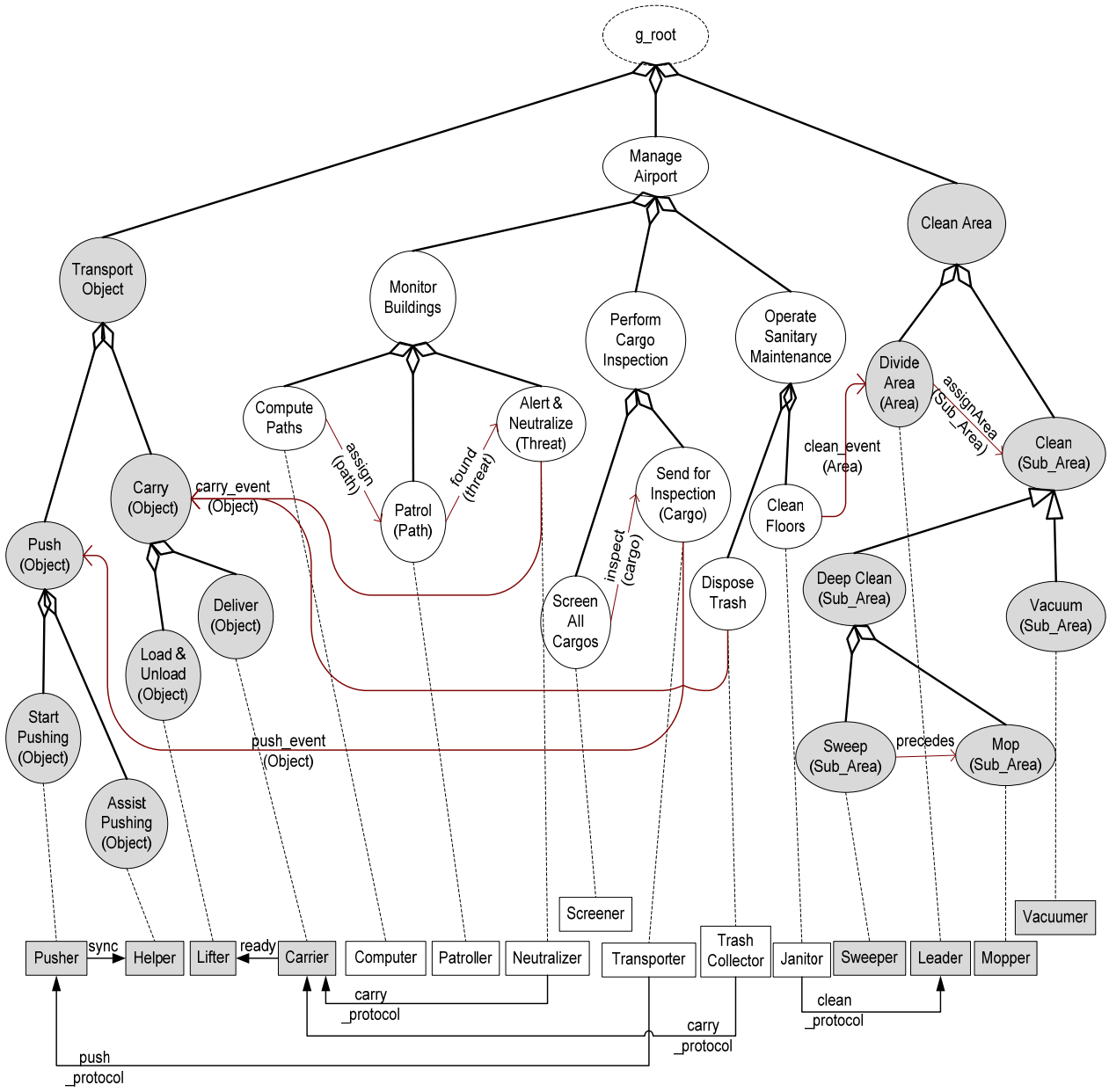

Figure 6.11. Compositional design of the complete CRAM application. Reused goals and roles are in gray.

From the organizational design produced by the ad-hoc approach, we have a total of 29 goals, 5 triggers, 14 roles and 6 protocols. Among these components, we have 12 reused goals, 1 reused trigger, 8 reused roles and 2 reused protocols. Globally, we have reused 23 components out of the 54 components used for the ad-hoc organizational design (43% of reuse). Note that I do not take precedes relations into considerations as they are generic components that do not depend on any applications. From the organizational design produced by the compositional approach, we have a total of 28 goals, 5 triggers, 14 roles and 6 protocols. Among these components, we have 14 reused goals, 5 reused triggers, 8 reused roles and 6 reused protocols. Globally, we have reused 33 components out of the 53 components used for the compositional design (62% of reuse). Therefore, in our example, the compositional approach results in a design with a higher reusability compared to the one obtained from the ad-hoc approach.

In addition, as suggested by Sametinger [103], reusing components often leads to a lot of redundant components. This is particularly the case in the ad-hoc design for which each sub-organization is integrated as a new subtree. For instance, the design in Figure 6.10 shows two subtrees containing the goals for a sub-organization starting with the carry goal. This is necessary as both the Neutralize and the Dispose Trash goals require the completion of a carry goal in order to be achieved. However, in my compositional approach, service goals and roles appear only once as separate subtrees and do not need to be duplicated whenever another application goal requires the same service. Hence, goals related to the carry operation only appear once even though two different goals require it for their achievement.

Following the definition of *flexibility* suggested by Eden and Mens [34] and Bansiya and Davis [4], I define flexibility of a design model DM as the number of design components from DM that need to be modified in order to add a new requirement to the system. In order to have a meaningful comparison of the designs regarding flexibility, I only consider new requirements related to the secondary tasks as changes to the main requirements are handled similarly in both approaches. Let us assume that the requirements for the application have changed and these requirements are not met by the current carry sub-organizations (in the case of the ad-hoc design) and by the current carry operation from the Transportation service (in the case of the compositional design). Let us also assume that another sub-organization (or another service) exists and can be used to meet the new requirements. In the case of the ad-hoc design, a replacement of the carry sub-organization involves modifying all goals, roles, triggers and

protocols belonging to the old sub-organization. Hence, from Figure 6.10, goals *Activate Alert*, *Find Trash Bins*, roles *Alarm*, *Trash Collector*, triggers *eliminate(threat)*, *remove(thrash)* and protocol *trash_protocol*, all need to be modified. Therefore, there are 7 components that would need to be modified in order to cater for the replacement of the current carry sub-organization. On the contrary, by providing clear interfaces, organization designs can be changed without any modifications. Hence, replacing any service provider does not require any modifications. Therefore, the composition design of the CRAM application is more flexible than the ad-hoc design of the same application.

6.2 Adaptive Target Tracking

As another proof-of-concept, I propose the design and implementation of a Wireless Sensor Networks application that uses one service and I demonstrate how the design models used in this work can lead to an adaptive system. In particular, I propose a surveillance application in which sensors are used to collaboratively monitor and track all vehicles entering an area. In this application, sensors are deployed over a large area. As sensors have a limited sensing range, no one sensor can cover the entire area. Hence, agents, which are controlling sensors, need to collaborate in order to provide data covering the entire area of interest. In particular, in order to conserve energy, we would like to provide the maximum coverage with the minimum number of agents. In the absence of targets, agents not monitoring must be in a sleep state in order to conserve energy. Once a target is detected, all sensors in the vicinity must be activated in order to provide the maximum number of measurements that will be used in order to properly locate and identify the target.

For this particular application, I design a Time Synchronization service that is used by sensing roles in order to add timestamp information on the sensing data they are gathering. Then, I design two different providers of the Time Synchronization service. These designs are based on two well-known time synchronization algorithms, namely FTSP [80] and RBS [35]. My goal is to demonstrate how my framework allows the permutation of two provider designs without any modifications to the main design.

In addition, I show how the organizational approach allows the system to autonomously adapt to overcome sensor failures and loss of performance due to capability degradation.

6.2.1 Time Synchronization Service

The goal of a Time Synchronization service is to allow an agent to synchronize its internal clock with another agent. For our example, we assume that agents only need to synchronize their clocks with the base station. The specification of this service is shown in Figure 6.12.

This service has one operation, *synchronize*, which initiates a synchronization process with the base station. This operation periodically returns the current clock offset. The clock offset is defined as the difference between the time on the current node and the time at the base station. Note that the operation does not stop after returning one value. It is executed periodically (the period depends on service providers) until a stop message is received. The offset values and stop messages are exchanged via the *sync* protocol. The operation is instantiated via the *synchronize* trigger that passes the requester agent's information. This information is necessary as any time synchronization provider would need to know which nodes need to synchronize.

```
<service name= Time Synchronization>
  <operation name= synchronize>
    <connector>
            <protocol> sync </protocol>
            <event> synchronize(requester) </event>
    </connector>
    <conditions>
            <pre> A clock is available </pre>
            <post> The clock offset is returned </post>
    </conditions>
  </operation>
</service>
```

Figure 6.12. Time Synchronization Service specification

I propose two organization designs implementing the Time Synchronization service: the FTSP organization and the RBS organization. The FTSP organization is based on the Flooding Time Synchronization Protocol (FTSP) [80]. In FTSP, the base station periodically floods the network with a synchronization message containing its current time. Interested nodes can use the departure and arrival time of these messages to compute the Base Station time and adjust their clock offset accordingly. The details of the algorithm used to compute the offset can be found in [80]. The RBS organization is based on the Reference Broadcast Synchronization (RBS) [35]. In RBS, a beacon node periodically sends a beacon message. A pair of nodes interested in synchronizing use the arrival time of the beacon message to compute their clock offset. They do so by assuming simultaneous reception of the beacon message and exchange their reception time, allowing them to have an estimate of their clock offset. The details of this algorithm can be found in [35].

6.2.2 FTSP Organization

The design of the *FTSP Organization* is shown in Figure 6.13. The top-level goal is the *FSTP* goal. This goal is decomposed into two conjunctive subgoals: *Compute Time,* which is achieved by the role *Receiver* and *Broadcast Time,* which is achieved by the role *Reference*. This organization provides the *Time Synchronization Service*. The entry connection point providing the *synchronize* operation is made of the goal *Compute Time* and the role *Receiver*. Essentially, the agent representing the Base Station plays the reference role and broadcasts synchronization messages that include its current time. Any agent who wants to synchronize plays the receiver role and adjusts its time based on the time received from the reference role (the base station).

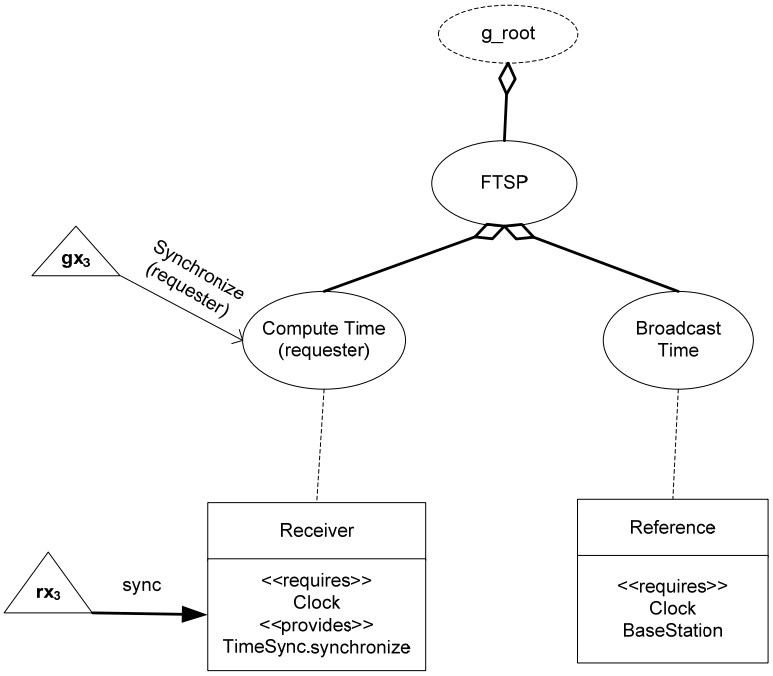

Figure 6.13. FTSP Organization.

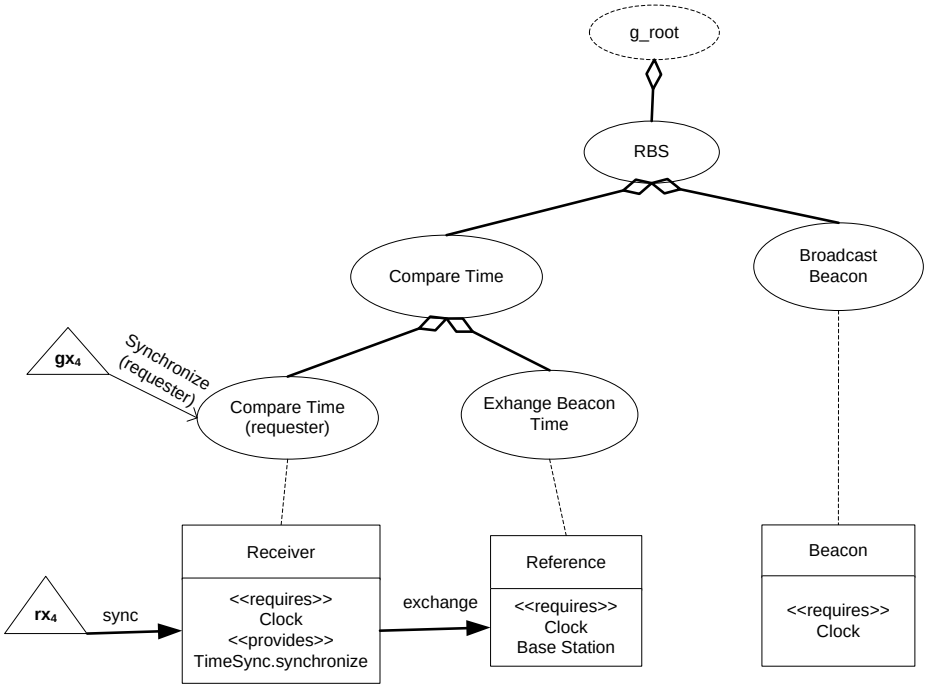

Figure 6.14. RBS Organization

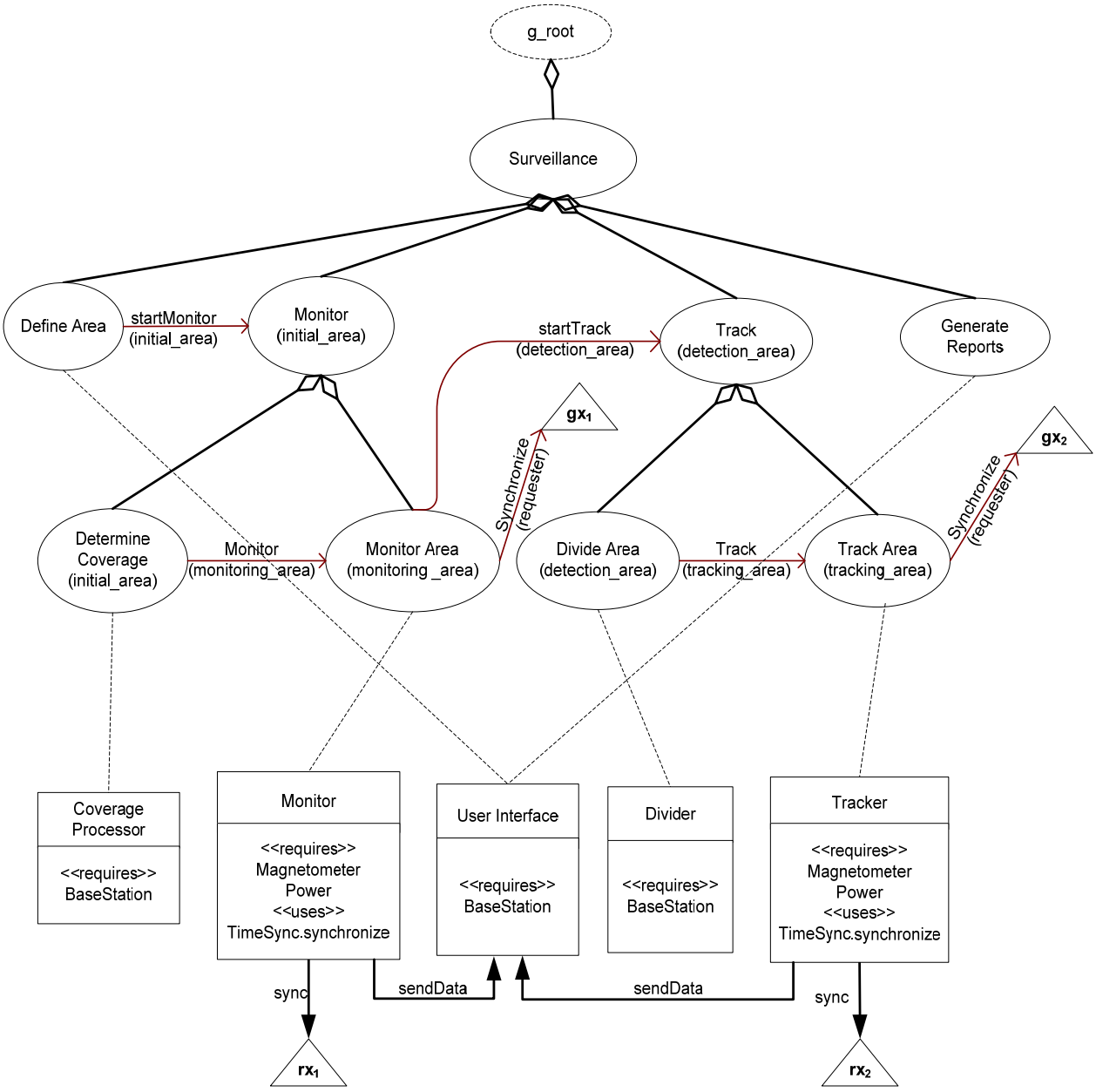

Figure 6.15. Surveillance Organization

6.2.3 RBS Organization

The design of the *RBS Organization* is shown in Figure 6.14. The top-level goal is the *RBS* goal. This goal is decomposed into two conjunctive subgoals: *Compute Time* and *Broadcast Beacon.* The goal Compare Time is further decomposed into subgoals *Compare Time* and *Exchange Beacon Time*. In our design, the goal Compare Time is achieved by role Receiver, the goal Exchange Beacon is achieved by the role Reference and finally the goal Broadcast Beacon is achieved by the role Beacon. This organization provides the *Time Synchronization Service*. The entry connection point providing the *synchronize* operation is made of the goal *Compare Time* and the role *Receiver*. Essentially, the agent playing the broadcast role broadcasts beacon messages. The agent who wants to synchronize plays the receiver role and exchanges synchronization messages with the agent (Base Station) playing the reference role.

6.2.4 Surveillance application

The main application we are trying to design is the surveillance application. From the requirements of our surveillance system, we derived the organization design presented in Figure 6.15. The top-level goal is the *Surveillance* goal. This goal is decomposed into four subgoals: *Define Area*, *Monitor*, *Track*, and *Generate Reports*. The Monitor and Track goals are further decomposed into *Determine Coverage*, *Monitor Area*, *Divide Area* and *Track Area*. The leaf goals are the goals that are actively pursue by the organization by being assigned to agents. Essentially, once an area is defined via the achievement of the *Define Area* goal, an event is generated that triggers the instantiation of the parameterized goal *Monitor (initial_area)*. As the area parameter might be too large for any single agent, the goal *Determine Coverage* is in charge of dividing it into smaller subareas that can potentially be covered by a single agent. Each of these subareas (monitoring_area) is used as a parameter to a *Monitor Area* goal. Each Monitor Area goal can in its turn initiate a *Track* goal for a given detection area. Once again, the detection area defined for the track goal might be too vast for a single agent. Hence, the *Divide Area* goal breaks up the detection area, which results in the creation of a *Track Area* goal for each subarea (tracking_area) identified. Finally, all application data gathered are aggregated in a user-friendly report by the *Generate Reports* goal. Next, we define the roles that can achieve the leaf goals

identified in the goal model. For each role, we specify the behavior that is followed by agents enacting this role. Examples of role behaviors (represented as a state machine) are introduced in Section 6.2.7. We identified five roles necessary to achieve the goals: User Interface, Coverage Processor, Divider, Monitor and Tracker. The User Interface role achieves goals Define Area and Generate Reports. The Coverage Processor role achieves the goal Determine Coverage. The Monitor role achieves the goal Monitor Area. The Divider role achieves the goal Divide Area. Finally, the Tracker role achieves the goal Track Area. Moreover, the surveillance application uses the Time Synchronization service via two connection points: ⟨Monitor Area, Monitor⟩ and ⟨Track Area, Tracker⟩

6.2.5 Compositional Design

The Surveillance application uses the Time Synchronization service to timestamp data messages sent by the agents. It uses the *synchronize* operations from the *Time Synchronization* Service that can be provided by either the *FTSP* organization or the *RBS* organization. Hence, to obtain a complete Surveillance application we can compose it with either the FSTP organization or the RBS organization. The choice can be made based on the capabilities required by the service or some quality of service information. This decision is out of the scope of the framework proposed here. I compose the surveillance application with both providers in order to demonstrate how my framework allows the permutation of providers without any modifications to the main surveillance organization design.

The exit connection points of the Surveillance organization are ⟨Monitor Area, Monitor⟩ and ⟨Track Area, Tracker⟩, with both using the synchronize operation. Let us first compose the Surveillance organization with the FTSP organization. The FTSP organization exposes entry connection point ⟨Compute Time, Receiver⟩. In order to do the composition, we create a configuration comprising each of the exit connection points. Hence, we create configuration $config_1$ and $config_2$ such that:

$config_1$= ⟨⟨Monitor Area, Monitor⟩, synchronize, ⟨Compute Time, Receiver⟩⟩ and

$config_2$ = ⟨⟨Track Area, Tracker⟩, synchronize, ⟨Compute Time, Receiver⟩⟩.

Hence, we define organizations *Surveillance_FTSP* as:

$$\textbf{\textit{Surveillance\_FTSP = (Surveillance}} \vdash^{\textbf{config1}} \textbf{\textit{FTSP)}} \vdash^{\textbf{config2}} \textbf{\textit{FTSP}}$$

Figure 6.16 shows the composite Surveillance_FTSP organization. The details of this composition are in Appendix A.3.

Next, we compose the Surveillance organization with the RBS organization. To do this composition, the original Surveillance organization does not need to be modified as it already possesses all the required interface to be composed with any organization providing the Time Synchronization service. As we have seen, the RBS organization exposes entry connection point ⟨Compare Time, Receiver⟩. In order to do the composition, we create a configuration comprising each of the exit connection points. Hence, we create configuration $config_1$ and $config_2$ such that:

$config_3$= ⟨⟨Monitor Area, Monitor⟩, synchronize, ⟨Compare Time, Receiver⟩⟩ and

$config_4$ = ⟨⟨Track Area, Tracker⟩, synchronize, ⟨Compare Time, Receiver⟩⟩.

Finally, we define organizations *Surveillance_RBS* as:

$$\textbf{\textit{Surveillance\_RBS = (Surveillance}} \vdash^{\textbf{config3}} \textbf{\textit{RBS)}} \vdash^{\textbf{config4}} \textbf{\textit{RBS}}$$

Figure 6.17 shows the composite Surveillance_RBS organization. The details of this composition are in Appendix A.4.

Therefore, the initial Surveillance organization can be composed with any Time Synchronization provider. This composition results in either the Surveillance_FTSP organization or the Surveillance_RBS organization.

In the next Sections, I give more details about the architecture and low-level design steps that can be used to implement systems designed with the approach proposed in this dissertation. The goal is to demonstrate the usefulness of organizational design models to build adaptive and autonomous multiagent systems. The implementation is based on the Surveillance_FTSP design proposed in this Section.

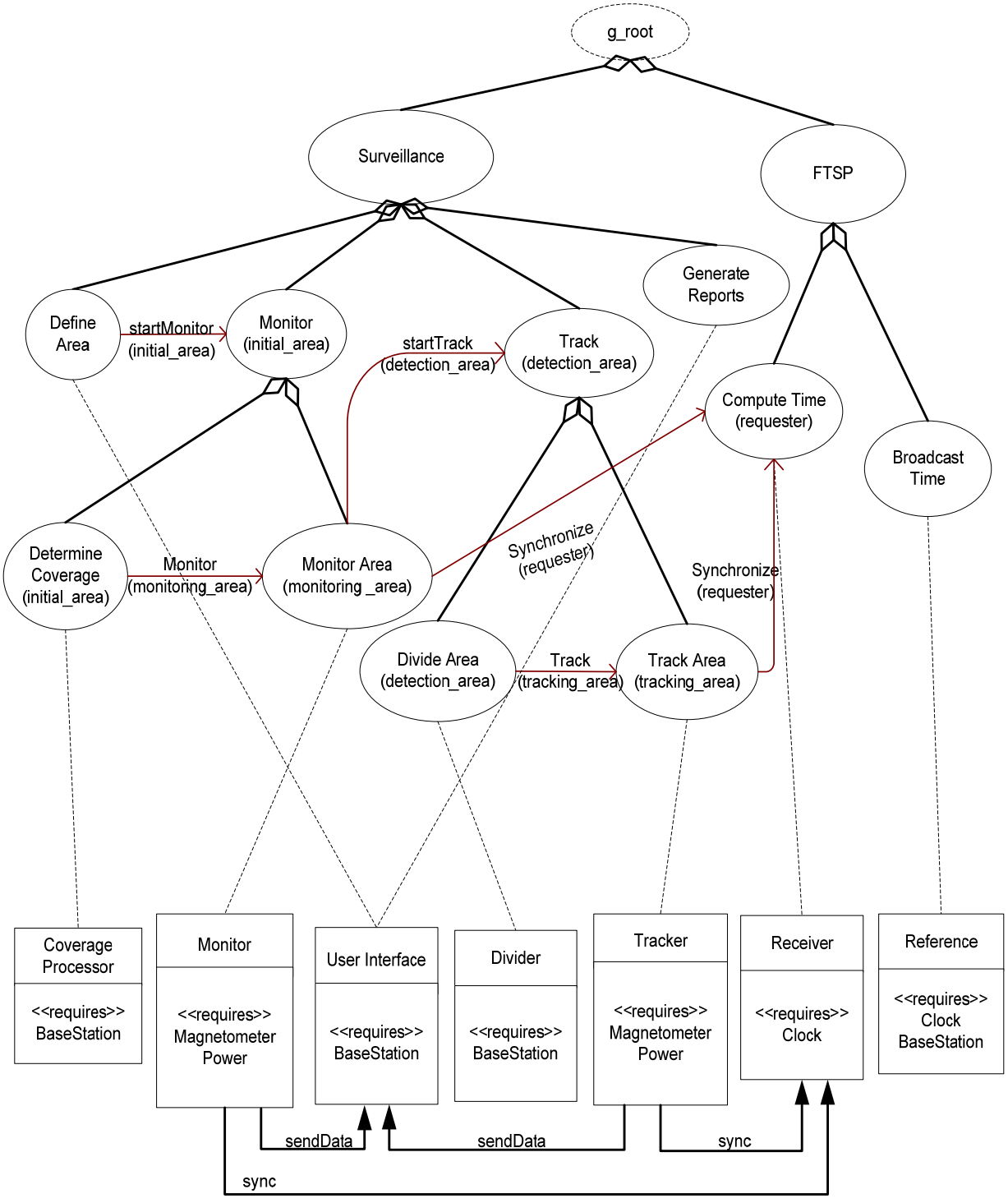

Figure 6.16. Surveillance composed with FTSP

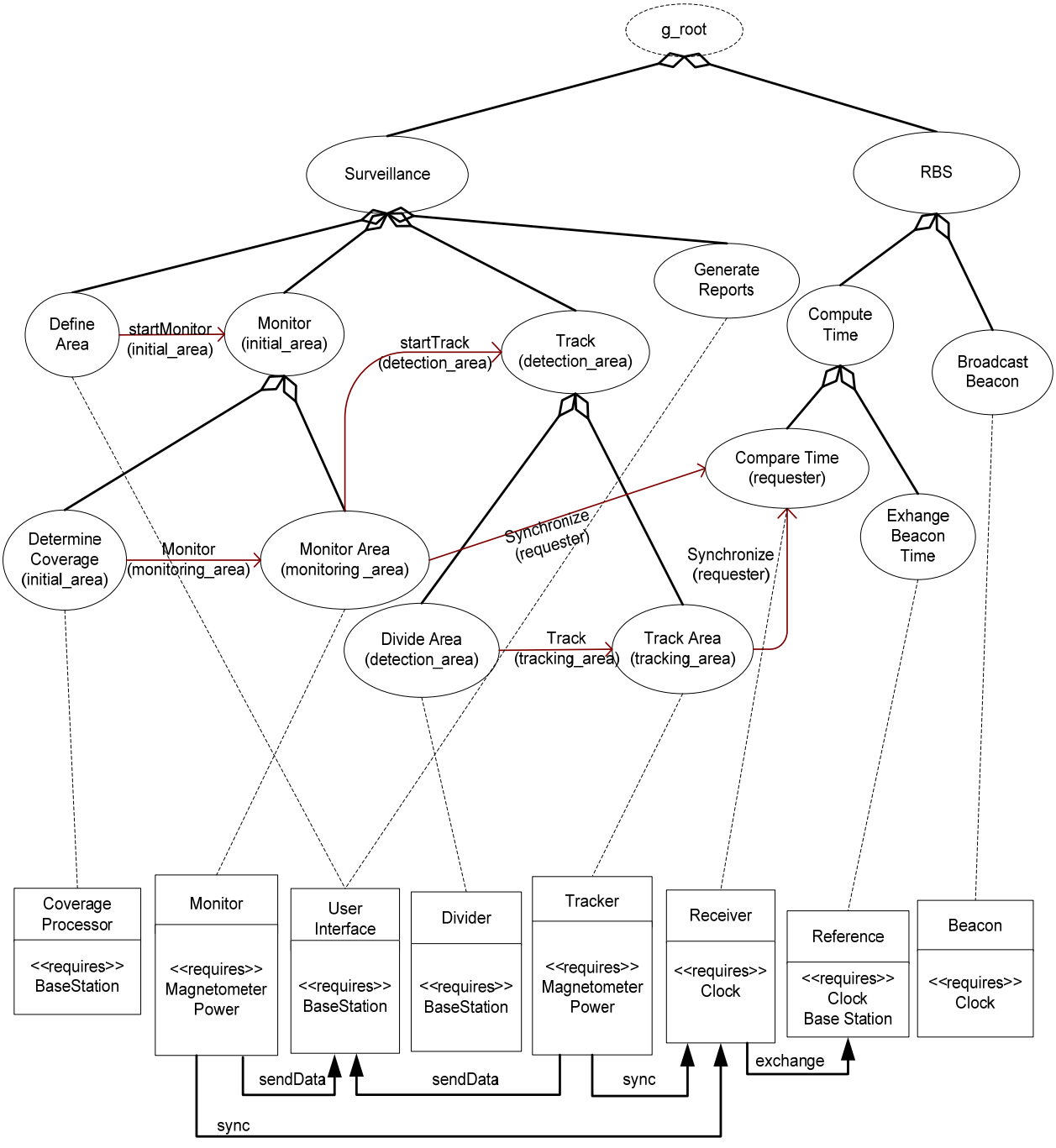

Figure 6.17. Surveillance composed with RBS

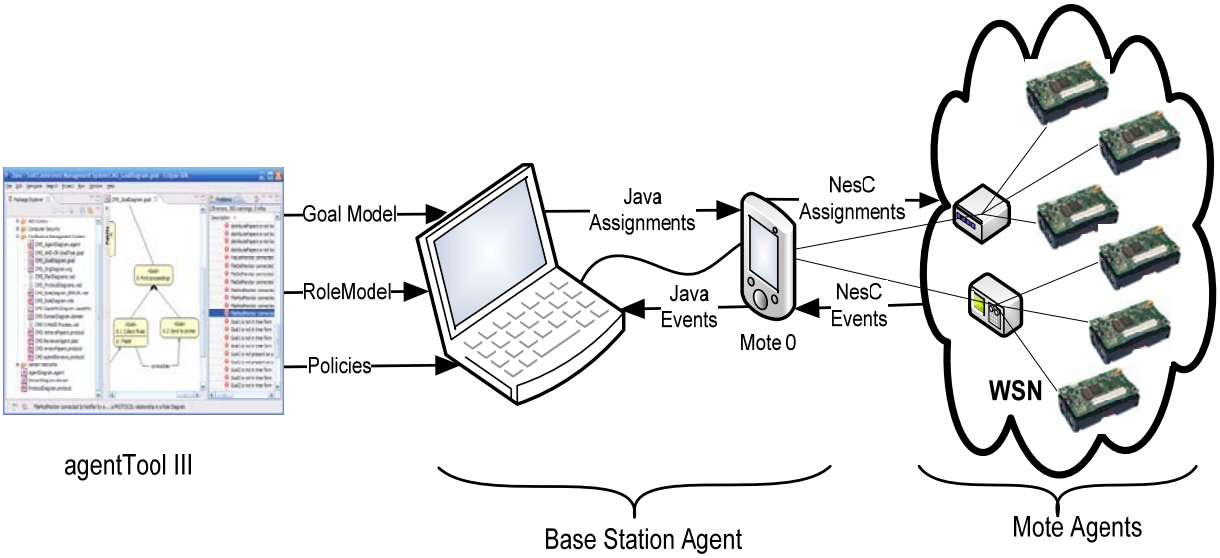

Figure 6.18. Overall System Architecture

6.2.6 System Architecture

In what follows, we consider the Surveillance_FTSP design presented in Figure 6.16. Once the organization design models are defined, we define the agents that will be participating in the organization. Agents are designed to assume roles in an organization. They are designed separately and can participate in the organization as long as they have the required capabilities. In this section, I introduce our agent architecture that exploits the design models to build an organizational knowledge.

In our application, we have two types of agent: Base Station Agents and Mote Agents. We use one Base Station Agent that runs on a PC platform. Our application also contains several Mote Agents running on the Crossbow mote platform, which is a sensor network platform [22].

The overall system architecture is presented in Figure 6.18. The design models are designed using agentTool III (aT^3), a multiagent development environment built on the Eclipse platform. aT^3 supports the development and validation of design models that can be automatically translated into platform specific runtime models. These runtime models are used

by the Base Station agent to make decision about the reconfiguration of the organization. We chose to have the entire organizational knowledge in the Base Station agent because it has more computational resources than the motes and it is less prone to failure. Therefore, in our system, the Base Station agent is the only agent that posses the organizational knowledge and decides the next configuration of the organization. Moreover, we assume that all agents are within communication range of the Base Station agent. The Base Station agent runs on a laptop with a base station mote (mote 0) attached to it. This mote acts as a gateway and allows the Base Station agent running on a PC to interact with the rest of the agents exclusively running on the mote platform.

Once assignments have been made by the Base Station agent, they are passed on to the Mote agents who execute them and return feedback based on possible events that are of interest to the organization (goal failure, goal completion, or application specific events) regarding their assignments. Consequently, the Mote agents have some limitations on their autonomy as they must agree to play their assigned role and pursue the assigned goal. Nonetheless, depending on how the role was designed, they can have freedom in the choice of the specific actions necessary to play a role. The organizational roles presented in this example have been designed with specific steps that all agents must follow.

The Base Station agent and the Mote agents are all agents participating in the same organization and cooperating in order to achieve the main organizational goal. Both types of agents follow the same general architecture that consists of two main components: the *Control Component* and the *Execution Component* (see Figure 6.19). The Control Component performs all organization-level reasoning while the Execution Component provides all the application specific reasoning. This architecture has been designed to facilitate extensibility and reusability and to provide a clear separation between organization control and application. With this architecture, control components can be modified to cater for different organization control mechanisms without sacrificing compatibility with the rest of the system.

The Control Component possesses an *organizational knowledge* component that stores all the knowledge about the structure of the organization and a *control manager* that makes the decisions. The organizational knowledge is created based on organization design models and updated at runtime as organization events arrive. In addition, agents are added to this knowledge base as they appear and register to participate in the organization. Based on the organizational

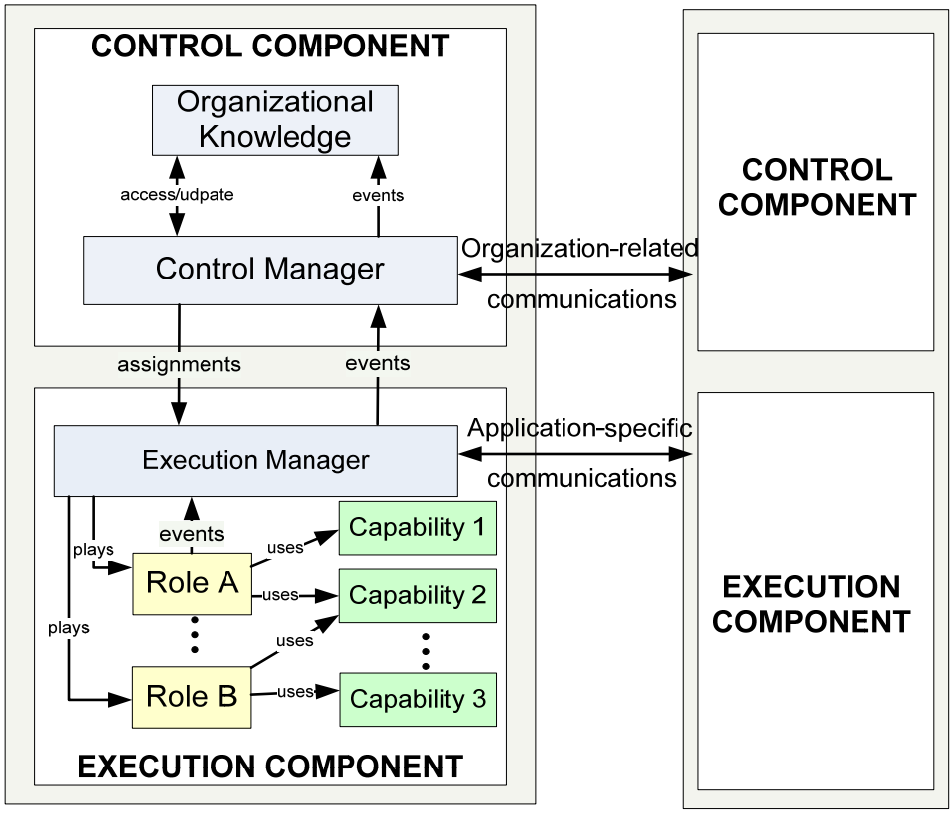

Figure 6.19. Generic Agent Architecture

knowledge, the control manager can reason about the state of the entire organization and decide to reconfigure the organization by including or canceling goals, or by modifying current assignments. As each Control Component has its own view of the organization, a deliberation process might be needed in order to reach a consensus about the next state of the organization. However, in the system described in this research, we have opted to store the entire organizational knowledge at the Base Station agent who can make decision alone. All decisions taken by the Base Station agent are passed on to the Control Component of all the agents that are affected, which then forward these assignments to their attached Execution Components.

The Execution Component corresponds to the application specific part of the agent. It is notified by its Control Component about what role to play in the organization. Once it has been assigned a role, the Execution Component uses its capabilities to execute the plan that has been provided for that role at design time. During role execution, an Execution Component may need

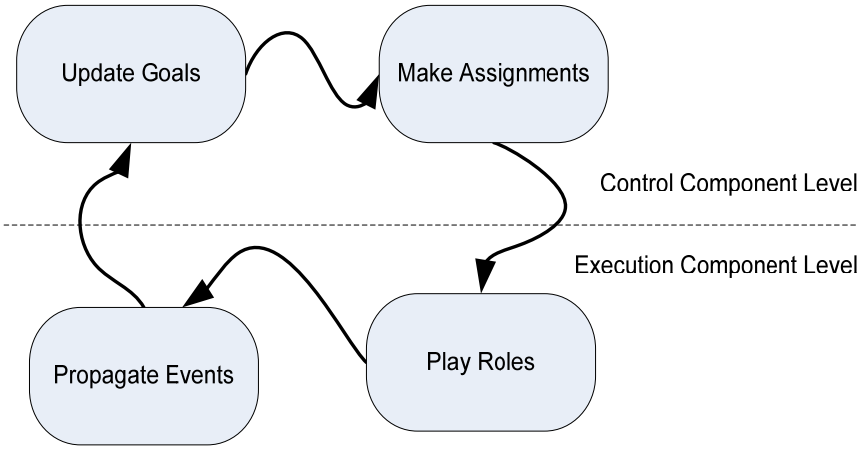

Figure 6.20. Runtime Phases

to coordinate with other Execution Components in order to exchange application data. In our case, the Mote agents report their sensor data to the Base Station agent (acting as a sink) via their Execution Components.

6.2.7 System Implementation

At runtime, the system cycles through four main phases: update goals, make assignments, play roles, propagate events (Figure 6.20**)**. The *update goals* and *make assignment* phase are organization-related phases and are only be performed by agents participating in the organization control, in our case, the Base Station agent. *Play roles* and *propagate events* are application-related phases that concern all agents playing a role in the organization. Once goals are updated in response to organizational events, the Base Station agent needs to assign agents to play roles to achieve the newly added goals. Once an agent has been assigned to play a role, it follows the role's plan in order to achieve its goal and reports all events to the Base Station agent.

To make assignments, the Base Station uses a first-fit greedy algorithm to find an agent capable of playing a role to achieve a goal. In addition, we specify reorganization policies that guide the system when assigning the goals to agents [49]. Those policies typically specify the kind of assignments the system should preferably make or avoid. For instance, we specified

```
function makeAssignments(activeGoalSet)returns assignmentSet

1.  for each goal g in activeGoalSet.unassigned
2.    assignment.goal ← g
3.    do
4.      for each role r in Knowledge.roles
5.        if r.achieves(g) then assignment.role ← r; break
6.      end loop
7.      for each agent a in Knowledge.agents
8.        if a.possess(r.requiredCapabilities)
9.         then assignment.agent ← a; break
10.     end loop
11.   until passPolicies(assignment) //may remove policy
12. assignmentSet.add(assignment)
13. end loop
14. return assignmentSet
```

Figure 6.21. Assignment Algorithm

policies to requiring that agents pursuing a monitor goal should together provide a coverage of 100% of the area of interest. Policies are expressed as conditional statements related to one or more assignments. The policies used in our systems are discussed in Section 6.2.8. The assignment algorithm is shown in Figure 6.21.

For each unassigned goal (line 1), we get the first role that can achieve it (line 5) and the first agent that has all the required capabilities to achieve that role (line 8). The assignment thereby produced is checked for policies compliance in line 11. If it fails, the *passPolicies* method removes the roles and agents that caused the assignment to fail and a new assignment is sought. If no assignment exists, the least important policy is deactivated to ensure that the system can progress. In fact, the policies are guidance policies [49] that aims at guiding the system towards the desired behavior without constraining it. They can be abandoned if they prevent the system from progressing .i.e. no assignment meeting the policy can be found. Multiple policies can be arranged by importance order, allowing the systems to ignore the least important policy first when unable to find an assignment.

The application consists of sensors arranged in a grid. As mentioned before, each Mote agent is represented by an actual sensor and has the required capabilities to play the Monitor role and the Track role. There is also the Base Station agent who possesses all the required

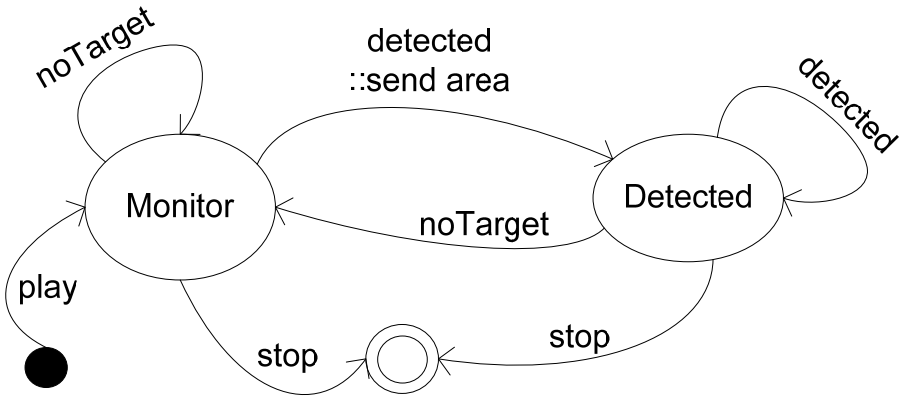

Figure 6.22. Plan for the Monitor role

capabilities to play the User Interface, Coverage Processor, and Divider roles. The Mote agents have been implemented in nesC [45], a component-based programming language that is currently used to program the Berkeley motes [22] running on TinyOS [53] operating system. The Base Station agent is implemented in Java.

The Base Station agent gets the area via a user interface (*User Interface* role) and divides it into subareas such that a unique sensor can cover each subareas (*Coverage Processor* role). In fact, Sensors can only be given areas that fall entirely within their sensing range. This division is made while insuring that the minimum number of sensors cover the entire initial area. Mote agents assigned to the *Monitor* role execute the role's plan. This plan, represented as a finite-state machine, is shown in Figure 6.22**.**

For the *Monitor* role, agents sense the magnetic field at the rate of 1Hz as long as no target is detected (*Monitor* state). Once a target is detected, the agent generates an event to initiate a track over an area equal to twice its sensing radius (*Detected* state). This area has been chosen as a prediction of where the target could soon be located. This event results in the Base Station agent activating the *Track Area* goal (Figure 6.16). The resultant configuration includes the Base Station agent playing the *Divider* role in order to select a set of subareas that need to be tracked. Subsequently, agents capable of sensing the tracking area are assigned to the *Tracking* role for a subarea. The plan for the *Track* role is depicted in Figure 6.23. If the tracking agent

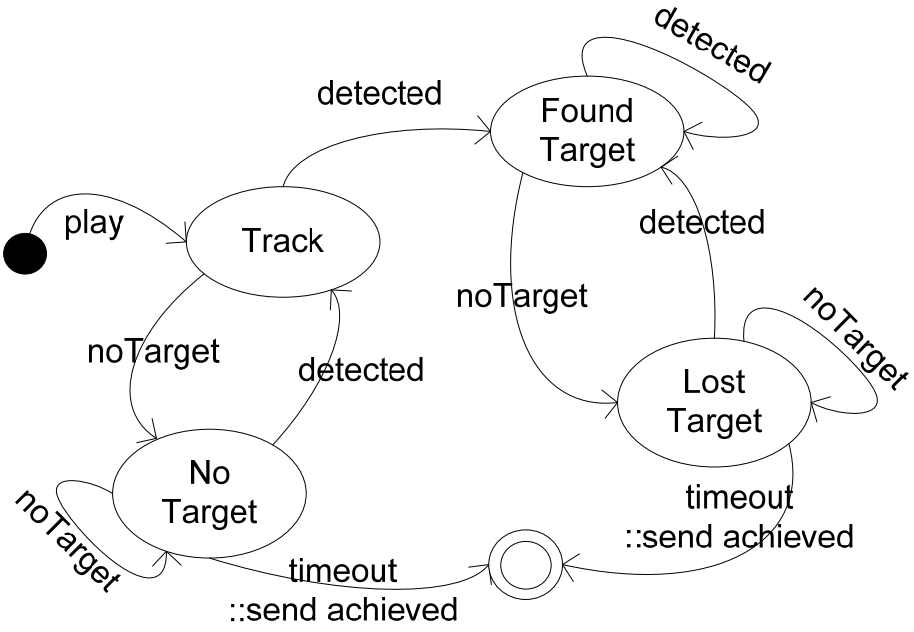

Figure 6.23. Plan for the Tracker role

(i.e. the agent playing the tracking role) is on the target trajectory, it will eventually detect the target (*Found Target* state). Based on the speed of the target, a tracking agent may have to wait for a certain time before detecting the target (*No Target* state). Taking the target speed into consideration, we have set the wait time such that the tracking agent always detects a target moving in its direction. Whenever the target is lost (*Lost Target* state) or the target has never been detected (*No Target* state), the agent sends a message to the Base Station agent indicating that it has achieved its role.

In addition, the agent playing the monitor role and the tracker role are playing the receiver role in order to keep their clock synchronized with the Base Station clock. Hence, for the Receiver roles, agents keep track of the offset between their clock and the clock at the Base Station. This allows them to send their data along with the Base Station time when the data were gathered.

Finally, in order to maintain the sensor topology, sensors periodically send a beacon message indicating that there are still in the network. If a beacon is not received after a certain

time, the Base Station considers that the agent has failed and reassigns any roles that the agent had to other agents. Moreover, in order to reduce the number of messages sent in the application, the beacon messages are also used to report capability status updates (such as power level of the sensors).

6.2.8 Experimental Results

The actual testbed for our experiments consisted of 25 sensors evenly distributed in a 5×5 grid with integer (x, y) coordinates ranging from (0, 0) to (4, 4). The sensors covered an area of 100ft x 100ft. We chose TOSSIM [76] as a simulation environment since it uses nesC and it can emulate the execution of the real code on the motes without the need for deployment. In addition, TOSSIM scripting language Tython [29] allowed us to interact with the simulator environment by inserting failures and simulating moving targets. We simulated the moving target as a magnetometer source that can linearly affect the magnetometer readings of the nearby sensors based on their distance. We set up the detection threshold value such that sensors can detect the moving target for up to 30 ft. We also set the target to move on straight lines at the constant speed of 2ft/s. For all our experiments, we assumed that communication is reliable and that all nodes are within communication range to the Base Station.

Our first set of experiments attempted to see how the systems can recover from sensors failures. When an agent fails while achieving a goal, the organization triggers a reorganization that may result in another agent assigned to the failed goal or in another set of alternative goals that are assigned to the available agents. For these experiments, we were interested in the failure of the monitor agents. Such failures were recognized by the organization when agents failed to send beacon messages. When this happened, the organization tried to find another set of monitoring agents that can insure a total coverage of the surveillance area. Initially, the organization tried to find an agent that can cover the area previously by the failed agent. If it was unsuccessful, all the current monitor agents were deassigned and replaced by a new set of monitor agents capable of covering the area. The adaptation of the system was measured in term of the amount of the surveillance area covered by monitoring agents after various failure ratios. The coverage was computed as the average observed over 5 runs of 1000 simulation seconds. Throughout the simulation, a failure of a random monitor agent was introduced at a constant rate

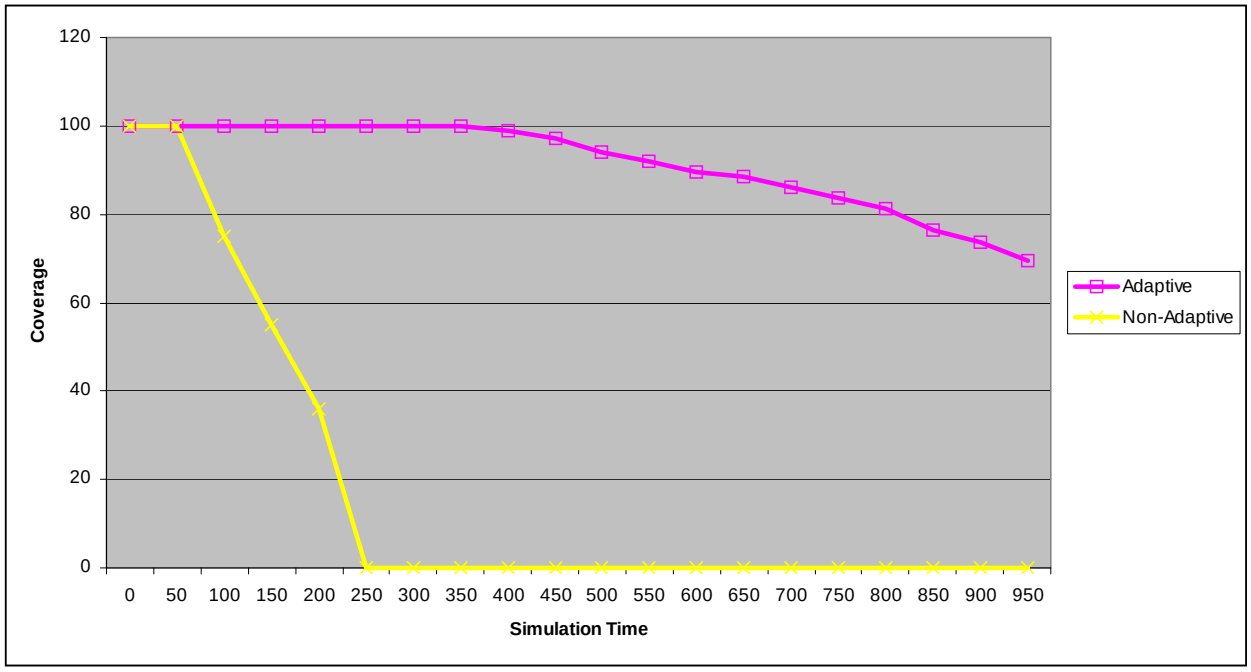

Figure 6.24. Coverage obtained by injecting a monitor failure every 50 seconds

(every 50 seconds) between 100 and 900 seconds. In addition, we measured the coverage obtained in a non-adaptive version of the system for which four agents were statically assigned to the monitor role at design time and were not replaced after a failure.

Figure 6.24 shows the results obtained for our adaptive system and for a non-adaptive version. These results show that even with 6 sensor failures (350 seconds), our systems can still provide 100% coverage, whereas the non-adaptive version cannot cover the area once all its four static sensors have failed (250 seconds). By the end of the simulation, the adaptive system would have lost 70% of its sensors but can still cover more than 65% of the area. These large coverage values with few sensors are due to the fact that on average, each sensor can cover 25% of the area. Nevertheless, the organization was able to reorganized in the face of failures and reassign the failed monitoring goals to available sensors in order to continue to insure a maximum coverage of the surveillance area.

Our second set of experiments was intended to show how the system can adapt to decreasing energy levels. Like most surveillance systems, most of the energy is consumed during the surveillance phase, monitoring for potential targets [50]. Having a non-adaptive design with static monitoring agents leads to the complete energy depletion, while other agents have most of their energy unused. Hence, we were interested in having the system adapt in order to maintain a uniform level of energy among all the nodes while trying to always insure a maximum coverage. For that, we introduced several guidance policies aiming at putting a lower bound on the energy required to play a monitor role. This allowed agents to give up the monitoring role when their energy dropped below a certain threshold. We defined three *energy policies* (EP) and one *coverage policies* (CP) ordered by least-important:

- all monitor agents should have less than 20% of their energy used (EP20)
- all monitor agents should have less than 50% of their energy used (EP50)
- all monitor agents should cover 100% of the surveillance area (CP100)

As explained earlier, the system can satisfy all the policies as long as satisfying assignment can be made. If no assignment can be found, the system deactivates the least important policy among all the active ones in order to proceed. Therefore, with EP20 active, every time a monitor agent's energy used rose above 20%, there was a reorganization aiming at finding another agent to play the monitor role. Whenever there were no agents with less than 20% of their energy used or those with less than 20% of their energy used could not cover 100% of the area, the system deactivated EP20. This process continued until all the policies were deactivated, in which case the system assigned monitoring roles without ensuring maximum coverage.

We compared the energy level of each agent at the end of a simulation with and without the policies. The system running without energy policies kept the same monitor roles throughout the entire simulation, whereas the system with policies behaved as indicated above. We used the number of radio messages sent as a measure of energy consumption, which is reasonable given that radio communication largely dominates energy on the motes [105]. Note that during the monitor or track roles, agents are constantly sending sensor readings and beacon messages back the base station.

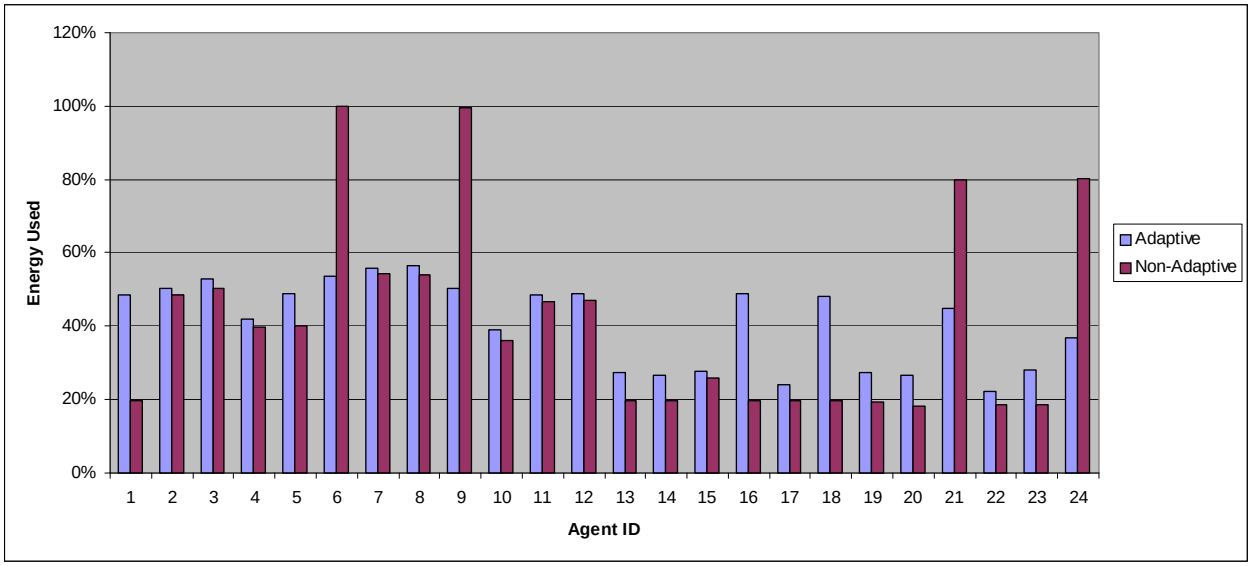

Figure 6.25. Energy Used discrepancies between the adaptive and non-adaptive system

Figure 6.25 shows the energy consumed for both systems. The results were observed over a run of 1000 simulation seconds with 3 targets appearing one at a time along the same path. Globally, there was a target present 20% of the simulation time. Without policy (non-adaptive version), agents 6, 9, 21 and 24 have used more energy than any other agents. This is due to the fact that these agents were playing the monitor role throughout the entire simulation. On the other hand, we observe that the adaptive version helps keep the energy level uniform among all agents. In fact, even though both systems used 41% of their global energy, the standard deviation for the adaptive system was 12% whereas the one for the non-adaptive version was 28%. These results support the fact that our adaptive system was able to reorganize in order to maintain a uniform distribution of the energy used as directed by the policies.

6.3 Summary

In this chapter, I proposed two applications that demonstrate the validity and usefulness of my design approach. The Cooperative Robotic for Airport Management application was

designed to exemplify how services can be iteratively composed to design a larger application. In addition, I proposed an ad-hoc design that uses the notion of sub-organization and compared it with my approach. Moreover, I proposed a compositional design for a Surveillance application. This design was then implemented on a simulator to demonstrate the adaptive properties gained from developing applications based on the models supported by my approach. My implementation demonstrated the following adaptive properties (defined in more detail in [73]):

- *Self-configuration*: The system was able to reconfigure itself when new goals appear in the organization in order to achieve them.
- *Self-healing*: The system was able to reorganize to overcome the loss of a sensor and the loss of performance due to capability degradation.

Therefore, the implementation proposed here exemplified some of the important characteristic of the design models used in this research. It is important to note that the design used for this application, which was obtained using the compositional approach, is not different than any other OMACS-based organizational design. The composed design models are translated into runtime models using the same engine that was previously available for OMACS models. In fact, all the entities in the final composed design are defined by the OMACS metamodel and this design benefits from the same adaptive capabilities as any OMACS-based design. Other implementations using OMACS-based organizational designs can be found in [28] and [89].

CHAPTER 7 - CONCLUSION AND FUTURE WORK

"A whole is that which has beginning, middle and end." — *Aristotle*

""Tut, tut, child" said the Duchess. "Everything has got a moral if you can only find it." "
—*Lewis Carroll, Alice in Wonderland*

In this chapter, I present a summary of the work presented in this dissertation, along with its major contributions. Then I proceed with a discussion of the benefits and drawbacks of the proposed approach along with a brief outline of future directions related to this research.

7.1 Summary

In this dissertation, I presented a formal compositional framework to design composable multiagent organizations. This framework was then used to develop a service-oriented approach for designing adaptive organization-based multiagent systems. I used *category theory* [5] to formalize the composition of organizations. This mathematical framework allowed us to formally represent organization design models and derive their compositions. In the composition framework proposed, the goal models and the role models were composed separately and then the composition for organization designs was derived.

In Chapter 4, I proposed a formalization of two core design models (the goal model and the role model) using categorical concepts and showed a construction to derive the composition of two organizations. This construction guaranteed the correctness of the composed organizations. However, this composition process relies on the definition of configurations, which indicate what elements from the models can be merged. As the composition process is general, there are no constraints on what configurations are allowed and designers can potentially

define composite organization that may not be implementable into a coherent system. For this reason, in Chapter 5, I proposed a service-oriented framework to help specify configurations necessary to compose organizations. The main idea is to express configurations in terms of services required and provided instead of category theory formalisms. In essence, service mappings are an implicit way of defining configurations. In the service-oriented framework, multiagent organizations are viewed as reusable components that use and provide services. I defined and formalized all the entities required to develop these *reusable organizations* and proved that the composition of services can be obtained using the category theoretic composition framework developed in Chapter 4.

Finally, to demonstrate the usefulness and validity of the compositional framework to design real-world applications, I developed two applications with several services (Chapter 6). For the first application, I designed several cooperative robotic services and showed how they can be reused to build other organizations. I gave some examples to show how several services can be composed to create a complex organization design. The second application was a wireless sensor networks application that used one service, the time synchronization service. I provided two different designs of that service and showed how the service-oriented framework allowed the permutation of those two designs without any modifications to the main design. Furthermore, I implemented this application in order to demonstrate some adaptive properties of the system. Through some experiments, I showed how the system can autonomously reconfigure and recover from failures.

Therefore, as part of this research, three main contributions have been made.

1- **I have developed a general algebraic composition framework that formally characterizes the composition of two or more organization design models.**

As discussed in Section 5.4, no other work exists that formally composes organization-based multiagent systems designs. Moreover, the approach followed in this framework serves as a blueprint to formally specify organizational design models as categories. As stated in Section 4.5, there are very few works combining multiagent systems and category theory. Formalizing multiagent design models is necessary to apply various types of model transformations and verifications. In fact, formal models are easily

verified and can be efficiently converted into runtime models, as suggested by Blair et al. [7].

2- I have derived a specific service-oriented composition framework that allows the compositional design of organizations

In general, decomposition is considered as a key property to tackle the growing complexity of software [8]. I showed that by providing an approach for the compositional design of organizations, my framework facilitates the development of complex OMAS. As opposed to most approaches that only use decomposition to benefit separation of concerns during design and do not provide a rigorous composition process, my approach provides a formalization that allows the systematic composition of organizations. I have combined service-oriented principles with organizational concepts in order to provide MAS designers with predefined reusable multiagent organizations. As a result, system designers can easily build complex organizations by using collections of simpler organization modules wrapped up as services.

3- I have demonstrated the usefulness and validity of the proposed framework for developing adaptive organization-based multiagent systems.

This evaluation is an important step in showing that the compositional approach facilitates reuse and increases flexibility of organizational designs. Through two examples, I showed how my approach helps integrate simpler organizational designs into more complex designs. Moreover, I also showed that the design models used in this research help provide adaptive mechanisms at runtime. This adaptive behavior is an indispensable feature in today's complex systems.

7.2 Discussion and Future Work

This section discusses some of the benefits and drawbacks of my approach and proposes how this work can be extended to overcome some of the limitations.

As stated before, the approach proposed in this dissertation allows system designers to incrementally build multiagent organization designs. However, without a systematic

implementation, any design produced out of this process might loose its properties. In Chapter 6, I showed the adaptive behavior resulting from using the design approach advocated in this research. However, even though those properties are available at design time, they can be lost due to a poor implementation of the organizational concepts. Therefore, it is necessary to have a model-driven implementation to guarantee the preservation of the important properties existing in the design models.

Moreover, I believe there is a need for more guidance to help designers develop organizational services as proposed in Chapter 5. In fact, a service may be called multiple times and the designer may run into some performance issues if the service is not designed properly. For instance, consider a cleaning service that creates a map of a room before cleaning it. If the mapping process is expensive in terms of time and resources (agents), it would be wise to insure that this mapping process is not repeated every time there is a service request to clean that room. Hence, if such information is available, we would like to take it into consideration when designing the service. To take care of that problem, I propose the use of service design patterns that would guide designers when designing services. These design patterns would reduce design errors and help increase the performance of the services.

In addition to design patterns, it would also be necessary to provide a Quality of Service (QoS) feature to help designer decide which service provider to choose and to document what QoS they could expect. I believe that an approach similar to the integration of QoS into UML models [25] would be suitable for the framework proposed here.

Finally, the compositional design framework presented in this research needs to be supported by a rigorous methodology and a tool. O-MaSE is an extensible process framework that can be extended by adding new process fragments necessary for the compositional design of multiagent organizations. In addition, agentTool III is a development environment that already supports O-MaSE processes. Hence, this tool would be a good candidate to support compositional design.

References

[1] F. Achermann and O. Nierstrasz, *A calculus for reasoning about software composition.* Theoretical Computer Science, 2005. **331**(2-3): p. 367-396.

[2] C. Atkinson, et al., *Component-based product line engineering with UML*. 2002: Addison-Wesley Longman Publishing Co., Inc. 506.

[3] S. Awodey, *Category theory*. 2006: Oxford University Press, USA.

[4] J. Bansiya and C.G. Davis, *A Hierarchical Model for Object-Oriented Design Quality Assessment.* IEEE Transactions on Software Engineering 2002. **28**(1): p. 4-17.

[5] M. Barr and C. Wells, *Category theory for computing science*. Prentice Hall international series in computer science. 1990, New York: Prentice Hall.

[6] R. Ben-Natan, *Corba: a guide to common object request broker architecture*, ed. B.-N. Ron. 1995: McGraw-Hill, Inc. 353.

[7] G. Blair, N. Bencomo, and R.B. France, *Models@ run.time.* Computer, 2009. **42**(10): p. 22-27.

[8] G. Booch, *Object-oriented analysis and design with applications (2nd ed.)*, ed. A.-W.C. Series. 1994: Benjamin-Cummings Publishing Co., Inc. 608.

[9] A. Boronat, et al., *Formal Model Merging Applied to Class Diagram Integration.* Electronic Notes in Theoretical Computer Science, 2007. **166**: p. 5-26.

[10] M.E. Bratman and P. Intentions, *Practical Reasoning*. 1987: Harvard Univ. Press: Cambridge, MA.

[11] F.M.T. Brazier, et al., *DESIRE: Modelling Multi-Agent Systems in a Compositional Formal Framework.* IJCIS, 1997. **6**(1): p. 67-94.

[12] P. Bresciani, et al., *Tropos: An Agent-Oriented Software Development Methodology.* Autonomous Agents and Multi-Agent Systems, 2004. **8**(3): p. 203-236.

[13] S. Brinkkemper, *Method engineering: engineering of information systems development methods and tools.* Information and Software Technology, 1996. **38**(4): p. 275-280.

[14] R. Brooks, *A robust layered control system for a mobile robot.* Robotics and Automation, IEEE Journal of, 1986. **2**(1): p. 14-23.

[15] G. Brunet, et al., *A manifesto for model merging*, in *Proceedings of the 2006 international workshop on Global integrated model management*. 2006, ACM: Shanghai, China.

[16] P. Buneman, S. Davidson, and A. Kosky, *Theoretical aspects of schema merging*, in *Advances in Database Technology — EDBT '92*. 1992. p. 152-167.

[17] I. Cafezeiro and E.H. Haeusler, *Semantic interoperability via category theory*, in *26th international conference on Conceptual modeling*. 2007: Auckland, New Zealand.

[18] L. Cao, C. Zhang, and M. Zhou, *Engineering Open Complex Agent Systems: A Case Study.* Systems, Man, and Cybernetics, Part C: Applications and Reviews, IEEE Transactions on, 2008. **38**(4): p. 483-496.

[19] K.M. Carley and M.J. Prietula, *Computational Organization Theory*. 1994: Lawrence Erlbaum Associates.

[20] A. Chella, et al., *Agile PASSI: An agile process for designing agents.* International Journal of Computer Systems Science & Engineering, 2006. **21**(2): p. 133-144.

[21] M. Cossentino, et al., *The PASSI and Agile PASSI MAS Meta-models Compared with a Unifying Proposal*, in *Multi-Agent Systems and Applications IV*. 2005. p. 183-192.

[22] Crossbow. *Wireless sensor networks (mica motes).* [cited 2009; Available from: http://www.xbow.com/.

[23] C. Cubillos, S. Gaete, and B. Crawford, *Design of an agent-based system for passenger transportation using PASSI.* Lecture Notes in Computer Science, 2007. **4528**: p. 531.

[24] M. Dastani, F. Arbab, and F.d. Boer, *Coordination and composition in multi-agent systems*, in *Proceedings of the fourth international joint conference on Autonomous agents and multiagent systems*. 2005, ACM: The Netherlands.

[25] M.A. de Miguel. *General framework for the description of QoS in UML*. in Sixth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing. 2003.

[26] S.A. DeLoach, *OMACS: a Framework for Adaptive, Complex Systems*, in *Handbook of Research on Multi-Agent Systems: Semantics and Dynamics of Organizational Models*, V. Dignum, Editor. 2009, IGI Global: Hershey, PA.

[27] S.A. DeLoach and M. Miller, *A Goal Model for Adaptive Complex Systems.* International Journal of Computational Intelligence: Theory and Practice, 2010. **5**(2).

[28] S.A. DeLoach, W.H. Oyenan, and E. Matson, *A capabilities-based model for adaptive organizations.* Autonomous Agents and Multi-Agent Systems, 2008. **16**(1): p. 13-56.

[29] M. Demmer, et al., *Tython: A Dynamic Simulation Environment for Sensor Networks*, U.o.C. Berkeley, Editor. 2005.

[30] V. Dignum, *A model for organizational interaction: based on agents, founded in logic*. 2004, SIKS Dissertation Series.

[31] V. Dignum and F. Dignum, *A landscape of agent systems for the real world*. 2006, Utrecht University.

[32] V. Dignum, J. Vázquez-Salceda, and F. Dignum, *OMNI: Introducing Social Structure, Norms and Ontologies into Agent Organizations*, in *Programming Multi-Agent Systems*. 2005. p. 181-198.

[33] S. Dustdar and W. Schreiner, *A survey on web services composition.* Int. J. Web Grid Serv., 2005. **1**(1): p. 1-30.

[34] A.H. Eden and T. Mens, *Measuring software flexibility.* IEEE Proceedings Software, 2006. **153**(3): p. 113-125.

[35] J. Elson and Kay Römer, *Wireless sensor networks: a new regime for time synchronization.* SIGCOMM Comput. Commun. Rev., 2003. **33**(1): p. 149-154.

[36] A. Estefania, J. Vicente, and B. Vicente, *Multi-Agent System Development Based on Organizations.* Electronic Notes in Theoretical Comp. Sci., 2006. **150**(3): p. 55-71.

[37] J. Ferber, *Multi-Agent Systems: An Introduction to Distributed Artificial Intelligence*. 1999: {Addison-Wesley Professional}.

[38] J. Ferber, et al. *Organization models and behavioural requirements specification for multi-agent systems*. in MultiAgent Systems, 2000. Proceedings. Fourth International Conference on. 2000.

[39] J. Ferber, O. Gutknecht, and F. Michel, *From Agents to Organizations: An Organizational View of Multi-agent Systems*, in *Agent-Oriented Software Engineering IV*. 2004. p. 443-459.

[40] J.L. Fiadeiro, *Categories for software engineering*. 2005: Springer.

[41] M.S. Fox, *An Organizational View of Distributed Systems.* Systems, Man and Cybernetics, IEEE Transactions on, 1981. **11**(1): p. 70-80.

[42] W. Frakes and C. Terry, *Software reuse: metrics and models.* ACM Computing Surveys (CSUR), 1996. **28**(2): p. 415-435.

[43] J.C. Garcia-Ojeda, S.A. DeLoach, and Robby. *agentTool Process Editor: Supporting the Design of Tailored Agent-based Processes*. in Proceedings of the 24th Annual ACM Symposium on Applied Computing 2009. Honolulu, Hawaii.

[44] J.C. Garcia-Ojeda, et al. *O-MaSE: A Customizable Approach to Developing Multiagent Development Processes*. in 8th International Workshop on Agent Oriented Software Engineering 2007.

[45] D. Gay, et al. *The nesC language: A holistic approach to networked embedded systems*. in Programming language design and implementation (PLDI '03). 2003.

[46] J.A. Goguen, *A categorical manifesto.* Mathematical Structures in Computer Science, 1991.

[47] J.L. Gross and J. Yellen, *Graph theory and its applications*. 2006: CRC press.

[48] M. Hannoun, et al., *MOISE: An Organizational Model for Multi-agent Systems*, in *Advances in Artificial Intelligence*. 2000. p. 156-165.

[49] S. Harmon, S. DeLoach, and Robby, *Trace-Based Specification of Law and Guidance Policies for Multi-Agent Systems*, in *ESAW: Engineering Societies in the Agents World VIII*. 2008. p. 333-349.

[50] T. He, et al., *VigilNet: An integrated sensor network system for energy-efficient surveillance.* ACM Trans. Sen. Netw., 2006. **2**(1): p. 1-38.

[51] P. Hell and J. Nešetril, *Graphs and homomorphisms*. 2004: Oxford University Press, USA.

[52] C. Herrmann, et al., *An Algebraic View on the Semantics of Model Composition*, in *Model Driven Architecture- Foundations and Applications*. 2007. p. 99-113.

[53] J. Hill, et al., *System architecture directions for networked sensors.* SIGPLAN Notice, 2000. **35**(11): p. 93-104.

[54] P. Hitzler, et al. *What Is Ontology Merging?* 2005.

[55] B. Horling and V. Lesser, *A survey of multi-agent organizational paradigms.* Knowl. Eng. Rev., 2004. **19**(4): p. 281-316.

[56] J. Hübner, J. Sichman, and O. Boissier. *MOISE+: Towards a Structural, Functional, and Deontic model for MAS organization*.

[57] J.F. Hübner and J.S. Sichman. *Using the MOISE+ model for a cooperative framework of MAS reorganization*. in Proc. 17th Brazilian Symposium on Artificial Intelligence (SBIA 04). 2004. São Luís, Brasil: Advances in Artificial Intelligence.

[58] M.P. Huget, *Representing Goals in Multiagent Systems*, in *Proc. 4th Int'l Symp. Agent Theory to Agent Implementation*. 2004. p. 588–593.

[59] M.P. Huget and J. Odell. *Representing agent interaction protocols with agent UML*. in Autonomous Agents and Multiagent Systems, 2004. AAMAS 2004. Proceedings of the Third International Joint Conference on. 2004.

[60] M.N. Huhns and M.P. Singh, *Service-oriented computing: key concepts and principles.* Internet Computing, IEEE, 2005. **9**(1): p. 75-81.

[61] M.N. Huhns, et al., *Research Directions for Service-Oriented Multiagent Systems.* IEEE Internet Computing, 2005. **9**(6): p. 65-70.

[62] C.A. Iglesias, M. Garijo, and J.C. González, *A Survey of Agent-Oriented Methodologies*, in *Intelligent Agents V. Agent Theories, Architectures, and Languages: 5th International Workshop, ATAL'98, Paris, France, July 1998. Proceedings*. 2000. p. 630-630.

[63] I. Jacobson, G. Booch, and J. Rumbaugh, *The unified software development process*. 1999: Addison-Wesley.

[64] C. Jeanneret, R. France, and B. Baudry, *A reference process for model composition*, in *Proceedings of the 2008 AOSD workshop on Aspect-oriented modeling*. 2008, ACM: Brussels, Belgium.

[65] N.R. Jennings, *An agent-based approach for building complex software systems.* Commun. ACM, 2001. **44**(4): p. 35-41.

[66] N.R. Jennings, *On agent-based software engineering.* Artificial Intelligence, 2000. **117**(2): p. 277-296.

[67] N.R. Jennings, K. Sycara, and M. Wooldridge, *A Roadmap of Agent Research and Development.* Autonomous Agents and Multi-Agent Systems, 1998. **1**(1): p. 7-38.

[68] M.W. Johnson, P. McBurney, and S. Parsons, *A Mathematical Model of Dialog.* Electronic Notes in Theoretical Computer Science, 2005. **141**(5): p. 33-48.

[69] T. Juan, A. Pearce, and L. Sterling, *ROADMAP: extending the gaia methodology for complex open systems*, in *Proceedings of the first international joint conference on Autonomous agents and multiagent systems*. 2002, ACM: Bologna, Italy.

[70] M.B. Juric, *Business Process Execution Language for Web Services BPEL and BPEL4WS 2nd Edition*. 2006: Packt Publishing.

[71] N. Kavantzas, et al., *Web services choreography description language version 1.0.* W3C Working Draft, 2004. **17**: p. 10-20041217.

[72] H.J. Keisler, *Elementary calculus.* Bull. Amer. Math. Soc. 83 (1977), 205-208. DOI: 10.1090/S0002-9904-1977-14264-X PII: S 0002-9904 (1977) 14264-X, 1977.

[73] J.O. Kephart and D.M. Chess, *The vision of autonomic computing.* Computer, 2003. **36**(1): p. 41-50.

[74] M. Kolp, P. Giorgini, and J. Mylopoulos, *Multi-Agent Architectures as Organizational Structures.* Autonomous Agents and Multi-Agent Systems, 2006. **13**(1): p. 3-25.

[75] K.-K. Lau and Z. Wang, *Software Component Models.* IEEE Trans. Softw. Eng., 2007. **33**(10): p. 709-724.

[76] P. Levis, et al. *TOSSIM: accurate and scalable simulation of entire TinyOS applications*. in SenSys '03: Proceedings of the 1st international conference on Embedded networked sensor systems. 2003. Los Angeles, California.

[77] Q. Li and S. Payandeh, *Manipulation of Convex Objects via Two-agent Point-contact Push.* The International Journal of Robotics Research, 2007. **26**(4): p. 377-403.

[78] C. Luo and S.X. Yang. *A real-time cooperative sweeping strategy for multiple cleaning robots*. in Intelligent Control, 2002. Proceedings of the 2002 IEEE International Symposium on. 2002.

[79] S. Mac Lane, *Categories for the working mathematician*. 1998: Springer verlag.

[80] M. Maróti, et al., *The flooding time synchronization protocol*, in *Proceedings of the 2nd international conference on Embedded networked sensor systems*. 2004, ACM: Baltimore, MD, USA.

[81] V. Matena, B. Stearns, and L. Demichiel, *Applying Enterprise JavaBeans: Component-Based Development for the J2EE Platform*. 2003: Pearson Education. 496.

[82] E. Matson and S. DeLoach. *Capability in Organization Based Multi-agent Systems*. in Proceedings of the Intelligent and Computer Systems (IS'03) Conference. 2003.

[83] N. Medvidovic and R.N. Taylor, *A Classification and Comparison Framework for Software Architecture Description Languages.* IEEE Trans. Softw. Eng., 2000. **26**(1): p. 70-93.

[84] M. Morandini, et al., *Tool-Supported Development with Tropos: The Conference Management System Case Study*, in *Agent-Oriented Software Engineering VIII*. 2008. p. 182-196.

[85] S. Nejati, et al., *Matching and Merging of Statecharts Specifications*, in *Proceedings of the 29th international conference on Software Engineering*. 2007, IEEE Computer Society.

[86] N. Niu, S. Easterbrook, and M. Sabetzadeh, *A Category-theoretic Approach to Syntactic Software Merging*, in *Proceedings of the 21st IEEE International Conference on Software Maintenance*. 2005, IEEE Computer Society.

[87] J. Odell, *Objects and agents compared.* Technology, 2002. **1**(1): p. 41-53.

[88] OMG, *{UML 2.0 Superstructure Specification}*. 2005: Framingham, Massachusetts.

[89] W.H. Oyenan and S.A. DeLoach, *Towards a Systematic Approach for Designing Autonomic Systems.* Web Intelligence and Agent Systems (WIAS): An International Journal, 2010. **8**(1).

[90] L. Padgham and M. Winikoff, *Prometheus: A Methodology for Developing Intelligent Agents*, in *Agent-Oriented Software Engineering III*. 2003. p. 174-185.

[91] L.E. Parker, *ALLIANCE: an architecture for fault tolerant multirobot cooperation.* Robotics and Automation, IEEE Transactions on, 1998. **14**(2): p. 220-240.

[92] L.E. Parker, *Current state of the art in distributed autonomous mobile robotics*. Tokyo, Japan ed. Distributed Autonomous Robotic Systems, ed. L.E. Parker, G. Bekey, and J. Barhen. Vol. 4. 2000: Springer-Verlag.

[93] J. Pavón and J. Gómez-Sanz, *Agent Oriented Software Engineering with INGENIAS*, in *Multi-Agent Systems and Applications III*. 2003. p. 1069-1069.

[94] C. Peltz, *Web Services Orchestration and Choreography.* Computer, 2003. **36**(10): p. 46 - 52.

[95] R.A. Pottinger and P.A. Bernstein, *Merging models based on given correspondences*, in *Proceedings of the 29th international conference on Very large data bases - Volume 29*. 2003, VLDB Endowment: Berlin, Germany.

[96] A. Rao and M. Georgeff. *Modeling Rational Agents within a BDI-Architecture*. in Proceedings of the 2nd International Conference on Principles of Knowledge

Representation and Reasoning (KR'91). 1991: Morgan Kaufmann publishers Inc.: San Mateo, CA, USA.

[97] D. Richards, *Merging individual conceptual models of requirements.* Requirements Engineering, 2003. **8**(4): p. 195-205.

[98] D. Rogerson, *Inside COM*. 1997: Microsoft Press. 416.

[99] D. Rus, B. Donald, and J. Jennings. *Moving furniture with teams of autonomous robots*. in Intelligent Robots and Systems 95. 'Human Robot Interaction and Cooperative Robots', Proceedings. 1995 IEEE/RSJ International Conference on. 1995.

[100] S.J. Russel and P. Norvig, *Artificial intelligence*. 2003: Prentice-Hall.

[101] D.E. Rydeheard and R.M. Burstall, *Computational category theory*. Prentice Hall International (UK) Ltd. Hertfordshire, UK, UK. 1988: Prentice Hall, 1988. 257.

[102] M. Sabetzadeh and S. Easterbrook. *An algebraic framework for merging incomplete and inconsistent views*. in Requirements Engineering, 2005. Proceedings. 13th IEEE International Conference on. 2005.

[103] J. Sametinger, *Software engineering with reusable components. 1997.* Berlin, New York: Springer. **16**: p. 272.

[104] Scott J. Harmon, et al., *Leveraging Organizational Guidance Policies with Learning to Self-Tune Multiagent Systems*, in *The Second IEEE International Conference on Self-Adaptive and Self-Organizing Systems*. 2008: Venice, Italy.

[105] V. Shnayder, et al. *PowerTOSSIM: Efficient Power Simulation for TinyOS Applications*. in ACM Conference on Embedded Networked Sensor Systems (SenSys). 2004.

[106] J.S.o. Sichman, M.V. Dignum, and C. Castelfranchi, *Agents Organizations: A Concise Overview.* Journal of the Brazilian Computer Society, 2005. **11 (1)**: p. 3-8.

[107] M. Sims, D. Corkill, and V. Lesser, *Automated organization design for multi-agent systems.* Autonomous Agents and Multi-Agent Systems, 2008. **16**(2): p. 151-185.

[108] T. Soboll, *On the Construction of Transformation Steps in the Category of Multiagent Systems*, in *Intelligent Computer Mathematics*. p. 184-190.

[109] D.J. Stilwell and J.S. Bay. *Toward the development of a material transport system using swarms of ant-like robots*. in Robotics and Automation, 1993. Proceedings., 1993 IEEE International Conference on. 1993.

[110] H. Subramanian and C.H. Dagli. *Cooperative Cleaning for distributed Autonomous robot systems using Fuzzy Cognitive Maps*. in Fuzzy Information Processing Society, 2003. NAFIPS 2003. 22nd International Conference of the North American. 2003.

[111] K.P. Sycara, *Multiagent systems.* AI magazine, 1998. **19**(2): p. 79-92.

[112] J. Thangarajah, L. Padgham, and M. Winikoff, *Prometheus design tool*, in *Proceedings of the fourth international joint conference on Autonomous agents and multiagent systems*. 2005, ACM: The Netherlands.

[113] A. van Lamsweerde, et al., *The KAOS Project: Knowledge Acquisition in Automated Specification of Software.* Proceedings AAAI Spring Symposium Series, 1991: p. 59-62.

[114] J. Vázquez-Salceda, *The role of norms and electronic institutions in multi-agent systems applied to complex domains. The HARMONIA framework.* AI Communications, 2003. **16**(3): p. 209-212.

[115] I.A. Wagner, et al., *Cooperative Cleaners: A Study in Ant Robotics.* Int. J. Rob. Res., 2008. **27**(1): p. 127-151.

[116] Z.-D. Wang, E. Nakano, and T. Matsukawa. *Cooperating multiple behavior-based robots for object manipulation*. in Intelligent Robots and Systems '94. 'Advanced Robotic Systems and the Real World', IROS '94. Proceedings of the IEEE/RSJ/GI International Conference on. 1994.

[117] G. Weiss, *Multiagent systems: a modern approach to distributed artificial intelligence*. 2000: The MIT press.

[118] M. Wester-Ebbinghaus, et al. *Towards Organization–Oriented Software Engineering*. in Software Engineering. 2007. Hamburg.

[119] M.F. Wood and S.A. DeLoach, *An overview of the multiagent systems engineering methodology*, in *First international workshop, AOSE 2000 on Agent-oriented software engineering*. 2001, Springer-Verlag New York, Inc.: Limerick, Ireland.

[120] M. Wooldridge, *Introduction to Multiagent Systems*. 2001: John Wiley & Sons, Inc. 376.

[121] M. Wooldridge and P. Ciancarini, *Agent-Oriented Software Engineering: The State of the Art*, in *Agent-Oriented Software Engineering*. 2001. p. 55-82.

[122] M. Wooldridge and N. Jennings, *Intelligent Agents: Theory and Practice.* Knowledge Engineering Review, 1995. **10**(2): p. 115-152.

[123] M. Wooldridge, N.R. Jennings, and D. Kinny, *The Gaia Methodology for Agent-Oriented Analysis and Design.* Autonomous Agents and Multi-Agent Systems, 2000. **3**(3): p. 285-312.

[124] E.S.K. Yu, *Modelling strategic relationships for process reengineering*. 1995, PhD Thesis, University of Toronto: Ontario, Canada.

[125] F. Zambonelli, N.R. Jennings, and M. Wooldridge, *Developing multiagent systems: The Gaia methodology.* ACM Trans. Softw. Eng. Methodol., 2003. **12**(3): p. 317-370.

[126] C. Zhong and S.A. DeLoach, *An Investigation of Reorganization Algorithms*, in *International Conference on Artificial Intelligence (IC-AI'2006)*. 2006, CSREA Press: Las Vegas, Nevada.

Appendix A - Detailed mappings for the compositions

A.1 Composition details for the Search and Rescue Application

This section presents the composition details for the Search and Rescue application presented in Example 5.6.

From Figure 5.11, Φ_1 is an organization homomorphism. We have:

Φ_1: $O_0 \rightarrow$ Search such that $\Phi_1 = \langle f_1, g_1, h_1, i_1, j_1, k_1 \rangle$, where:

- $f_1 = \{\langle g\_root, g\_root\rangle, \langle \mathbf{g_1, Explore}\rangle, \langle \mathbf{g_2, gx_1}\rangle\}$;
- $g_1 = \varnothing$;
- $h_1 = \{\langle |g_1, g_2|, |Explore, gx_1|\rangle\}$;
- $i_1 = \{\langle \mathbf{r_1, Searcher}\rangle, \langle \mathbf{r_2, rx_1}\rangle\}$;
- $j_1 = \{\langle |r_1, r_2|, |Searcher, rx_1|\rangle\}$;
- $k_1 = \varnothing$;

From Figure 5.11, Φ_2 is an organization homomorphism. We have:

Φ_2: $O_0 \rightarrow$ Rescue such that $\Phi_2 = \langle f_2, g_2, h_2, i_2, j_2, k_2 \rangle$ where:

- f2 = $\{\langle g\_root, g\_root\rangle, \langle \mathbf{g_1, gx_2}\rangle, \langle \mathbf{g_2, ID\ \ Victim}\rangle\}$;
- g2 = $\varnothing$;
- h2 = $\{\langle |g_1, g_2|, |gx_2, ID\ Victim|\rangle\}$;
- i2 = $\{\langle r_1, rx_2\rangle, \langle r_2, Identifier\rangle\}$;
- j2 = $\{\langle |r_1, r_2|, |rx_2, Identifier|\rangle\}$;
- k2 = $\varnothing$;

From Figure 5.11, Φ_1' is an organization homomorphism. We have:

Φ_1': Search $\rightarrow$ S&R such that $\Phi_1' = \langle f_1', g_1', h_1', i_1', j_1', k_1' \rangle$ where:

- f_1' = {⟨g_root, g_root⟩, ⟨Search, Search⟩, ⟨Explore, Explore⟩, ⟨Divide Area, Divide Area⟩, ⟨**gx_1, ID Victim**⟩};
- g_1' = {⟨|g_root, Search| , |g_root, Search|⟩, ⟨|Search, Explore|, |Search, Explore|⟩, ⟨|Search, Divide Area|, |Search, Divide Area|⟩}
- h_1' = {⟨ |**Explore, gx_1**|, |**Explore, ID Victim**|⟩, ⟨newArea, newArea⟩};
- i_1'= {⟨Searcher, Searcher⟩, ⟨Divider, Divider⟩, ⟨**rx_1, Identifier**⟩};
- j_1' = {⟨|searcher, rx_1|, |Searcher, Identifier|⟩};
- k_1' = {⟨|Searcher, Explore|, |Searcher, Explore|⟩, ⟨|Divider, Divide Area|, |Divider, Divide Area|⟩}

From Figure 5.11, Φ_2' is an organization homomorphism. We have:

Φ_2': Rescue $\rightarrow$ S&R such that Φ_2' = ⟨ f_2', g_2', h_2', i_2', j_2', k_2' ⟩ where:

- f_2' = {⟨g_root, g_root⟩, ⟨Rescue, Rescue⟩, ⟨ID Victim, ID Victim⟩, ⟨Pickup Victim, Pickup Victim⟩, ⟨**gx_2**, **Explore**⟩}
- g_2' = {⟨|g_root, Rescue|, |g_root, Rescue|⟩, ⟨|Rescue, ID Victim|, |Rescue, ID Victim|⟩, ⟨|Rescue Pickup, Victim|, |Rescue Pickup, Victim|⟩}
- h_2' = {⟨|gx_2, ID Victim|, |Explore, ID Victim|⟩, ⟨precedes, precedes⟩}
- i_2'= {⟨Identifier, Identifier⟩, ⟨Rescuer, Rescuer⟩, ⟨**rx_2, Searcher**⟩}
- j_2' = {⟨|**rx_2, Identifier|, |Searcher, Identifier|**⟩}
- k_2' = {⟨|Identifier, ID Victim|, |Identifier, ID Victim|⟩, ⟨|Rescuer, Pickup Victim|, |Rescuer, Pickup Victim|⟩}

S&R along with homomorphism Φ_1' and Φ_2' (Figure 5.11) represent the pushout of organization O_0 with homomorphism Φ_1 and Φ_2. In fact, we have:

- f_1' $\circ$ f_1 = {⟨g_root, g_root⟩, ⟨**g_1, Explore**⟩, ⟨**g_2, ID Victim**⟩}
- f_2' $\circ$ f_2 = {⟨g_root, g_root⟩, ⟨**g_1, Explore**⟩, ⟨**g_2, ID Victim**⟩}
- g_1' $\circ$ g_1 = $\varnothing$;
- g_2' $\circ$ g_2 = $\varnothing$;
- h_1' $\circ$ h_1 = {⟨ |g_1, g_2|, |Explore, ID Victim|⟩};

- $h_2' \circ h_2 = \{\langle |g_1, g_2|, |\text{Explore, ID Victim}|\rangle\}$;
- $i_1' \circ i_1 = \{\langle \mathbf{r_1, Searcher}\rangle, \langle \mathbf{r_2, Identifier}\rangle\}$;
- $i_2' \circ i_2 = \{\langle \mathbf{r_1, Searcher}\rangle, \langle \mathbf{r_2, Identifier}\rangle\}$;
- $j_1' \circ j_1 = \{\langle |r_1, r_2|, |\text{Searcher, Identifier}|\rangle\}$;
- $j_1' \circ j_1 = \{\langle |r_1, r_2|, |\text{Searcher, Identifier}|\rangle\}$;
- $k_1' \circ k_1 = \varnothing$;
- $k_2' \circ k_2 = \varnothing$;

Hence, we can see that $f_1' \circ f_1 = f_2' \circ f_2$, $g_1' \circ g_1 = g_2' \circ g_2$, $h_1' \circ h_1 = i_2' \circ i_2$, $j_1' \circ j_1 = k_2' \circ k_2$. As $\Phi_1' \circ \Phi_1 = \langle f_1' \circ f_1, g_1' \circ g_1, h_1' \circ h_1, i_1' \circ i_1, j_1' \circ j_1, k_1' \circ k_1\rangle$ and $\Phi_2' \circ \Phi_2 = \langle f_2' \circ f_2, g_2' \circ g_2, h_2' \circ h_2, i_2' \circ i_2, j_2' \circ j_2, k_2' \circ k_2\rangle$, we have $\mathbf{\Phi_1' \circ \Phi_1 = \Phi_2' \circ \Phi_2}$.

□

A.2 Composition details for the CRAM application

This section presents the composition details for the CRAM application presented in Section 6.1.5.

Organization Homomorphisms for constructing Cram_carry$_1$ from composition (C1):

From Figure 6.6, Φ_1 is an organization homomorphism. We have:

Φ_1: Shared$_1$ $\rightarrow$ Cram such that $\Phi_1 = \langle f_1, g_1, h_1, i_1, j_1, k_1 \rangle$, where:

- $f_1 = \{\langle g_0, g\_root\rangle, \langle \mathbf{g_1, Dispose\ Trash}\rangle, \langle \mathbf{g_2, gx_4}\rangle\}$;
- $g_1 = \varnothing$;
- $h_1 = \{\langle |g_1, g_2|, |Dispose\ trash, gx_4|\rangle\}$;
- $i_1 = \{\langle \mathbf{r_1, Trash\ Collector}\rangle, \langle \mathbf{r_2, rx_4}\rangle\}$;
- $j_1 = \{\langle |r_1, r_2|, |Trash\ Collector, rx_4|\ \rangle\}$;
- $k_1 = \varnothing$;

From Figure 6.6, Φ_2 is an organization homomorphism. We have:

Φ_2: Shared$_1$ $\rightarrow$ Transportation such that $\Phi_2 = \langle f_2, g_2, h_2, i_2, j_2, k_2 \rangle$ where:

- $f2 = \{\langle g_0, g\_root\ \rangle, \langle \mathbf{g_1, gx_6}\rangle, \langle \mathbf{g_2, Carry}\ \rangle\}$;
- $g2 = \varnothing$;
- $h2 = \{\langle |g_1, g_2|, |gx_6, Carry|\rangle\}$;
- $i2 = \{\langle r_1, rx_6\rangle, \langle r_2, Carrier\ \rangle\}$;
- $j2 = \{\langle |r_1, r_2|, |rx_6, Carrier|\ \rangle\}$;
- $k2 = \varnothing$;

From Figure 6.6, Φ_1' is an organization homomorphism. We have:

Φ_1': Cram $\rightarrow$ Cram_carry$_1$ such that $\Phi_1' = \langle f_1', g_1', h_1', i_1', j_1', k_1' \rangle$ where:

- f_1' = {⟨ g_root, g_root ⟩, ⟨ Manage Airport, Manage Airport ⟩, ⟨ Operate Sanitary Maintenance, Operate Sanitary Maintenance ⟩, ⟨ Clean Floors, Clean Floors ⟩, ⟨ Dispose Trash, Dispose Trash ⟩, ⟨**gx$_3$, gx$_3$**⟩, ⟨ **gx$_4$, Carry** ⟩, ⟨Monitor Buildings, Monitor Buildings ⟩, ⟨Compute Paths, Compute Paths ⟩, ⟨Patrol, Patrol⟩, ⟨Alert and

Neutralize, Alert and Neutralize ⟩, ⟨**gx_1, gx_1**⟩, ⟨Perform Cargo Inspection, Perform Cargo Inspection,⟩, ⟨Screen All Cargos, Screen All Cargos ⟩, ⟨Send for Inspection, Send for Inspection⟩, ⟨**gx_2, gx_2**⟩};

- g_1' = {⟨|g_root, Manage Airport| , |g_root, Manage Airport|⟩, ⟨|Manage Airport, Operate Sanitary Maintenance|, |Manage Airport, Operate Sanitary Maintenance|⟩, ⟨|Operate Sanitary Maintenance, Clean Floors|, |Operate Sanitary Maintenance, Clean Floors|⟩, ⟨|Operate Sanitary Maintenance, Dispose Trash|, |Operate Sanitary Maintenance, Dispose Trash|⟩, ⟨|Manage Airport, Monitor Buildings|, |Manage Airport, Monitor Buildings|⟩, ⟨|Monitor Buildings, Compute Paths|, |Monitor Buildings, Compute Paths|⟩, ⟨|Monitor Buildings, Patrol|, |Monitor Buildings, Patrol|⟩, ⟨|Monitor Buildings, Alert and Neutralize|, |Monitor Buildings, Alert and Neutralize|⟩, ⟨|Manage Airport, Perform Cargo Inspection|, | Perform Cargo Inspection, Screen All Cargos|⟩, ⟨| Perform Cargo Inspection, Send for Inspection |⟩}
- h_1' = {⟨ |**Clean Floors, gx_3**|, |**Clean Floors, gx_3**| ⟩, ⟨ |**Dispose trash, gx_4**|, |**Dispose trash, Carry**|⟩, ⟨|**Send for Inspection, gx_2**|, |**Send for Inspection, gx_2**|⟩, ⟨|**Alert and Neutralize, gx_1**|, |**Alert and Neutralize, gx_1**|⟩, ⟨found, found⟩, ⟨assign, assign⟩};
- i_1'= {⟨Janitor, Janitor⟩, ⟨ Trash Collector, Trash Collector ⟩, ⟨**rx_3, rx_3**⟩, ⟨**rx_4, Carrier**⟩, ⟨Computer, Computer⟩, ⟨Patroller, Patroller⟩, ⟨Neutralizer, Neutralizer⟩, ⟨**rx1, rx_1**⟩, ⟨Screener, Screener⟩, ⟨Transporter, Transporter⟩, ⟨**rx_2, rx_2**⟩ };
- j_1' = {⟨|Janitor, rx_3|, |Janitor, rx_3|⟩, ⟨|Trash Collector, rx_4|, |Trash Collector, Carrier|⟩, ⟨|Neutralizer, rx_1|, |Neutralizer, rx_1|⟩, ⟨|transporter, rx_2|, |Transporter, rx_2|⟩};
- k_1' = {⟨|Janitor, Clean Floors|, |Janitor, Clean Floors|⟩, ⟨|Trash Collector, Dispose Trash|, |Trash Collector, Dispose Trash|⟩, ⟨|Computer, Compute Paths|, |Computer, Compute Paths|⟩, ⟨|Patroller, Patrol|⟩, ⟨|Patroller, Patrol|⟩, ⟨|Neutralizer, Alert and Neutralize|, |Neutralizer, Alert and Neutralize|⟩, ⟨Screener, Screen All Cargos⟩, ⟨Transporter, Send for Inspection⟩}

From Figure 6.6, Φ_2' is an organization homomorphism. We have:

Φ_2': Transportation → Cram_carry$_1$ such that Φ_2' = ⟨ f_2', g_2', h_2', i_2', j_2', k_2' ⟩ where:

- f_2' = {⟨g_root, g_root⟩, ⟨Transport Object, Transport Object⟩, ⟨Carry, Carry⟩, ⟨Deliver, Deliver⟩, ⟨Load and Unload, Load and Unload⟩, ⟨**gx_6**, **Dispose Trash**⟩, ⟨Push, Push⟩, ⟨Start Pushing, Start Pushing⟩, ⟨ Assist Pushing, Assist Pushing⟩, ⟨**gx_5**, **gx_5**⟩ }
- g_2' = {⟨|g_root, Transport Object|, |g_root, Transport Object|⟩, ⟨|Transport Object, Carry|, |Transport Object, Carry|⟩, ⟨|Carry, Deliver|, |Carry, Deliver|⟩, ⟨|Carry, Load and Unload|, |Carry, Load and Unload |⟩, ⟨|Transport Object, Push|, |Transport Object, Push|⟩, ⟨|Push, Start Pushing|, |Push, Start Pushing|⟩, ⟨|Push, Assist Pushing|⟩, ⟨|Push, Assist Pushing|⟩}
- h_2' = {⟨|gx_6, Carry|, |Dispose Trash, Carry|⟩, ⟨|gx_5, Push|, |gx_5, Push|⟩}
- i_2'= {⟨Carrier, Carrier ⟩, ⟨Lifter, Lifter ⟩, ⟨**rx_6, Trash Collector**⟩, ⟨Pusher, Pusher⟩, ⟨Helper, Helper⟩, ⟨**rx_5, rx_5**⟩}
- j_2' = {⟨ready, ready⟩, ⟨|**rx_6, Carrier|**, |**Trash Collector, Carrier|**⟩, ⟨sync, sync⟩, ⟨|**rx_5, Pusher|**, |**rx_5, Pusher|**⟩}
- k_2' = {⟨|Carrier, Deliver|, |Carrier, Deliver|⟩, ⟨|Lifter, Load and Unload |, |Lifter, Load and Unload |⟩, ⟨|Pusher, Start Pushing|, |Pusher, Start Pushing|⟩, ⟨|Helper, Assist Pushing|, |Helper, Assist Pushing|⟩}

Cram_carry$_1$ along with homomorphism Φ_1' and Φ_2' (Figure 6.6) represent the pushout of organization Shared$_1$ with homomorphism Φ_1 and Φ_2 . In fact, we have:

- f_1' ○ f_1 = {⟨g_0, g_root⟩, ⟨**g_1, Dispose Trash**⟩, ⟨**g_2, Carry**⟩}
- f_2' ○ f_2 = {⟨g_0, g_root⟩, ⟨**g_1, Dispose Trash**⟩, ⟨**g_2, Carry**⟩}
- g_1' ○ g_1 = ∅;
- g_2' ○ g_2 = ∅;
- h_1' ○ h_1 = {⟨ |g_1, g_2|, |Dispose trash, Carry|⟩};
- h_2' ○ h_2 = {⟨ |g_1, g_2|, |Dispose trash, Carry|⟩};
- i_1' ○ i_1 = { ⟨ **r_1, Trash Collector**⟩, ⟨ **r_2, Carrier**⟩};
- i_2' ○ i_2 = { ⟨ **r_1, Trash Collector**⟩, ⟨ **r_2, Carrier**⟩};
- j_1' ○ j_1 = {⟨ |r_1, r_2|, |Trash Collector, Carrier| ⟩};
- j_2' ○ j_2 = {⟨ |r_1, r_2|, |Trash Collector, Carrier| ⟩};

- $k_1' \circ k_1 = \varnothing$;
- $k_2' \circ k_2 = \varnothing$;

Hence, we can see that $f_1' \circ f_1 = f_2' \circ f_2$, $g_1' \circ g_1 = g_2' \circ g_2$, $h_1' \circ h_1 = i_2' \circ i_2$, $j_1' \circ j_1 = k_2' \circ k_2$. As $\Phi_1' \circ \Phi_1 = \langle f_1' \circ f_1, g_1' \circ g_1, h_1' \circ h_1, i_1' \circ i_1, j_1' \circ j_1, k_1' \circ k_1 \rangle$ and $\Phi_2' \circ \Phi_2 = \langle f_2' \circ f_2, g_2' \circ g_2, h_2' \circ h_2, i_2' \circ i_2, j_2' \circ j_2, k_2' \circ k_2 \rangle$, we have $\mathbf{\Phi_1' \circ \Phi_1 = \Phi_2' \circ \Phi_2}$.

Organization Homomorphisms for constructing Cram_carry$_2$ from composition (C2):

From Figure 6.7, Φ_1 is an organization homomorphism. We have:

Φ_1: Shared$_2 \rightarrow$ Cram_carry$_1$ such that $\Phi_1 = \langle f_1, g_1, h_1, i_1, j_1, k_1 \rangle$, where:

- f_1 = {⟨g_0, g_root⟩, ⟨g_1, Transport Object⟩, ⟨g_2, Carry⟩, ⟨g_3, Deliver⟩, ⟨g_4, Load and unload⟩, ⟨g_5, Alert and Neutralize⟩, ⟨**g_6, gx_1**⟩, ⟨g_7, Push⟩, ⟨g_8, Start Pushing⟩, ⟨g_9, Assist Pushing⟩, ⟨**g_{10}, gx_5**⟩};
- g_1 = {⟨|g_0,g_1 |, | g_root, Transport Object |⟩, ⟨|g_1,g_2 |, |Transport Object, Carry |⟩, ⟨| g_2,g_3|, |Carry, Deliver |⟩, ⟨|g_2,g_4 |, |Carry, Load and Unload |⟩, ⟨|g_1,g_7 |, |Transport Object, Push|⟩, ⟨|g_7,g_8 |, |Push, Start Pushing |⟩, ⟨|g_7,g_9 |, |Push, Assist Pushing|⟩};
- h_1 = {⟨|g_5, g_6|, |Alert and Neutralize, gx_1|⟩, ⟨|g_{10}, g_7|, |gx_5, Push|⟩};
- i_1= {⟨r_1, Carrier⟩, ⟨r_2, Lifter⟩, ⟨r_3, Neutralizer⟩, ⟨**r_4, rx_1**⟩, ⟨r_5, Pusher⟩, ⟨r_6, Helper⟩, ⟨**r_7, rx_5**⟩};
- j_1 = {⟨|r_1, r_2|, |Carrier, Lifter|⟩, ⟨|r_3, r_4|, |**Neutralizer, rx_1**|⟩, ⟨|r_5, r_6|, |Pusher, Helper|⟩, ⟨|r_7, r_5|, |**rx_5, Pusher**|⟩};
- k_1 = {⟨|r_1,g_3|, |Carrier, Deliver|⟩, ⟨|r_2, g_4|, |Lifter, Load and Unload|⟩, ⟨|r_5, g_8|, |Pusher, Start Pushing|⟩, ⟨|r_6, g_9|, |helper, Assist Pushing|⟩};

From Figure 6.7, Φ_2 is an organization homomorphism. We have:

Φ_2: Shared$_2 \rightarrow$ Transportation such that $\Phi_2 = \langle f_2, g_2, h_2, i_2, j_2, k_2 \rangle$ where:

- f_2 = {⟨g_0, g_root⟩, ⟨g_1, Transport Object⟩, ⟨g_2, Carry⟩, ⟨g_3, Deliver⟩, ⟨g_4, Load and unload⟩, ⟨**g_5, gx_6**⟩, ⟨g_6, Carry⟩, ⟨g_7, Push⟩, ⟨g_8, Start Pushing⟩, ⟨g_9, Assist Pushing⟩, ⟨**g_{10}, gx_5**⟩};

- g_2 = {⟨|g_0,g_1|, |g_root, Transport Object|⟩, ⟨|g_1,g_2|, |Transport Object, Carry|⟩, ⟨|g_2,g_3|, |Carry, Deliver |⟩, ⟨|g_2,g_4 |, |Carry, Load and Unload |⟩, ⟨|g_1,g_7 |, |Transport Object, Push|⟩, ⟨|g_7,g_8 |, |Push, Start Pushing |⟩, ⟨|g_7,g_9 |, |Push, Assist Pushing|⟩};
- h_2 = {⟨|g_5, g_6|, | gx_6, Carry|⟩, ⟨|g_{10}, g_7|, |gx_5, Push|⟩};
- i_2 = {⟨r_1, Carrier⟩, ⟨r_2, Lifter⟩, ⟨**r_3, rx_6**⟩, ⟨r_4, Carrier⟩, ⟨r_5, Pusher⟩, ⟨r_6, Helper⟩, ⟨**r_7, rx_5**⟩};
- j_2 = {⟨|r_1, r_2|, |Carrier, Lifter|⟩, ⟨|r_3, r_4|, | **rx_6, Carrier**|⟩, ⟨|r_5, r_6|, |Pusher, Helper|⟩, ⟨|r_7, r_5|, |**rx_5, Pusher**|⟩};
- k_2 = {⟨|r_1,g_3|, |Carrier, Deliver|⟩, ⟨|r_2, g_4|, |Lifter, Load and Unload|⟩, ⟨|r_5, g_8|, |Pusher, Start Pushing|⟩, ⟨|r_6, g_9|, |helper, Assist Pushing|⟩};

From Figure 6.7, Φ_1' is an organization homomorphism. We have:

Φ_1': Cram_carry$_1$ → Cram_carry$_2$ such that Φ_1' = ⟨ f_1', g_1', h_1', i_1', j_1', k_1' ⟩ where:

- f_1' = {⟨g_root, g_root⟩, ⟨Transport Object, Transport Object⟩, ⟨Carry, Carry⟩, ⟨Deliver, Deliver⟩, ⟨Load and Unload, Load and Unload⟩, ⟨Push, Push⟩, ⟨Start Pushing, Start Pushing⟩, ⟨Assist Pushing, Assist Pushing⟩, ⟨**gx_5, gx_5**⟩, ⟨Manage Airport, Manage Airport⟩, ⟨Operate Sanitary Maintenance, Operate Sanitary Maintenance⟩, ⟨Clean Floors, Clean Floors⟩, ⟨Dispose Trash, Dispose Trash⟩, ⟨**gx_3, gx_3**⟩, ⟨Monitor Buildings, Monitor Buildings⟩, ⟨Compute Paths, Compute Paths⟩, ⟨Patrol, Patrol⟩, ⟨Alert and Neutralize, Alert and Neutralize ⟩, ⟨**gx_1, carry**⟩, ⟨Perform Cargo Inspection, Perform Cargo Inspection,⟩, ⟨Screen All Cargos, Screen All Cargos ⟩, ⟨Send for Inspection, Send for Inspection⟩, ⟨**gx_2, gx_2**⟩}
- g_1' = {⟨|g_root, Transport Object|, |g_root, Transport Object |⟩, ⟨|Transport Object, Carry|, | Transport Object, Carry|⟩, ⟨|Carry, Deliver|, | Carry, Deliver|⟩, ⟨|Carry, Load and Unload |, | Carry, Load and Unload |⟩, ⟨|g_root, Manage Airport| , |g_root, Manage Airport|⟩, ⟨|Manage Airport, Operate Sanitary Maintenance|, |Manage Airport, Operate Sanitary Maintenance|⟩, ⟨|Operate Sanitary Maintenance, Clean Floors|, |Operate Sanitary Maintenance, Clean Floors|⟩, ⟨|Operate Sanitary Maintenance, Dispose Trash|, |Operate Sanitary Maintenance, Dispose Trash|⟩,

⟨|Manage Airport, Monitor Buildings|, |Manage Airport, Monitor Buildings|⟩, ⟨|Monitor Buildings, Compute Paths|, |Monitor Buildings, Compute Paths|⟩, ⟨|Monitor Buildings, Patrol|, |Monitor Buildings, Patrol|⟩, ⟨|Monitor Buildings, Alert and Neutralize|, |Monitor Buildings, Alert and Neutralize|⟩, ⟨|Manage Airport, Perform Cargo Inspection|, | Perform Cargo Inspection, Screen All Cargos|⟩, ⟨| Perform Cargo Inspection, Send for Inspection |⟩}

- h_1' = {⟨ |**Clean Floors, gx_3**|, |Clean Floors, gx_3|⟩, ⟨assign, assign⟩, ⟨found, found⟩, ⟨ |Dispose Trash, Carry|, |Dispose Trash, Carry| ⟩, ⟨|**Alert and Neutralize, gx_1**|, |Alert and Neutralize, Carry|⟩, ⟨inspect, inspect⟩, ⟨|**Send for Inspection, gx_2**|, |Send for Inspection, gx_2|⟩, ⟨|**gx_5, Push**|, |gx_5, Push| ⟩};
- i_1'={⟨Carrier, Carrier ⟩, ⟨Lifter, Lifter ⟩, ⟨Pusher, Pusher⟩, ⟨Helper, Helper⟩, ⟨**rx_5, rx_5**⟩, ⟨Trash Collector, Carrier⟩, ⟨Janitor, Janitor⟩, ⟨**rx_3, rx_3**⟩, ⟨Computer, Computer⟩, ⟨Patroller, Patroller⟩, ⟨Neutralizer, Neutralizer⟩, ⟨**rx_1, Carrier**⟩, ⟨Screener, Screener⟩, ⟨Transporter, Transporter⟩, ⟨**rx_2, rx_2**⟩ };
- j_1' = {⟨|**Janitor, rx_3**|, |Janitor, rx_3|⟩, ⟨|Trash Collector, Carrier|, |Trash Collector, Carrier|⟩, ⟨|**Neutralizer, rx_1**|, |Neutralizer, Carrier|⟩, ⟨|**Transporter, rx_2**|, |Transporter, rx_2|⟩, ⟨ready, ready⟩, ⟨sync, sync⟩, ⟨|**rx_5, Pusher|,** |rx_5, Pusher|⟩};
- k_1' = {⟨|Carrier, Deliver|, |Carrier, Deliver|⟩, ⟨|Lifter, Load and Unload |, |Lifter, Load and Unload |⟩, ⟨|Pusher, Start Pushing|, |Pusher, Start Pushing|⟩, ⟨|Helper, Assist Pushing|, |Helper, Assist Pushing|⟩, ⟨|Janitor, Clean Floors|, |Janitor, Clean Floors|⟩, ⟨|Trash Collector, Dispose Trash|, |Trash Collector, Dispose Trash|⟩, ⟨|Computer, Compute Paths|, |Computer, Compute Paths|⟩, ⟨|Patroller, Patrol|⟩, ⟨|Patroller, Patrol|⟩, ⟨|Neutralizer, Alert and Neutralize|, |Neutralizer, Alert and Neutralize|⟩, ⟨Screener, Screen All Cargos⟩, ⟨Transporter, Send for Inspection⟩};

From Figure 6.7, Φ_2' is an organization homomorphism. We have:

Φ_2': Transportation $\rightarrow$ Cram_carry$_2$ such that Φ_2' = ⟨ f_2', g_2', h_2', i_2', j_2', k_2' ⟩ where:

- f_2' = {⟨g_root, g_root⟩, ⟨Transport Object, Transport Object⟩, ⟨Carry, Carry⟩, ⟨Deliver, Deliver⟩, ⟨Load and Unload, Load and Unload⟩, ⟨**gx_6**, **Alert and Neutralize**⟩, ⟨Push, Push⟩, ⟨Start Pushing, Start Pushing⟩, ⟨ Assist Pushing, Assist Pushing⟩, ⟨**gx_5**, **gx_5**⟩ }
- g_2' = {⟨|g_root, Transport Object|, |g_root, Transport Object|⟩, ⟨|Transport Object, Carry|, |Transport Object, Carry|⟩, ⟨|Carry, Deliver|, |Carry, Deliver|⟩, ⟨|Carry, Load and Unload|, |Carry, Load and Unload |⟩, ⟨|Transport Object, Push|, |Transport Object, Push|⟩, ⟨|Push, Start Pushing|, |Push, Start Pushing|⟩, ⟨|Push, Assist Pushing|⟩, ⟨|Push, Assist Pushing|⟩}
- h_2' = {⟨|gx_6, Carry|, |Alert and Neutralize, Carry|⟩, ⟨|gx_5, Push|, |gx_5, Push|⟩}
- i_2'= {⟨Carrier, Carrier ⟩, ⟨Lifter, Lifter ⟩, ⟨**rx_6, Neutralizer**⟩, ⟨Pusher, Pusher⟩, ⟨Helper, Helper⟩, ⟨**rx_5, rx_5**⟩}
- j_2' = {⟨ready, ready⟩, ⟨|**rx_6, Carrier|**, |**Neutralizer, Carrier|**⟩, ⟨sync, sync⟩, ⟨|**rx_5, Pusher|**, |**rx_5, Pusher|**⟩}
- k_2' = {⟨|Carrier, Deliver|, |Carrier, Deliver|⟩, ⟨|Lifter, Load and Unload |, |Lifter, Load and Unload |⟩, ⟨|Pusher, Start Pushing|, |Pusher, Start Pushing|⟩, ⟨|Helper, Assist Pushing|, |Helper, Assist Pushing|⟩}

Cram_carry$_2$ along with homomorphism Φ_1' and Φ_2' (Figure 6.7) represent the pushout of organization Shared$_2$ with homomorphism Φ_1 and Φ_2 . In fact, we have:

- f_1' ○ f_1 = {⟨g_0, g_root⟩, ⟨g_1, Transport Object⟩, ⟨g_2, Carry⟩, ⟨g_3, Deliver⟩, ⟨g_4, Load and unload⟩, ⟨g_5, Alert and Neutralize⟩, ⟨**g_6, Carry**⟩, ⟨g_7, Push⟩, ⟨g_8, Start Pushing⟩, ⟨g_9, Assist Pushing⟩, ⟨**g_{10}, gx_5**⟩}
- f_2' ○ f_2 = {⟨g_0, g_root⟩, ⟨g_1, Transport Object⟩, ⟨g_2, Carry⟩, ⟨g_3, Deliver⟩, ⟨g_4, Load and unload⟩, ⟨g_5, Alert and Neutralize⟩, ⟨**g_6, Carry**⟩, ⟨g_7, Push⟩, ⟨g_8, Start Pushing⟩, ⟨g_9, Assist Pushing⟩, ⟨**g_{10}, gx_5**⟩}
- g_1' ○ g_1 = {⟨|g_0,g_1 |, | g_root, Transport Object |⟩, ⟨|g_1,g_2 |, |Transport Object, Carry |⟩, ⟨| g_2,g_3|, |Carry, Deliver |⟩, ⟨|g_2,g_4 |, |Carry, Load and Unload |⟩, ⟨|g_1,g_7 |, |Transport Object, Push|⟩, ⟨|g_7,g_8 |, |Push, Start Pushing |⟩, ⟨|g_7,g_9 |, |Push, Assist Pushing|⟩};

- g_2' ○ g_2 = {⟨|g_0,g_1 |, | g_root, Transport Object |⟩, ⟨|g_1,g_2 |, |Transport Object, Carry |⟩, ⟨| g_2,g_3|, |Carry, Deliver |⟩, ⟨|g_2,g_4 |, |Carry, Load and Unload |⟩, ⟨|g_1,g_7 |, |Transport Object, Push|⟩, ⟨|g_7,g_8 |, |Push, Start Pushing |⟩, ⟨|g_7,g_9 |, |Push, Assist Pushing|⟩};
- h_1' ○ h_1 = {⟨|g_5, g_6|, |Alert and Neutralize, Carry|⟩, ⟨|g_{10}, g_7|, |gx_5, Push|⟩};
- h_2' ○ h_2 = {⟨|g_5, g_6|, |Alert and Neutralize, Carry|⟩, ⟨|g_{10}, g_7|, |gx_5, Push|⟩};
- i_1' ○ i_1 = {⟨r_1, Carrier⟩, ⟨r_2, Lifter⟩, ⟨r_3, Neutralizer⟩, ⟨**r_4, Carrier**⟩, ⟨r_5, Pusher⟩, ⟨r_6, Helper⟩, ⟨**r_7, rx_5**⟩};
- i_2' ○ i_2 = {⟨r_1, Carrier⟩, ⟨r_2, Lifter⟩, ⟨r_3, Neutralizer⟩, ⟨**r_4, Carrier**⟩, ⟨r_5, Pusher⟩, ⟨r_6, Helper⟩, ⟨**r_7, rx_5**⟩};
- j_1' ○ j_1 = {⟨|r_1, r_2|, |Carrier, Lifter|⟩, ⟨|r_3, r_4|, |**Neutralizer, Carrier**|⟩, ⟨|r_5, r_6|, |Pusher, Helper|⟩, ⟨|r_7, r_5|, |**rx_5, Pusher**|⟩};
- j_2' ○ j_2 = {⟨|r_1, r_2|, |Carrier, Lifter|⟩, ⟨|r_3, r_4|, |**Neutralizer, Carrier**|⟩, ⟨|r_5, r_6|, |Pusher, Helper|⟩, ⟨|r_7, r_5|, |**rx_5, Pusher**|⟩};
- k_1' ○ k_1 = {⟨|r_1,g_3|, |Carrier, Deliver|⟩, ⟨|r_2, g_4|, |Lifter, Load and Unload|⟩, ⟨|r_5, g_8|, |Pusher, Start Pushing|⟩, ⟨|r_6, g_9|, |helper, Assist Pushing|⟩};
- k_2' ○ k_2 = {⟨|r_1,g_3|, |Carrier, Deliver|⟩, ⟨|r_2, g_4|, |Lifter, Load and Unload|⟩, ⟨|r_5, g_8|, |Pusher, Start Pushing|⟩, ⟨|r_6, g_9|, |helper, Assist Pushing|⟩};

Hence, we can see that f_1' ○ f_1 = f_2' ○ f_2, g_1' ○ g_1 = g_2' ○ g_2, h_1' ○ h_1 = i_2' ○ i_2, j_1' ○ j_1 = k_2' ○ k_2. As Φ_1' ○ Φ_1 = ⟨ f_1' ○ f_1, g_1' ○ g_1, h_1' ○ h_1, i_1' ○ i_1, j_1' ○ j_1, k_1' ○ k_1⟩ and Φ_2' ○ Φ_2 = ⟨ f_2' ○ f_2, g_2' ○ g_2, h_2' ○ h_2, i_2' ○ i_2, j_2' ○ j_2, k_2' ○ k_2⟩, we have **Φ_1' ○ Φ_1 = Φ_2' ○ Φ_2.**

Organization Homomorphisms for constructing Cram_push from composition (C3):

From Figure 6.8, Φ_1 is an organization homomorphism. We have:

Φ_1: $Shared_3$ → $Cram\_carry_2$ such that Φ_1 = ⟨ f_1, g_1, h_1, i_1, j_1, k_1 ⟩, where:

- f_1 = {⟨g_0, g_root⟩, ⟨g_1, Transport Object⟩, ⟨g_2, Carry⟩, ⟨g_3, Deliver⟩, ⟨g_4, Load and unload⟩, ⟨g_5, Send for Inspection⟩, ⟨**g_6, gx_2**⟩, ⟨g_7, Push⟩, ⟨g_8, Start Pushing⟩, ⟨g_9, Assist Pushing⟩, ⟨**g_{10}, gx_5**⟩};

- g_1 = {⟨|g_0,g_1 |, | g_root, Transport Object |⟩, ⟨|g_1,g_2 |, |Transport Object, Carry |⟩, ⟨| g_2,g_3|, |Carry, Deliver |⟩, ⟨|g_2,g_4 |, |Carry, Load and Unload |⟩, ⟨|g_1,g_7 |, |Transport Object, Push|⟩, ⟨|g_7,g_8 |, |Push, Start Pushing |⟩, ⟨|g_7,g_9 |, |Push, Assist Pushing|⟩};
- h_1 = {⟨|g_5, g_6|, | Send for Inspection, gx_2|⟩, ⟨|g_{10}, g_7|, |gx_5, Push|⟩};
- i_1= {⟨r_1, Carrier⟩, ⟨r_2, Lifter⟩, ⟨r_3, Transporter⟩, ⟨**r_4, rx_2**⟩, ⟨r_5, Pusher⟩, ⟨r_6, Helper⟩, ⟨**r_7, rx_5**⟩};
- j_1 = {⟨|r_1, r_2|, |Carrier, Lifter|⟩, ⟨|r_3, r_4|, |**Transporter, rx_2**|⟩, ⟨|r_5, r_6|, |Pusher, Helper|⟩, ⟨|r_7, r_5|, |**rx_5, Pusher**|⟩};
- k_1 = {⟨|r_1,g_3|, |Carrier, Deliver|⟩, ⟨|r_2, g_4|, |Lifter, Load and Unload|⟩, ⟨|r_5, g_8|, |Pusher, Start Pushing|⟩, ⟨|r_6, g_9|, |helper, Assist Pushing|⟩};

From Figure 6.8, Φ_2 is an organization homomorphism. We have:

Φ_2: $Shared_3 \rightarrow$ Transportation such that $\Phi_2 = \langle f_2, g_2, h_2, i_2, j_2, k_2 \rangle$ where:

- f_2 = {⟨g_0, g_root⟩, ⟨g_1, Transport Object⟩, ⟨g_2, Carry⟩, ⟨g_3, Deliver⟩, ⟨g_4, Load and unload⟩, ⟨**g_5, gx_5**⟩, ⟨g_6, Push⟩, ⟨g_7, Push⟩, ⟨g_8, Start Pushing⟩, ⟨g_9, Assist Pushing⟩, ⟨**g_{10}, gx_5**⟩};
- g_2 = {⟨|g_0,g_1|, |g_root, Transport Object|⟩, ⟨|g_1,g_2|, |Transport Object, Carry|⟩, ⟨|g_2,g_3|, |Carry, Deliver |⟩, ⟨|g_2,g_4 |, |Carry, Load and Unload |⟩, ⟨|g_1,g_7 |, |Transport Object, Push|⟩, ⟨|g_7,g_8 |, |Push, Start Pushing |⟩, ⟨|g_7,g_9 |, |Push, Assist Pushing|⟩};
- h_2 = {⟨|g_5, g_6|, | gx_5, Push|⟩, ⟨|g_{10}, g_7|, |gx_5, Push|⟩};
- i_2 = {⟨r_1, Carrier⟩, ⟨r_2, Lifter⟩, ⟨**r_3, rx_5**⟩, ⟨r_4, Pusher⟩, ⟨r_5, Pusher⟩, ⟨r_6, Helper⟩};
- j_2 = {⟨|r_1, r_2|, |Carrier, Lifter|⟩, ⟨|r_3, r_4|, | **rx_5, Pusher**|⟩, ⟨|r_5, r_6|, |Pusher, Helper|⟩, ⟨**r_7, rx_5**⟩};
- k_2 = {⟨|r_1,g_3|, |Carrier, Deliver|⟩, ⟨|r_2, g_4|, |Lifter, Load and Unload|⟩, ⟨|r_5, g_8|, |Pusher, Start Pushing|⟩, ⟨|r_6, g_9|, |helper, Assist Pushing|⟩};

From Figure 6.8, Φ_1' is an organization homomorphism. We have:

Φ_1': $Cram\_carry_2 \rightarrow$ Cram_push such that $\Phi_1' = \langle f_1', g_1', h_1', i_1', j_1', k_1' \rangle$ where:

- f_1' = {⟨g_root, g_root⟩, ⟨Transport Object, Transport Object⟩, ⟨Carry, Carry⟩, ⟨Deliver, Deliver⟩, ⟨Load and Unload, Load and Unload⟩, ⟨Push, Push⟩, ⟨Start Pushing, Start Pushing⟩, ⟨Assist Pushing, Assist Pushing⟩, ⟨**gx_5, Send for Inspection**⟩, ⟨Manage Airport, Manage Airport⟩, ⟨Operate Sanitary Maintenance, Operate Sanitary Maintenance⟩, ⟨Clean Floors, Clean Floors⟩, ⟨Dispose Trash, Dispose Trash⟩, ⟨**gx_3, gx_3**⟩, ⟨Monitor Buildings, Monitor Buildings⟩, ⟨Compute Paths, Compute Paths⟩, ⟨Patrol, Patrol⟩, ⟨Alert and Neutralize, Alert and Neutralize ⟩, ⟨Perform Cargo Inspection, Perform Cargo Inspection,⟩, ⟨Screen All Cargos, Screen All Cargos ⟩, ⟨Send for Inspection, Send for Inspection⟩, ⟨**gx_2, Push**⟩}
- g_1' = {⟨|g_root, Transport Object|, |g_root, Transport Object |⟩, ⟨|Transport Object, Carry|, | Transport Object, Carry|⟩, ⟨|Carry, Deliver|, | Carry, Deliver|⟩, ⟨|Carry, Load and Unload |, | Carry, Load and Unload |⟩, ⟨|g_root, Manage Airport| , |g_root, Manage Airport|⟩, ⟨|Manage Airport, Operate Sanitary Maintenance|, |Manage Airport, Operate Sanitary Maintenance|⟩, ⟨|Operate Sanitary Maintenance, Clean Floors|, |Operate Sanitary Maintenance, Clean Floors|⟩, ⟨|Operate Sanitary Maintenance, Dispose Trash|, |Operate Sanitary Maintenance, Dispose Trash|⟩, ⟨|Manage Airport, Monitor Buildings|, |Manage Airport, Monitor Buildings|⟩, ⟨|Monitor Buildings, Compute Paths|, |Monitor Buildings, Compute Paths|⟩, ⟨|Monitor Buildings, Patrol|, |Monitor Buildings, Patrol|⟩, ⟨|Monitor Buildings, Alert and Neutralize|, |Monitor Buildings, Alert and Neutralize|⟩, ⟨|Manage Airport, Perform Cargo Inspection|, | Perform Cargo Inspection, Screen All Cargos|⟩, ⟨| Perform Cargo Inspection, Send for Inspection |⟩}
- h_1' = {⟨ |**Clean Floors, gx_3**|, |Clean Floors, gx_3|⟩, ⟨assign, assign⟩, ⟨found, found⟩, ⟨ |Dispose Trash, Carry|, |Dispose Trash, Carry| ⟩, ⟨|**Alert and Neutralize, Carry**|, |Alert and Neutralize, Carry|⟩, ⟨inspect, inspect⟩, ⟨|**Send for Inspection, gx_2**|, |Send for Inspection, Push|⟩, ⟨|**gx_5, Push**|, |Send for Inspection, Push|⟩};
- i_1'={⟨Carrier, Carrier ⟩, ⟨Lifter, Lifter ⟩, ⟨Pusher, Pusher⟩, ⟨Helper, Helper⟩, ⟨**rx_5, Transporter**⟩, ⟨Trash Collector, Carrier⟩, ⟨Janitor, Janitor⟩, ⟨**rx_3, rx_3**⟩, ⟨Computer,

Computer⟩, ⟨Patroller, Patroller⟩, ⟨Neutralizer, Neutralizer⟩, ⟨Screener, Screener⟩, ⟨Transporter, Transporter⟩, ⟨**rx_2, Pusher**⟩ };

- j_1' = {⟨|**Janitor, rx_3**|, |Janitor, rx_3|⟩, ⟨|Trash Collector, Carrier|, |Trash Collector, Carrier|⟩, ⟨|**Neutralizer, Carrier**|, |Neutralizer, Carrier|⟩, ⟨|**Transporter, rx_2**|, |Transporter, Pusher|⟩, ⟨ready, ready⟩, ⟨sync, sync⟩, ⟨|**rx_5, Pusher|,** |Transporter, Pusher|⟩};
- k_1' = {⟨|Carrier, Deliver|, |Carrier, Deliver|⟩, ⟨|Lifter, Load and Unload |, |Lifter, Load and Unload |⟩, ⟨|Pusher, Start Pushing|, |Pusher, Start Pushing|⟩, ⟨|Helper, Assist Pushing|, |Helper, Assist Pushing|⟩, ⟨|Janitor, Clean Floors|, |Janitor, Clean Floors|⟩, ⟨|Trash Collector, Dispose Trash|, |Trash Collector, Dispose Trash|⟩, ⟨|Computer, Compute Paths|, |Computer, Compute Paths|⟩, ⟨|Patroller, Patrol|⟩, ⟨|Patroller, Patrol|⟩, ⟨|Neutralizer, Alert and Neutralize|, |Neutralizer, Alert and Neutralize|⟩, ⟨Screener, Screen All Cargos⟩, ⟨Transporter, Send for Inspection⟩};

From Figure 6.8, Φ_2' is an organization homomorphism. We have:

Φ_2': Transportation → Cram_push such that Φ_2' = ⟨ f_2', g_2', h_2', i_2', j_2', k_2' ⟩ where:

- f_2' = {⟨g_root, g_root⟩, ⟨Transport Object, Transport Object⟩, ⟨Carry, Carry⟩, ⟨Deliver, Deliver⟩, ⟨Load and Unload, Load and Unload⟩, ⟨Push, Push⟩, ⟨Start Pushing, Start Pushing⟩, ⟨ Assist Pushing, Assist Pushing⟩, ⟨**gx_5**, **Send for Inspection**⟩ }
- g_2' = {⟨|g_root, Transport Object|, |g_root, Transport Object|⟩, ⟨|Transport Object, Carry|, |Transport Object, Carry|⟩, ⟨|Carry, Deliver|, |Carry, Deliver|⟩, ⟨|Carry, Load and Unload|, |Carry, Load and Unload |⟩, ⟨|Transport Object, Push|, |Transport Object, Push|⟩, ⟨|Push, Start Pushing|, |Push, Start Pushing|⟩, ⟨|Push, Assist Pushing|⟩, ⟨|Push, Assist Pushing|⟩}
- h_2' = {⟨|gx_5, Push|, |Send for Inspection, Push|⟩}
- i_2'= {⟨Carrier, Carrier ⟩, ⟨Lifter, Lifter ⟩, ⟨Pusher, Pusher⟩, ⟨Helper, Helper⟩, ⟨**rx_5, Transporter**⟩}
- j_2' = {⟨ready, ready⟩, ⟨sync, sync⟩, ⟨|**rx_5, Pusher|,** |**Transporter, Pusher**|⟩}

- k_2' = {⟨|Carrier, Deliver|, |Carrier, Deliver|⟩, ⟨|Lifter, Load and Unload |, |Lifter, Load and Unload |⟩, ⟨|Pusher, Start Pushing|, |Pusher, Start Pushing|⟩, ⟨|Helper, Assist Pushing|, |Helper, Assist Pushing|⟩}

Cram_push along with homomorphism Φ_1' and Φ_2' (Figure 6.8) represent the pushout of organization $Shared_3$ with homomorphism Φ_1 and Φ_2. In fact, we have:

- f_1' ○ f_1 = {⟨g_0, g_root⟩, ⟨g_1, Transport Object⟩, ⟨g_2, Carry⟩, ⟨g_3, Deliver⟩, ⟨g_4, Load and unload⟩, ⟨g_5, Send for Inspection⟩, ⟨**g_6, Push**⟩, ⟨g_7, Push⟩, ⟨g_8, Start Pushing⟩, ⟨g_9, Assist Pushing⟩, ⟨**g_{10}, Send for Inspection**⟩};
- f_2' ○ f_2 = {⟨g_0, g_root⟩, ⟨g_1, Transport Object⟩, ⟨g_2, Carry⟩, ⟨g_3, Deliver⟩, ⟨g_4, Load and unload⟩, ⟨g_5, Send for Inspection⟩, ⟨**g_6, Push**⟩, ⟨g_7, Push⟩, ⟨g_8, Start Pushing⟩, ⟨g_9, Assist Pushing⟩, ⟨**g_{10}, Send for Inspection**⟩};
- g_1' ○ g_1 = {⟨|g_0,g_1 |, | g_root, Transport Object |⟩, ⟨|g_1,g_2 |, |Transport Object, Carry |⟩, ⟨| g_2,g_3|, |Carry, Deliver |⟩, ⟨|g_2,g_4 |, |Carry, Load and Unload |⟩, ⟨|g_1,g_7 |, |Transport Object, Push|⟩, ⟨|g_7,g_8 |, |Push, Start Pushing |⟩, ⟨|g_7,g_9 |, |Push, Assist Pushing|⟩};
- g_2' ○ g_2 = {⟨|g_0,g_1 |, | g_root, Transport Object |⟩, ⟨|g_1,g_2 |, |Transport Object, Carry |⟩, ⟨| g_2,g_3|, |Carry, Deliver |⟩, ⟨|g_2,g_4 |, |Carry, Load and Unload |⟩, ⟨|g_1,g_7 |, |Transport Object, Push|⟩, ⟨|g_7,g_8 |, |Push, Start Pushing |⟩, ⟨|g_7,g_9 |, |Push, Assist Pushing|⟩};
- h_1' ○ h_1 = {⟨|g_5, g_6|, | Send for Inspection, Push|⟩, ⟨|g_{10}, g_7|, |Send for Inspection, Push|⟩};
- h_2' ○ h_2 = {⟨|g_5, g_6|, | Send for Inspection, Push|⟩, ⟨|g_{10}, g_7|, |Send for Inspection, Push|⟩};
- i_1' ○ i_1 = {⟨r_1, Carrier⟩, ⟨r_2, Lifter⟩, ⟨r_3, Transporter⟩, ⟨**r_4, Pusher**⟩, ⟨r_5, Pusher⟩, ⟨r_6, Helper⟩, ⟨**r_7, Transporter**⟩};
- i_2' ○ i_2 = {⟨r_1, Carrier⟩, ⟨r_2, Lifter⟩, ⟨r_3, Transporter⟩, ⟨**r_4, Pusher**⟩, ⟨r_5, Pusher⟩, ⟨r_6, Helper⟩, ⟨**r_7, Transporter**⟩};
- j_1' ○ j_1 = {⟨|r_1, r_2|, |Carrier, Lifter|⟩, ⟨|r_3, r_4|, |**Transporter, Pusher**|⟩, ⟨|r_5, r_6|, |Pusher, Helper|⟩, ⟨|r_7, r_5|, |**Transporter, Pusher**|⟩}

- j_1' ○ j_1 = {⟨|r_1, r_2|, |Carrier, Lifter|⟩, ⟨|r_3, r_4|, |**Transporter, Pusher**|⟩, ⟨|r_5, r_6|, |Pusher, Helper|⟩, ⟨|r_7, r_5|, |**Transporter, Pusher**|⟩}
- k_1' ○ k_1 = {⟨|r_1,g_3|, |Carrier, Deliver|⟩, ⟨|r_2, g_4|, |Lifter, Load and Unload|⟩, ⟨|r_5, g_8|, |Pusher, Start Pushing|⟩, ⟨|r_6, g_9|, |helper, Assist Pushing|⟩};
- k_2' ○ k_2 = {⟨|r_1,g_3|, |Carrier, Deliver|⟩, ⟨|r_2, g_4|, |Lifter, Load and Unload|⟩, ⟨|r_5, g_8|, |Pusher, Start Pushing|⟩, ⟨|r_6, g_9|, |helper, Assist Pushing|⟩}

Hence, we can see that f_1' ○ f_1 = f_2' ○ f_2, g_1' ○ g_1 = g_2' ○ g_2, h_1' ○ h_1 = i_2' ○ i_2, j_1' ○ j_1 = k_2' ○ k_2. As Φ_1' ○ Φ_1 = ⟨ f_1' ○ f_1, g_1' ○ g_1, h_1' ○ h_1, i_1' ○ i_1, j_1' ○ j_1, k_1' ○ k_1⟩ and Φ_2' ○ Φ_2 = ⟨ f_2' ○ f_2, g_2' ○ g_2, h_2' ○ h_2, i_2' ○ i_2, j_2' ○ j_2, k_2' ○ k_2⟩, we have **Φ_1' ○ Φ_1 = Φ_2' ○ Φ_2.**

Organization Homomorphisms for constructing Cram_carry$_1$ from composition (C4):

From Figure 6.9, Φ_1 is an organization homomorphism. We have:

Φ_1: Shared$_4$ → Cram_push such that Φ_1 = ⟨ f_1, g_1, h_1, i_1, j_1, k_1 ⟩, where:

- f_1 = {⟨g_0, g_root⟩, ⟨**g_1, Clean Floors**⟩, ⟨**g_2, gx_3**⟩};
- g_1 = ∅;
- h_1 = {⟨|g_1, g_2|, |Clean Floors, gx_3|⟩};
- i_1={⟨**r_1, Janitor**⟩, ⟨**r_2, rx_3**⟩};
- j_1 = {⟨|r_1, r_2|, |Janitor, rx_3|⟩};
- k_1 = ∅;

From Figure 6.9, Φ_2 is an organization homomorphism. We have:

Φ_2: Shared$_4$ → Clean such that Φ_2 = ⟨ f_2, g_2, h_2, i_2, j_2, k_2 ⟩ where:

- f2 = {⟨ g_0, g_root ⟩, ⟨ **g_1, gx_7**⟩, ⟨ **g_2, Divide Area**⟩};
- g2 = ∅;
- h2 = {⟨ |g_1, g_2|, |gx_7, Divide Area|⟩};
- i2={ ⟨ r_1, rx_7⟩, ⟨ r_2, Leader⟩};
- j2 = {⟨ |r_1, r_2|, |rx_7, Leader| ⟩};
- k2 = ∅;

From Figure 6.9, Φ_1' is an organization homomorphism. We have:

Φ_1': Cram_push → Cram_clean such that Φ_1' = ⟨ f_1', g_1', h_1', i_1', j_1', k_1' ⟩ where:

- f_1' = {⟨g_root, g_root⟩, ⟨Transport Object, Transport Object⟩, ⟨Carry, Carry⟩, ⟨Deliver, Deliver⟩, ⟨Load and Unload, Load and Unload⟩, ⟨Push, Push⟩, ⟨Start Pushing, Start Pushing⟩, ⟨Assist Pushing, Assist Pushing⟩, ⟨Manage Airport, Manage Airport⟩, ⟨Operate Sanitary Maintenance, Operate Sanitary Maintenance⟩, ⟨Clean Floors, Clean Floors⟩, ⟨Dispose Trash, Dispose Trash⟩, ⟨**gx_3, Divide Area**⟩, ⟨Monitor Buildings, Monitor Buildings⟩, ⟨Compute Paths, Compute Paths⟩, ⟨Patrol, Patrol⟩, ⟨Alert and Neutralize, Alert and Neutralize ⟩, ⟨Perform Cargo Inspection, Perform Cargo Inspection,⟩, ⟨Screen All Cargos, Screen All Cargos ⟩, ⟨Send for Inspection, Send for Inspection⟩}
- g_1' = {⟨|g_root, Transport Object|, |g_root, Transport Object |⟩, ⟨|Transport Object, Carry|, | Transport Object, Carry|⟩, ⟨|Carry, Deliver|, | Carry, Deliver|⟩, ⟨|Carry, Load and Unload |, | Carry, Load and Unload |⟩, ⟨|g_root, Manage Airport| , |g_root, Manage Airport|⟩, ⟨|Manage Airport, Operate Sanitary Maintenance|, |Manage Airport, Operate Sanitary Maintenance|⟩, ⟨|Operate Sanitary Maintenance, Clean Floors|, |Operate Sanitary Maintenance, Clean Floors|⟩, ⟨|Operate Sanitary Maintenance, Dispose Trash|, |Operate Sanitary Maintenance, Dispose Trash|⟩, ⟨|Manage Airport, Monitor Buildings|, |Manage Airport, Monitor Buildings|⟩, ⟨|Monitor Buildings, Compute Paths|, |Monitor Buildings, Compute Paths|⟩, ⟨|Monitor Buildings, Patrol|, |Monitor Buildings, Patrol|⟩, ⟨|Monitor Buildings, Alert and Neutralize|, |Monitor Buildings, Alert and Neutralize|⟩, ⟨|Manage Airport, Perform Cargo Inspection|, | Perform Cargo Inspection, Screen All Cargos|⟩, ⟨| Perform Cargo Inspection, Send for Inspection |⟩}
- h_1' = {⟨ |**Clean Floors, gx_3**|, |Clean Floors, Divide Area|⟩, ⟨assign, assign⟩, ⟨found, found⟩, ⟨ |Dispose Trash, Carry|, |Dispose Trash, Carry| ⟩, ⟨|**Alert and Neutralize, Carry**|, |Alert and Neutralize, Carry|⟩, ⟨inspect, inspect⟩, ⟨|**Send for Inspection, Push**|, |Send for Inspection, Push|⟩};

- i_1'={⟨Carrier, Carrier ⟩, ⟨Lifter, Lifter ⟩, ⟨Pusher, Pusher⟩, ⟨Helper, Helper⟩, ⟨Trash Collector, Carrier⟩, ⟨Janitor, Janitor⟩, ⟨**rx_3, Leader**⟩, ⟨Computer, Computer⟩, ⟨Patroller, Patroller⟩, ⟨Neutralizer, Neutralizer⟩, ⟨Screener, Screener⟩, ⟨Transporter, Transporter⟩};
- j_1' = {⟨|**Janitor, rx_3**|, |Janitor, Leader|⟩, ⟨|Trash Collector, Carrier|, |Trash Collector, Carrier|⟩, ⟨|**Neutralizer, Carrier**|, |Neutralizer, Carrier|⟩, ⟨|**Transporter, Pusher**|, |Transporter, Pusher|⟩, ⟨ready, ready⟩, ⟨sync, sync⟩};
- k_1' = {⟨|Carrier, Deliver|, |Carrier, Deliver|⟩, ⟨|Lifter, Load and Unload |, |Lifter, Load and Unload |⟩, ⟨|Pusher, Start Pushing|, |Pusher, Start Pushing|⟩, ⟨|Helper, Assist Pushing|, |Helper, Assist Pushing|⟩, ⟨|Janitor, Clean Floors|, |Janitor, Clean Floors|⟩, ⟨|Trash Collector, Dispose Trash|, |Trash Collector, Dispose Trash|⟩, ⟨|Computer, Compute Paths|, |Computer, Compute Paths|⟩, ⟨|Patroller, Patrol|⟩, ⟨|Patroller, Patrol|⟩, ⟨|Neutralizer, Alert and Neutralize|, |Neutralizer, Alert and Neutralize|⟩, ⟨Screener, Screen All Cargos⟩, ⟨Transporter, Send for Inspection⟩};

From Figure 6.9, Φ_2' is an organization homomorphism. We have:

Φ_2': Clean → Cram_clean such that Φ_2' = ⟨ f_2', g_2', h_2', i_2', j_2', k_2' ⟩ where:

- f_2' = {⟨g_root, g_root⟩, ⟨Clean Area, Clean Area⟩, ⟨Divide Area, Divide Area⟩, ⟨Clean, Clean⟩, ⟨Deep Clean, Deep Clean⟩, ⟨Vacuum, Vacuum⟩, ⟨Sweep, Sweep⟩, ⟨Mop, Mop⟩, ⟨**gx_7**, **Clean Floors**⟩ }
- g_2' = {⟨|g_root, Clean Area|, |g_root, Clean Area|⟩, ⟨|Clean Area, Divide Area|, |Clean Area, Divide Area|⟩, ⟨|Clean Area, Clean|, |Clean Area, Clean|⟩, ⟨|Clean, Deep Clean|, |Clean, Deep Clean|⟩, ⟨|Clean, Vacuum|, |Clean, Vacuum|⟩, ⟨|Deep Clean, Sweep|, |Deep Clean, Sweep|⟩, ⟨|Deep Clean, Mop|⟩, ⟨|Deep Clean, Mop|⟩}
- h_2' = {⟨assignArea, assignArea⟩, ⟨precedes, precedes⟩, ⟨|gx_7, Divide Area|, |gx_7, Divide Area|⟩}
- i_2'= {⟨Sweeper, Sweeper⟩, ⟨Mopper, Mopper⟩, ⟨Vacuumer, Vacuumer⟩, ⟨Leader, Leader⟩, ⟨**rx_7, Janitor**⟩}
- j_2' = {⟨|**rx_7, Leader|**, |**Janitor, Leader**|⟩}

- k_2' = {⟨|Leader, Divide Area|, |Leader, Divide Area|⟩, ⟨|Sweeper, Sweep|, |Sweeper, Sweep|⟩, ⟨|Mopper, Mop|, |Mopper, Mop|⟩, ⟨|Vacuumer, Vacuum|, |Vacuumer, Vacuum|⟩}

Cram_clean along with homomorphism Φ_1' and Φ_2' (Figure 6.9) represent the pushout of organization $Shared_4$ with homomorphism Φ_1 and Φ_2. In fact, we have:

- f_1' ○ f_1 = {⟨g_0, g_root⟩, ⟨**g_1, Clean Floors**⟩, ⟨**g_2, Divide Area**⟩};
- f_2' ○ f_2 = {⟨g_0, g_root⟩, ⟨**g_1, Clean Floors**⟩, ⟨**g_2, Divide Area**⟩};
- g_1' ○ g_1 = ∅;
- g_2' ○ g_2 = ∅;
- h_1' ○ h_1 = {⟨|g_1, g_2|, |Clean Floors, Divide Area|⟩};
- h_2' ○ h_2 = {⟨|g_1, g_2|, |Clean Floors, Divide Area|⟩};
- i_1' ○ i_1 = {⟨**r_1, Janitor**⟩, ⟨**r_2, Leader**⟩};
- i_2' ○ i_2 = {⟨**r_1, Janitor**⟩, ⟨**r_2, Leader**⟩};
- j_1' ○ j_1 = {⟨|r_1, r_2|, |Janitor, Leader|⟩};
- j_1' ○ j_1 = {⟨|r_1, r_2|, |Janitor, Leader|⟩};
- k_1' ○ k_1 = ∅;
- k_2' ○ k_2 = ∅;

Hence, we can see that f_1' ○ f_1 = f_2' ○ f_2, g_1' ○ g_1 = g_2' ○ g_2, h_1' ○ h_1 = i_2' ○ i_2, j_1' ○ j_1 = k_2' ○ k_2. As Φ_1' ○ Φ_1 = ⟨ f_1' ○ f_1, g_1' ○ g_1, h_1' ○ h_1, i_1' ○ i_1, j_1' ○ j_1, k_1' ○ k_1⟩ and Φ_2' ○ Φ_2 = ⟨ f_2' ○ f_2, g_2' ○ g_2, h_2' ○ h_2, i_2' ○ i_2, j_2' ○ j_2, k_2' ○ k_2⟩, we have **Φ_1' ○ Φ_1 = Φ_2' ○ Φ_2.**

□

A.3 Composition details for the Surveillance application using FTSP

This section presents the composition details for the Surveillance organization composed with the FTSP organization as presented in Section 6.2.5.

Let Φ_1: $Shared_0 \rightarrow$ Surveillance be an organization homomorphism such that Surveillance is defined as shown in Figure 6.15 and $Shared_0 = \langle G_0, ET_0, EG_0, g\_root, R_0, P_0, participant_0, achieves_0\rangle$ is an organization such that:

- $G_0 = \{g\_root , g_1, g_2, g_3, g_4\}$,
- $ET_0 = \varnothing$,
- $EG_0 = \{ |g_1,g_2|, |g_3,g_4|\}$
- $R_0 = \{ r_1, r_2, r_3, r_4\}$
- $P_0 = \{p_1, p_2\}$
- $participant_0 = \{\langle p_1, \langle r_1, r_2\}\rangle, \langle p_2, \langle r_3, r_4\}\rangle\}$
- $achieves_0 = \varnothing$

We define $\Phi_1 = \langle f_1, g_1, h_1, i_1, j_1, k_1 \rangle$ such that:

- $f_1 = \{\langle g\_root, g\_root\rangle, \langle \mathbf{g_1, Monitor\ Area}\rangle, \langle \mathbf{g_2, gx_1}\rangle, \langle \mathbf{g_3, Track\ Area}\rangle, \langle \mathbf{g_4, gx_2}\rangle\}$;
- $g_1 = \varnothing$;
- $h_1 = \{\langle |g_1, g_2|, |Monitor\ Area, gx_1|\rangle, \langle |g_3, g_4|, |Track\ Area, gx_2|\rangle\}$;
- $i_1 = \{ \langle \mathbf{r_1, Monitor}\rangle, \langle \mathbf{r_2, rx_1}\rangle, \langle \mathbf{r_3, Tracker}\rangle, \langle \mathbf{r_4, rx_2}\rangle\}$;
- $j_1 = \{\langle |r_1, r_2|, |Monitor, rx_1| \rangle, \langle |r_3, r_4|, |Tracker, rx_2|\rangle\}$;
- $k_1 = \varnothing$;

Let Φ_2: $Shared_0 \rightarrow$ FTSP be an organization homomorphism such that FTSP is defined as shown in Figure 6.13. We define $\Phi_2 = \langle f_2, g_2, h_2, i_2, j_2, k_2 \rangle$ such that:

- $f2 = \{\langle g\_root, g\_root\rangle, \langle \mathbf{g_1, gx_3}\rangle, \langle \mathbf{g_2, Compute\ Time}\rangle\}$;
- $g2 = \varnothing$;
- $h2 = \{\langle |g_1, g_2|, |gx_3, Compute\ Time|\rangle\}$;
- $i2 = \{ \langle r_1, rx_3\rangle, \langle r_2, Receiver\rangle\}$;
- $j2 = \{\langle |r_1, r_2|, |rx_3, Receiver|\rangle\}$;

- $k2 = \varnothing$;

Let Φ_1': Surveillance $\rightarrow$ Surveillance_FTSP an organization homomorphism such that Surveillance_FTSP is defined as shown in Figure 6.16. We have $\Phi_1' = \langle f_1', g_1', h_1', i_1', j_1', k_1' \rangle$ where:

- f_1' = {⟨g_root, g_root⟩, ⟨Surveillance, Surveillance⟩, ⟨Monitor, Monitor⟩, ⟨Monitor Area, Monitor Area⟩, ⟨Determine Coverage, Determine Coverage⟩, ⟨**gx_1, Compute Time**⟩, ⟨Track, Track⟩, ⟨Track Area, Track Area⟩, ⟨Divide Area, Divide Area⟩, ⟨**gx_2, Compute Time**⟩, ⟨Generate Reports, Generate Reports⟩, ⟨Define Area, Define Area⟩};
- g_1' = {⟨|Surveillance, Monitor| , |Surveillance, Monitor|⟩, ⟨|Monitor, Monitor Area|, |Monitor, Monitor Area|⟩, ⟨|Monitor, Determine Coverage|, |Monitor, Determine Coverage|⟩, ⟨|Surveillance, Track| , |Surveillance, Track|⟩, ⟨|Track, Track Area|, |Track, Track Area|⟩, ⟨|Track, Divide Area|, |Track, Divide Area|⟩, ⟨|g_root, Surveillance| , |g_root, Surveillance|⟩, ⟨|Surveillance, Generate Reports|, |Surveillance, Generate Reports|⟩, ⟨|Surveillance, Define Area|, |Surveillance, Define Area|⟩}
- h_1' = {⟨ |**Monitor Area, gx_1**|, |**Monitor Area, Compute Time**|⟩, ⟨monitor, monitor⟩, ⟨ |**Track Area, gx_2**|, |**Track Area, Compute Time**|⟩, ⟨track, track⟩, ⟨startMonitor, startMonitor⟩, ⟨startTrack, startTrack⟩};
- i_1'= {⟨Monitor, Monitor⟩, ⟨Coverage Processor, Coverage Processor⟩, ⟨**rx_1, Receiver**⟩, ⟨Tracker, Tracker⟩, ⟨Divider, Divider⟩, ⟨**rx_2, Receiver**⟩, ⟨User Interface, User Interface⟩};
- j_1' = {⟨|Monitor, rx_1|, |Monitor, Receiver|⟩, ⟨|Tracker, rx_2|, |Tracker, Receiver|⟩, ⟨sendData, sendData⟩};
- k_1' = {⟨|Monitor, Monitor Area|, |Monitor, Monitor Area|⟩, ⟨|Coverage Processor, Determine Coverage|, |Coverage Processor, Determine Coverage|⟩, ⟨|Tracker, Track Area|, |Tracker, Track Area|⟩, ⟨|Divider, Divide Area|, |Divider, Divide Area|⟩, ⟨|User Interface, Generate Reports|, |User Interface, Generate Reports|⟩, ⟨|User Interface, Define Area|, |User Interface, Define Area|⟩}

Let Φ_2': FTSP $\rightarrow$ Surveillance_FTSP an organization homomorphism such that $\Phi_2' = \langle f_2', g_2', h_2', i_2', j_2', k_2' \rangle$ where:

- f_2' = {⟨g_root, g_root⟩, ⟨FTSP, FTSP⟩, ⟨Compute Time, Compute Time⟩, ⟨Broadcast Time, Broadcast Time⟩, ⟨**gx$_3$**, **Monitor Area**⟩}
- g_2' = {⟨|g_root, FTSP|, |g_root, FTSP|⟩, ⟨|FTSP, Compute Time|, |FTSP, Compute Time|⟩, ⟨|FTSP, Broadcast Time|, |FTSP, Broadcast Time|⟩}
- h_2' = {⟨|gx$_3$, Compute Time|, |Monitor Area, Compute Time|⟩}
- i_2'= {⟨Receiver, Receiver⟩, ⟨Reference, Reference⟩, ⟨**rx$_3$, Monitor**⟩}
- j_2' = {⟨|**rx$_3$, Receiver|,** |**Monitor, Receiver**|⟩}
- k_2' = {⟨|Receiver, Compute Time|, |Receiver, Compute Time|⟩, ⟨|Reference, Broadcast Time|, |Reference, Broadcast Time|⟩}

Surveillance_FTSP along with homomorphism Φ_1' and Φ_2' represent the pushout of organization Shared$_0$ with homomorphism Φ_1 and Φ_2. In fact, we have:

- $f_1' \circ f_1$ = {⟨g_root, g_root⟩, ⟨**g$_1$, Monitor Area**⟩, ⟨**g$_2$, Compute Time**⟩, ⟨**g$_3$, Track Area**⟩, ⟨**g$_4$, Compute Time**⟩}
- $f_2' \circ f_2$ = {⟨g_root, g_root⟩, ⟨**g$_1$, Monitor Area**⟩, ⟨**g$_2$, Compute Time**⟩, ⟨**g$_3$, Track Area**⟩, ⟨**g$_4$, Compute Time**⟩}
- $g_1' \circ g_1 = \varnothing$;
- $g_2' \circ g_2 = \varnothing$;
- $h_1' \circ h_1$ = {⟨ |g$_1$, g$_2$|, |Monitor Area, Compute Time|⟩, ⟨ |g$_3$, g$_4$|, |Track Area, Compute Time|⟩};
- $h_2' \circ h_2$ = {⟨ |g$_1$, g$_2$|, |Monitor Area, Compute Time|⟩, ⟨ |g$_3$, g$_4$|, |Track Area, Compute Time|⟩};
- $i_1' \circ i_1$ = { ⟨ **r$_1$, Monitor**⟩, ⟨ **r$_2$, Receiver**⟩, ⟨**r$_3$, Tracker**⟩, ⟨ **r$_4$, Receiver**⟩};
- $i_2' \circ i_2$ = { ⟨ **r$_1$, Monitor**⟩, ⟨ **r$_2$, Receiver**⟩, ⟨**r$_3$, Tracker**⟩, ⟨ **r$_4$, Receiver**⟩};
- $j_1' \circ j_1$ = {⟨|r$_1$, r$_2$|, |Monitor, Receiver|⟩, ⟨|r$_3$, r$_4$|, |Tracker, Receiver|⟩};
- $j_1' \circ j_1$ = {⟨|r$_1$, r$_2$|, |Monitor, Receiver|⟩, ⟨|r$_3$, r$_4$|, |Tracker, Receiver|⟩};

- $k_1' \circ k_1 = \varnothing$;
- $k_2' \circ k_2 = \varnothing$;

Hence, we can see that $f_1' \circ f_1 = f_2' \circ f_2$, $g_1' \circ g_1 = g_2' \circ g_2$, $h_1' \circ h_1 = i_2' \circ i_2$, $j_1' \circ j_1 = k_2' \circ k_2$. As $\Phi_1' \circ \Phi_1 = \langle f_1' \circ f_1, g_1' \circ g_1, h_1' \circ h_1, i_1' \circ i_1, j_1' \circ j_1, k_1' \circ k_1 \rangle$ and $\Phi_2' \circ \Phi_2 = \langle f_2' \circ f_2, g_2' \circ g_2, h_2' \circ h_2, i_2' \circ i_2, j_2' \circ j_2, k_2' \circ k_2 \rangle$, we have $\mathbf{\Phi_1' \circ \Phi_1 = \Phi_2' \circ \Phi_2}$.

□

A.4 Composition details for the Surveillance application using RBS

This section presents the composition details for the Surveillance organization composed with the RBS organization as presented in Section 6.2.5.

Let Φ_1: $Shared_0 \rightarrow$ Surveillance be an organization homomorphism such that Surveillance is defined as shown in Figure 6.15 and $Shared_0 = \langle G_0, ET_0, EG_0, g\_root, R_0, P_0, participant_0, achieves_0\rangle$ is an organization such that:

- $G_0 = \{g\_root , g_1, g_2, g_3, g_4\}$,
- $ET_0 = \varnothing$,
- $EG_0 = \{ |g_1,g_2|, |g_3,g_4|\}$
- $R_0 = \{ r_1, r_2, r_3, r_4\}$
- $P_0 = \{p_1, p_2\}$
- $participant_0 = \{\langle p_1, \langle r_1, r_2\}\rangle, \langle p_2, \langle r_3, r_4\}\rangle\}$
- $achieves_0 = \varnothing$

We define $\Phi_1 = \langle f_1, g_1, h_1, i_1, j_1, k_1 \rangle$ such that:

- $f_1 = \{\langle g\_root, g\_root\rangle, \langle \mathbf{g_1, Monitor\ Area}\rangle, \langle \mathbf{g_2, gx_1}\rangle, \langle \mathbf{g_3, Track\ Area}\rangle, \langle \mathbf{g_4, gx_2}\rangle\}$;
- $g_1 = \varnothing$;
- $h_1 = \{\langle |g_1, g_2|, |Monitor\ Area, gx_1|\rangle, \langle |g_3, g_4|, |Track\ Area, gx_2|\rangle\}$;
- $i_1 = \{ \langle \mathbf{r_1, Monitor}\rangle, \langle \mathbf{r_2, rx_1}\rangle, \langle \mathbf{r_3, Tracker}\rangle, \langle \mathbf{r_4, rx_2}\rangle\}$;
- $j_1 = \{\langle |r_1, r_2|, |Monitor, rx_1| \rangle, \langle |r_3, r_4|, |Tracker, rx_2|\rangle\}$;
- $k_1 = \varnothing$;

Let Φ_2: $Shared_0 \rightarrow$ RBS be an organization homomorphism such that RBS is defined as shown in Figure 6.14. We define $\Phi_2 = \langle f_2, g_2, h_2, i_2, j_2, k_2 \rangle$ such that:

- f2 = $\{\langle g\_root, g\_root\rangle, \langle \mathbf{g_1, gx_4}\rangle, \langle \mathbf{g_2, Compare\ Time}\rangle\}$;
- g2 = $\varnothing$;
- h2 = $\{\langle |g_1, g_2|, |gx_4, Compare\ Time|\rangle\}$;
- i2 = $\{ \langle r_1, rx_4\rangle, \langle r_2, Receiver\rangle\}$;
- j2 = $\{\langle |r_1, r_2|, |rx_4, Receiver|\rangle\}$;

- $k2 = \varnothing$;

Let Φ_1': Surveillance $\rightarrow$ Surveillance_RBS an organization homomorphism such that Surveillance_RBS is defined as shown in Figure 6.17. We have $\Phi_1' = \langle f_1', g_1', h_1', i_1', j_1', k_1' \rangle$ where:

- f_1' = {⟨g_root, g_root⟩, ⟨Surveillance, Surveillance⟩, ⟨Monitor, Monitor⟩, ⟨Monitor Area, Monitor Area⟩, ⟨Determine Coverage, Determine Coverage⟩, ⟨**gx_1, Compare Time**⟩, ⟨Track, Track⟩, ⟨Track Area, Track Area⟩, ⟨Divide Area, Divide Area⟩, ⟨**gx_2, Compare Time**⟩, ⟨Generate Reports, Generate Reports⟩, ⟨Define Area, Define Area⟩};
- g_1' = {⟨|Surveillance, Monitor| , |Surveillance, Monitor|⟩, ⟨|Monitor, Monitor Area|, |Monitor, Monitor Area|⟩, ⟨|Monitor, Determine Coverage|, |Monitor, Determine Coverage|⟩, ⟨|Surveillance, Track| , |Surveillance, Track|⟩, ⟨|Track, Track Area|, |Track, Track Area|⟩, ⟨|Track, Divide Area|, |Track, Divide Area|⟩, ⟨|g_root, Surveillance| , |g_root, Surveillance|⟩, ⟨|Surveillance, Generate Reports|, |Surveillance, Generate Reports|⟩, ⟨|Surveillance, Define Area|, |Surveillance, Define Area|⟩}
- h_1' = {⟨ |**Monitor Area, gx_1**|, |**Monitor Area, Compare Time**|⟩, ⟨monitor, monitor⟩, ⟨ |**Track Area, gx_2**|, |**Track Area, Compare Time**|⟩, ⟨track, track⟩, ⟨startMonitor, startMonitor⟩, ⟨startTrack, startTrack⟩};
- i_1'= {⟨Monitor, Monitor⟩, ⟨Coverage Processor, Coverage Processor⟩, ⟨**rx_1, Receiver**⟩, ⟨Tracker, Tracker⟩, ⟨Divider, Divider⟩, ⟨**rx_2, Receiver**⟩, ⟨User Interface, User Interface⟩};
- j_1' = {⟨|Monitor, rx_1|, |Monitor, Receiver|⟩, ⟨|Tracker, rx_2|, |Tracker, Receiver|⟩, ⟨sendData, sendData⟩};
- k_1' = {⟨|Monitor, Monitor Area|, |Monitor, Monitor Area|⟩, ⟨|Coverage Processor, Determine Coverage|, |Coverage Processor, Determine Coverage|⟩, ⟨|Tracker, Track Area|, |Tracker, Track Area|⟩, ⟨|Divider, Divide Area|, |Divider, Divide Area|⟩, ⟨|User Interface, Generate Reports|, |User Interface, Generate Reports|⟩, ⟨|User Interface, Define Area|, |User Interface, Define Area|⟩}

Let Φ_2': RBS $\rightarrow$ Surveillance_RBS an organization homomorphism such that $\Phi_2' = \langle f_2', g_2', h_2', i_2', j_2', k_2' \rangle$ where:

- f_2' = {⟨g_root, g_root⟩, ⟨RBS, RBS⟩, ⟨Compare Time, Compare Time⟩, ⟨Compare Time, Compare Time⟩, ⟨Exchange Beacon Time, Exchange Beacon Time⟩, ⟨**gx$_4$**, **Monitor Area**⟩, ⟨Broadcast Beacon, Broadcast Beacon ⟩}
- g_2' = {⟨|g_root, RBS|, |g_root, RBS|⟩, ⟨|RBS, Compare Time|, |RBS, Compare Time|⟩, ⟨|RBS, Broadcast Beacon|, |RBS, Broadcast Beacon|⟩, ⟨|Compare Time, Compare Time|, |Compare Time, Compare Time|⟩, ⟨|Compare Time, Exchange Beacon Time|, |Compare Time, Exchange Beacon Time|⟩}
- h_2' = {⟨|gx$_4$, Compare Time|, |Monitor Area, Compare Time|⟩}
- i_2'={⟨Receiver, Receiver⟩, ⟨Reference, Reference⟩, ⟨**rx$_4$, Monitor**⟩, ⟨Beacon,Beacon⟩}
- j_2' = {⟨|**rx$_4$, Receiver|,** |**Monitor, Receiver|**⟩, , ⟨Exchange, Exchange⟩}
- k_2' = {⟨|Receiver, Compare Time|, |Receiver, Compare Time|⟩, ⟨|Reference, Exchange Beacon Time|, |Reference, Exchange Beacon Time|⟩, ⟨|Beacon, Broadcast Beacon|, |Beacon, Broadcast Beacon|⟩}

Surveillance_RBS along with homomorphism Φ_1' and Φ_2' represent the pushout of organization $Shared_0$ with homomorphism Φ_1 and Φ_2. In fact, we have:

- $f_1' \circ f_1$ = {⟨g_root, g_root⟩, ⟨**g$_1$, Monitor Area**⟩, ⟨**g$_2$, Compare Time**⟩, ⟨**g$_3$, Track Area**⟩, ⟨**g$_4$, Compare Time**⟩}
- $f_2' \circ f_2$ = {⟨g_root, g_root⟩, ⟨**g$_1$, Monitor Area**⟩, ⟨**g$_2$, Compare Time**⟩, ⟨**g$_3$, Track Area**⟩, ⟨**g$_4$, Compare Time**⟩}
- $g_1' \circ g_1 = \varnothing$;
- $g_2' \circ g_2 = \varnothing$;
- $h_1' \circ h_1$ = {⟨ |g$_1$, g$_2$|, |Monitor Area, Compare Time|⟩, ⟨ |g$_3$, g$_4$|, |Track Area, Compare Time|⟩};
- $h_2' \circ h_2$ = {⟨ |g$_1$, g$_2$|, |Monitor Area, Compare Time|⟩, ⟨ |g$_3$, g$_4$|, |Track Area, Compare Time|⟩};

- $i_1' \circ i_1 = \{ \langle \mathbf{r_1, Monitor} \rangle, \langle \mathbf{r_2, Receiver} \rangle, \langle \mathbf{r_3, Tracker} \rangle, \langle \mathbf{r_4, Receiver} \rangle\}$;
- $i_2' \circ i_2 = \{ \langle \mathbf{r_1, Monitor} \rangle, \langle \mathbf{r_2, Receiver} \rangle, \langle \mathbf{r_3, Tracker} \rangle, \langle \mathbf{r_4, Receiver} \rangle\}$;
- $j_1' \circ j_1 = \{\langle |r_1, r_2|, |\text{Monitor, Receiver}| \rangle, \langle |r_3, r_4|, |\text{Tracker, Receiver}| \rangle\}$;
- $j_1' \circ j_1 = \{\langle |r_1, r_2|, |\text{Monitor, Receiver}| \rangle, \langle |r_3, r_4|, |\text{Tracker, Receiver}| \rangle\}$;
- $k_1' \circ k_1 = \varnothing$;
- $k_2' \circ k_2 = \varnothing$;

Hence, we can see that $f_1' \circ f_1 = f_2' \circ f_2$, $g_1' \circ g_1 = g_2' \circ g_2$, $h_1' \circ h_1 = i_2' \circ i_2$, $j_1' \circ j_1 = k_2' \circ k_2$. As $\Phi_1' \circ \Phi_1 = \langle f_1' \circ f_1, g_1' \circ g_1, h_1' \circ h_1, i_1' \circ i_1, j_1' \circ j_1, k_1' \circ k_1 \rangle$ and $\Phi_2' \circ \Phi_2 = \langle f_2' \circ f_2, g_2' \circ g_2, h_2' \circ h_2, i_2' \circ i_2, j_2' \circ j_2, k_2' \circ k_2 \rangle$, we have $\mathbf{\Phi_1' \circ \Phi_1 = \Phi_2' \circ \Phi_2}$.

□