Towards a Formal Reactive Autonomic Systems Framework using Category Theory

Heng Kuang

A Thesis

in

The Department

of

Computer Science and Software Engineering

Presented in Partial Fulfillment of the Requirements

for the Degree of Doctor of Philosophy (Computer Science)

at

Concordia University

Montreal, Quebec, Canada

August, 2013

CONCORDIA UNIVERSITY

SCHOOL OF GRADUATE STUDIES

This is to certify that the thesis prepared

By: **Heng Kuang**

Entitled: **Towards a Formal Reactive Autonomic Systems Framework using Category Theory**

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

DOCTOR OF PHILOSOPHY (Computer Science)

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

Signed by the final examining committee:

_____ Chair
Dr. M. Nokken

_____ External Examiner
Dr. P. Panangaden

_____ External to Program
Dr. A. Hamou-Lhadj

_____ Examiner
Dr. V. Haarslev

_____ Examiner
Dr. J. Paquet

_____ Thesis Co-Supervisor
Dr. O. Ormandjieva

_____ Thesis Co-Supervisor
Dr. J. Bentahar

Approved by _____
Dr. V. Haarslev, Graduate Program Director

August 12, 2013

_____
Dr. C. Trueman, Interim Dean
Faculty of Engineering & Computer Science

Abstract

Towards a Formal Reactive Autonomic Systems Framework using Category Theory

Heng Kuang, Ph.D.

Concordia University, 2013

Software complexity crisis is the main obstacle to further progress in IT industry, as the difficulty of managing complex and massive computing systems goes well beyond IT administrators' capabilities. One of the remaining options is autonomic computing, which helps to address complexity by using technology to manage technology in terms of hiding and removing low level complexities from end users.

Real-time reactive systems are some of the most complex systems that have become increasingly heterogeneous and intelligent. Thus, we want to add autonomic features to real-time reactive systems by building a formal framework, Reactive Autonomic Systems Framework (RASF), which can leverage specification, modeling and development of Reactive Autonomic Systems (RAS). With autonomic behavior, the real-time reactive systems are more self-managed to themselves and more adaptive to their environment.

Formal methods are proven approaches to ensure the correct operation of complex interacting systems. However, many current formal approaches do not have appropriate mechanisms to specify RAS and have not addressed well on verifying self-management behavior, which is one of the most important features of the RAS. The management of

evolving specifications and analysis of changes require a specification structure, which can isolate those changes in a small number of components and analyze the impacts of a change on interconnected components. Category theory has been proposed as a framework to offer that structure; it has a rich body of theory to reason about objects and their relations. Furthermore, category theory adopts a *correct by construction* approach by which components can be specified, proved and composed in the way of preserving their properties.

In the multi-agent community, agent-based approach is considered as a natural way to model and implement autonomic systems, as the ability of an autonomous agent can be easily mapped to the self-management behaviors in autonomic systems. Thus, many ideas from the Multi-Agent Systems (MAS) community can be adapted to implement the autonomic systems, such as the self-management behavior, automatic group formation, interfacing and evolution.

Therefore, in terms of achieving our research goal, we need to i) build an architecture and corresponding communication mechanism for modeling both reactive and autonomic behavior of the RAS, ii) formally specify the architecture, communication and behavior above using category theory, iii) design and implement the architecture, communication as well as behavior of the RAS model by the MAS approach with its implementation and iv) illustrate our RASF methodology and approach with case studies.

To my grandfather.

Acknowledgments

Firstly, I would like to express profound thanks to my esteemed supervisors, Dr. Olga Ormandjieva and Dr. Jamal Bentahar, for their insight as well as thoughtful guidance, help and constant encouragement through the stage of my thesis work and graduate study. Their good technical and financial support motivates me to work hard and produce a high quality result. This thesis would not have been possible without their support and help.

Secondly, I would like to express the deepest gratitude and respect to Dr. Stan Klasa, who gave me a lot of valuable advice and insight on category theory. I am also heartily thankful to all other members of our research group, who are Nassir Shafiei-Dizaji, Jinzi Huang and Noorulain Khurshid, for their devoted and hard work on our research project.

Thirdly, a warm thank you goes to all my examiners for their precious time to review my research proposal, doctoral seminar, thesis work and gave me a lot of helpful advice. A special thank goes to Dr. Joey Paquet and his student Emil Vassev for their suggestions on earlier stage of this research. Another special thank goes to Dr. Volker Haarslev as well as his student Araash Shaban-Nejad for the inspiration of using category theory. Dr. Abdelwahab Hamou-Lhadj, thank you for your valuable suggestions at my seminar.

Fourthly, I would like to thank Concordia University and Faculty of Engineering & Computer Science for offering me the precious opportunity as well as excellent academic environment to achieve my graduate studies and this thesis work.

Finally, I really appreciate my family members for their never ending understanding, encouragement and support. I could not have made it here without them.

Table of Content

List of Figures

List of Abbreviations

RAS ..... Reactive Autonomic System

RASF ..... Reactive Autonomic System Framework

MAS ..... Multi-Agent System

RAE ..... Reactive Autonomic Element

RAO ..... Reactive Autonomic Object

RAOL ..... Reactive Autonomic Object Leader

RAC ..... Reactive Autonomic Component

RACG ..... Reactive Autonomic Component Group

RACS ..... Reactive Autonomic Component Supervisor

RACGM ..... Reactive Autonomic Component Group Manager

ICL ..... Intelligent Control Loop

CAT ..... Category Theory

BDI ..... Belief-Desire-Intention

ADF ..... Agent Definition File

XML ..... Extensible Markup Language

Chapter 1: Introduction

This thesis presents the result of the research and practical work made towards the nature, specification, modeling, formalization and illustration of Reactive Autonomic Systems Framework (RASF). The thesis lays a ground for the RASF project that provides rigorous development of Reactive Autonomic Systems (RAS).

1.1 Research Motivation

Software complexity is the main obstacle to further progress in IT industry, as the difficulty of managing complex and massive computing systems goes well beyond IT administrators' capabilities. Although current software engineering methodologies and programming language innovations have extended the size as well as complexity of computing systems, only depending on those will not get IT industry through the present software complexity problem. One of the remaining options is autonomic computing, which helps to address complexity by using technology to manage technology in terms of hiding and removing low level complexities from end users [80, 38].

The term *autonomic* is derived from human autonomic nervous system that monitors heartbeat, blood pressure and body temperature without any conscious thought. This self-regulation and separation provides the ability for human beings to concentrate on high level objectives without managing specific details [58]. In a similar way, an autonomic computing system is able to manage itself by anticipating requirements and resolving problems with minimum human intervention. Thus, IT professionals can focus on business-oriented objectives instead of computing level tasks with implementation,

configuration and maintenance details [63].

We need to select a target system that can get benefit from applying the autonomic computing paradigm. Real-time reactive systems are some of the most complex systems; the complexity involved comes from their real-time as well as reactive characteristics: 1) involves concurrency; 2) have strict timing requirements; 3) must be reliable; 4) involves software and hardware components; 5) have become increasingly heterogeneous. Thus, we want to add autonomic features to real-time reactive systems by building a framework (RASF) that can leverage specification, modeling and development of the RAS. With autonomic behavior, real-time reactive systems are more self-managed to themselves and more adaptive to their environment; the RAS can simplify and enhance the experience of end-users through anticipating their needs in a complex, dynamic and uncertain environment.

1.2 Research Problems

As real-time reactive systems become more complex, testing and error-finding also become more difficult, especially for the autonomic behavior with self-management and self-evolving capabilities added. Race conditions in those systems are very difficult to be found by inputting sample data and checking if results are correct, as certain errors are time-based and only occur when processes send or receive data at particular time, in particular sequence or after learning. In order to find those errors by testing, all possible state combinations of the processes have to be executed, which are exponential in the number of states.

Formal methods are proven approaches to ensure correct operation of complex interacting systems, since a formal specification can be used to prove the properties of a system, check for particular types of errors, and it also can be used as an input for model checking. However, most of current formal approaches do not have an appropriate mechanism to specify RAS and have not addressed well on verifying emergent behavior (an emergent behavior can appear when a number of simple entities operate in an environment, forming more complex behaviors as a collective), which is one of the most important characteristics of the RAS. Figure 1 compares current formal methods for the specification of emergent behavior and more details about those formal methods and their comparison can be found in [51]. The management of analysis for changes requires a specification structure, which is able to isolate those changes within a small number of components and analyze the impacts of a change on interconnected components [185].

	Formal Basis	Visual Formalism	Adaptability to Programming	Tool Support	Modularity	Emergent Behaviour Verification
CSP	✓	×	×	✓	✓	×
WSCCS	✓	×	×	×	✓	✓
Temporal Logic	✓	×	×	×	✓	×
X-Machines	✓	✓	✓	×	✓	✓
Unity Logic	✓	×	×	×	✓	×
ASSL	✓	×	✓	✓	✓	×
PTN	✓	✓	×	×	✓	×
DESML	×	✓	✓	✓	✓	✓

Figure 1: Comparison of Formal Methods for Emergent Behavior Analysis [51]

The following describes some specification techniques which have been used to specify social, swarm, as well as emergent behavior [51]:

- WSCCS: it is a process algebra used to model social insects [170] and analyze non-linear aspects of social insects [156].
- X-Machines: they have been used to model cell biology [55, 56]; their modifications, such as Communicating Stream X-Machines [57], also have potential to specify swarms.
- Dynamic Emergent System Modeling Language (DESML) [82]: a variant of the UML that can be used to model emergent systems.
- Cellular Automata [119]: they have been used to model systems which exhibit emergent behavior.
- Artificial Physics [153]: it uses physics-based modeling to gauge emergent behavior and ensure formation flying as well as other constraints on swarms.

Category theory has been proposed as a framework to offer that structure; it also has been successfully used to provide composition primitives in both algebraic [189] and temporal logic [35] specification languages. Category theory has a rich body of theory to reason about objects and their relations (specifications as well as their interactions), and it is abstract enough for a wide range of different specification languages. Moreover, automation may be achieved in category theory, for example, the composition of two specifications can be derived automatically and the category of specifications follows some properties, such as co-completeness. Category theory for software specification has

adopted a *correct by construction* approach by which components are specified, proved, and composed in the way of preserving their properties [185].

Thus, the research problems targeted in this thesis are not only how to model reactive autonomic systems, but also how to formalize the RAS models using category theory constructs in terms of verifying autonomic properties and validating self-management behavior.

1.3 Research Goal and Objectives

According to the research problems we defined above, our research goal is to build a formal framework (RASF) which can leverage modeling, formal specification as well as development of the RAS. In order to achieve the research goal, we need to: a) Build an architecture and corresponding communication mechanism for modeling both reactive and autonomic behavior of the RAS; b) Formally specify the architecture, communication and behavior above using category theory as a formal method.

After having the RAS model and its formal specification, we should elaborate it to an instance model and implement it through a case study to support the value and feasibility of our research. In the multi-agent community, agent-based approach is considered as a natural way to model and implement autonomic systems, as the ability of an autonomous agent can be easily mapped to the self-management behaviors in autonomic systems. Moreover, the ability of Multi-Agent Systems (MAS) to make interactions among components explicit and control them in a flexible way supports a more distributed way [167]. Many ideas from the MAS community can be adapted to implement autonomic

systems, such as self-management behavior, automatic group formation, knowledge mining, agent coordination, agent adaptation, interfacing and evolution [190].

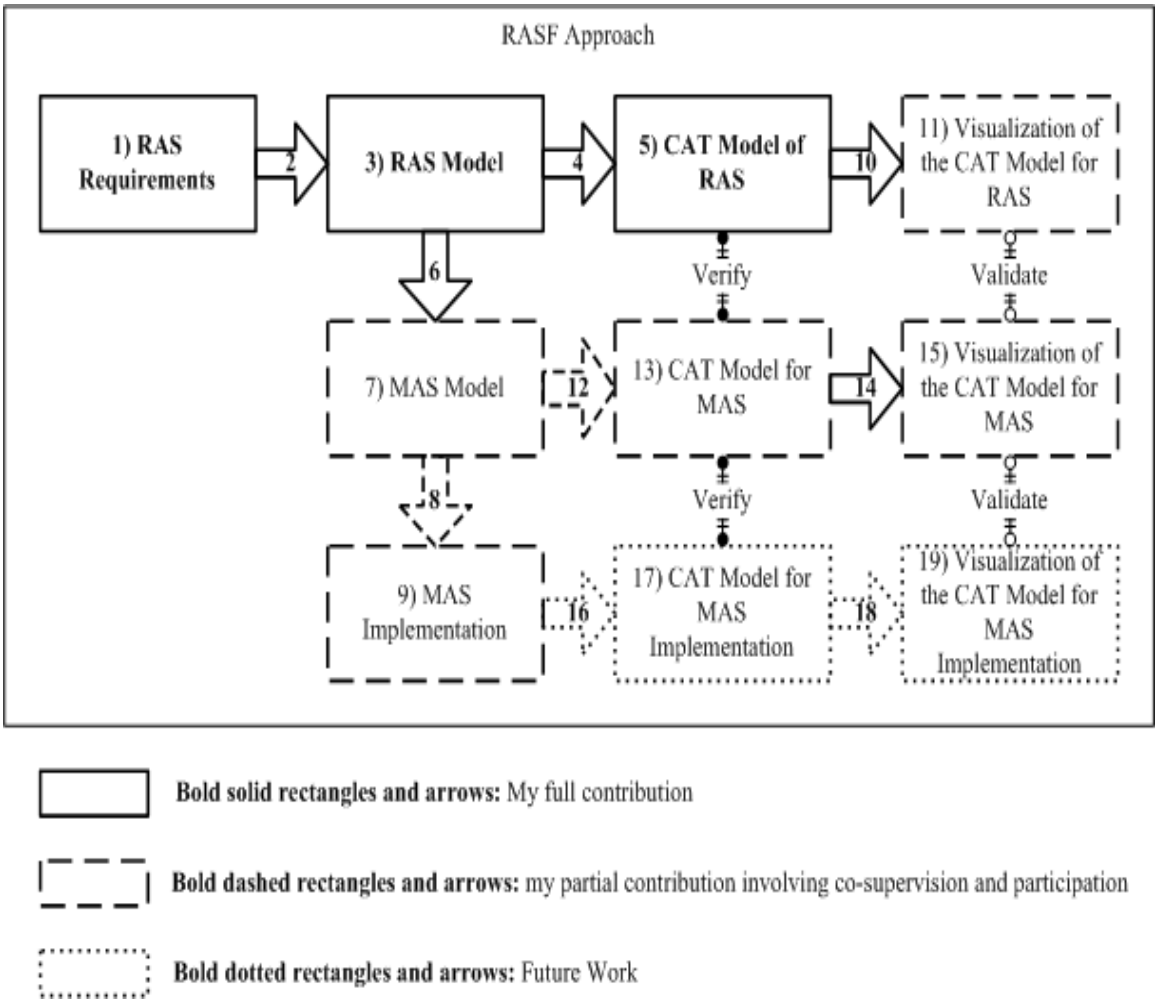


Figure 2: RASF Approach

Therefore, in terms of achieving our research goal, we also need to: c) design and implement the architecture, communication and behavior of the RAS model by a MAS approach; d) illustrate our methodology with a case study. Figure 2 depicts a perspective of our research methodology: 1) Build a RAS model based on the RAS requirements and properties; 2) Transfer the RAS model to its CAT (category theory) model using the category constructs; 3) Transfer the RAS model to its MAS model using the agent

constructs; 4) Transfer the MAS model to its CAT model; 5) Implement the MAS model using the Jadex framework, which allows for programming software agent in XML, Java and can be deployed on different kinds of middleware such as JADE; 6) Visualize the CAT mode of RAS model to its graphical representation in terms of the validation between categorical models; 7) Visualize the CAT model of MAS model to its graphical representation; 8) Transfer the MAS implementation to its CAT model; 9) Visualize the CAT model of MAS implementation to its graphical representation; 10) Apply the RASF approach to industrial projects in terms of supporting its feasibility.

1.4 Research Approach

As we started our research from scratch and it involves multiple fields (autonomic computing, multi-agent systems, category theory and real-time reactive systems), I did a comprehensive literature review on those fields before I started to develop our RASF approach. Figure 3 shows a road map of my research activities in order to achieve my contribution on the processes of 2, 4, 6, 10, 14 as well as corresponding outcome 1, 3, 5 illustrated in Figure 2. I also co-supervised and participated in the related processes of 8, 12 and corresponding outcome 7, 9, 11, 13, 15 that were conducted by the other three master students.

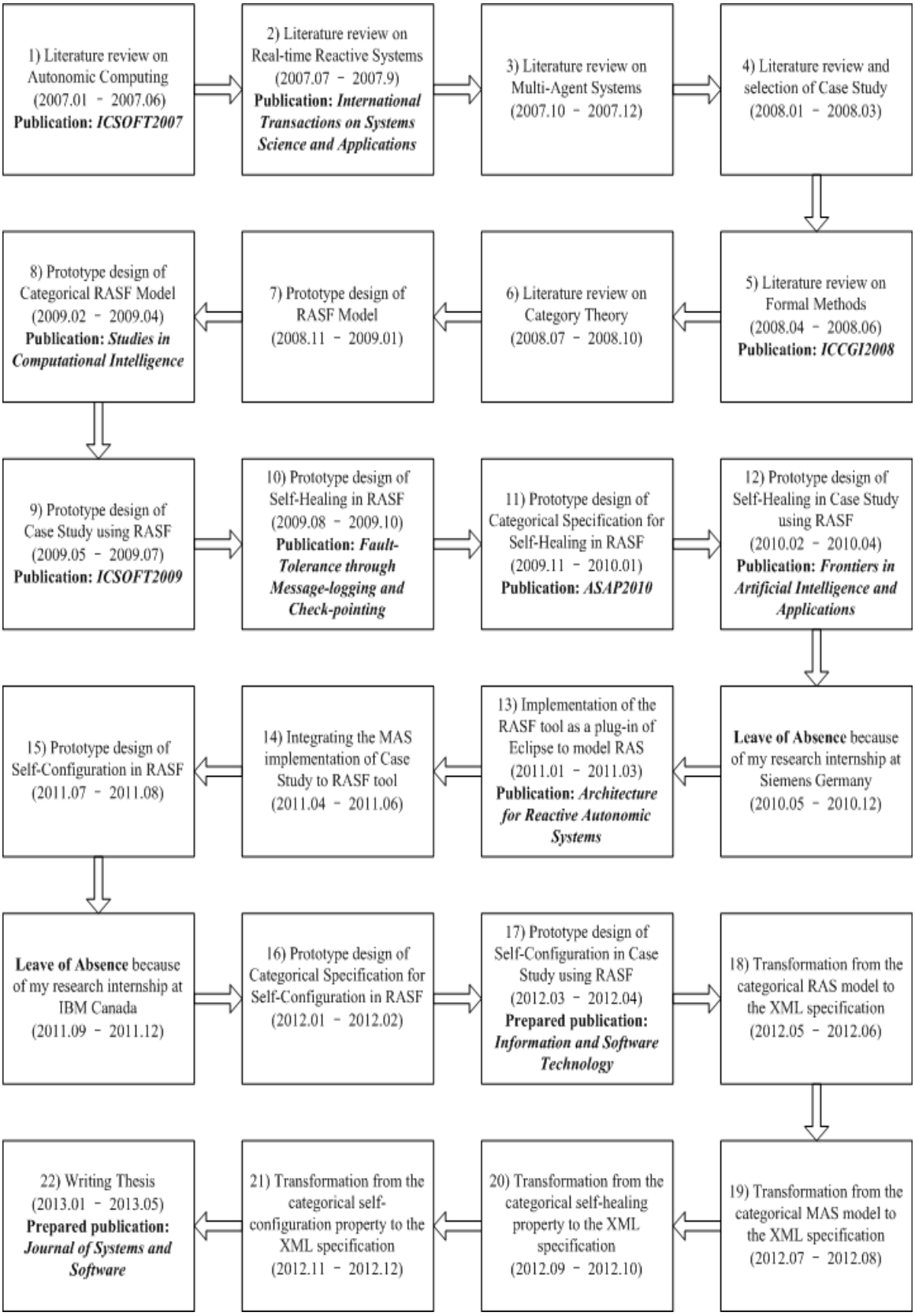


Figure 3: Road Map of Research Activities

1.5 Organization of the Thesis

This thesis is organized as follows:

Chapter 2 introduces the background as well as conceptual view of the autonomic computing paradigm. This chapter indicates some possible architecture perspectives and corresponding requirement specification for the RASF. We also introduced the agent-based computing technology that can be used to design and implement autonomic as well as reactive behavior for the RAS modeled by the RASF.

Chapter 3 presents the definitions, propositions and theorems of the category theory, which may be applied to specify reactive and autonomic behavior for the RAS, such as categories, morphisms, functors, limits, duality and naturality.

Chapter 4 gives an introduction to three case studies in terms of illustrating our research methodology and approach, which include Mars-world, Prospecting Asteroid Mission and an industrial project End-to-End iFix Tool.

Chapter 5 provides a comprehensive conceptual view of RASF. This chapter intends to capture and convey the significant architectural decisions for further design as well as implementation of the RASF.

Chapter 6 describes the prototype design of categorical RASF model, transformation from the categorical RAS model to its XML specification as well as transformation from categorical MAS model to its XML specification.

Chapter 7 presents the prototype design of self-healing property, prototype design of the categorical specification for self-healing and transformation from the categorical self-

healing property to its XML specification.

Chapter 8 illustrates the prototype design of self-configuration property, prototype design of categorical specification for the self-configuration and transformation from the categorical self-configuration property to its XML specification.

Chapter 9 gives an introduction to the implementation of RASF Integration Tool (RASFIT) and the integration of the MAS implementation to RASFIT, which includes the Eclipse plug-in module, Enterprise Architect module, Jadex module, CATCanvas module and model transformation module.

Chapter 10 describes the prototype design of self-healing and self-configuration in case studies using the RASF.

Chapter 11 presents conclusions to be drawn from this thesis work and offers the directions of future work on the RASF.

Chapter 2: Autonomic Systems and Multi-Agent Systems

This chapter states the research activities 1), 2), 3), 5) in Figure 3, which are literature review on autonomic computing, real-time reactive systems, multi-agent systems as well as formal methods. I had three publications [124, 88 & 176] related to that review. Software complexity is the main obstacle to the further progress in IT industry, as the difficulty of managing complex and massive computing systems goes well beyond IT administrators' capabilities. This complexity is derived from the following aspects:

- The need to integrate heterogeneous software environments into one cooperated computing system, and to extend that billions computing devices connected to the Internet.
- The rapid stream of changing and conflicting requirements at runtime requires timely and decisive responses.
- As the growing uncertainty of software environments due to unpredictable, diverse and interconnected computing systems, it is very difficult to anticipate and design interactions among the elements of those systems.

Although current software engineering methodologies (such as spiral development, incremental development, rapid application development and extreme programming) as well as programming language innovations have extended the size as well as complexity of computing systems, only depending on those two solutions will not get IT industry through the present software complexity crisis due to those three aspects above. Thus, one of the remaining options is autonomic computing, which helps to address complexity

by using technology to manage technology, in terms of hiding and removing low level complexities from end users [80, 38].

2.1 Definition of Autonomic Systems

The term *autonomic* is derived from human autonomic nervous system that monitors heartbeat, blood pressure and body temperature without any conscious thought. This self-regulation and separation provides the ability for human beings to concentrate on high level objectives without managing specific details [58]. In a similar way, an autonomic computing system can manage itself by resolving problems with minimum human intervention. Thus, IT professionals can focus on business-oriented objectives instead of computing level tasks with implementation, configuration and maintenance details [63].

Autonomic computing is not a totally new technology, but a goal-oriented and holistic computing paradigm to develop computer systems. Thus, autonomic computing is not a conventional computer systems paradigm, but a visionary approach which groups existing technologies together to achieve a common goal [155, 109]. The main goal of autonomic computing is similar to that of pervasive computing, which is a computing paradigm to create embedded, fitting and natural systems in terms of using them without managing them [183, 101]. The holistic approach means that autonomic computing does not specify that technology will be used to achieve those goals, and any existing technology that presents the pervasiveness and self-management behavior can be considered as autonomic computing, such as grid computing [3, 19], middleware [23, 172,

40], databases [69, 100], networking [133, 12, 144] and peer to peer applications [103].

Several researchers have proposed definitions for autonomic computing since 2001 according to its original vision from Horn [58]. Kephart and Chess defines the primary goal of autonomic computing as self-management, which can be further decomposed into self-configuration, self-healing, self-optimization and self-protection [80]. In addition, self-adaptive [2], self-organization [33] as well as self-knowledge [169] have also been proposed to define autonomic computing.

In terms of establishing a standardized definition for autonomic computing, Lin, MacArthur as well as Leaney [101] propose an application of software engineering methodology (IEEE standard for a Software Quality Metrics Methodology [68]) to address the lack of commonly accepted and quantifiable definition. This methodology provides a systematic way to define software projects by analyzing and identifying their quality requirements, which can be verified, applied and validated at each stage of development lifecycle. Figure 4 shows a list of quality factors with their definitions, and the Quality Metrics Framework [68] for autonomic computing is depicted in Figure 5; the more details about the openness, anticipatory, self-awareness and context-awareness in that figure can be found in [101].

Quality Factor	Definition
Anticipatory	The autonomic computing systems must have a projection of the user needs and actions in the future.
Context-awareness	The autonomic computing systems must find and generate rules for how best to interact with neighboring systems.
Openness	The autonomic computing systems must function in a heterogeneous world and implement open standards.
Self-awareness	The autonomic computing systems must be aware of its internal state.
Self-configuring	The autonomic computing systems must adapt automatically to the dynamically changing environments.
Self-healing	The autonomic computing systems must detect, diagnose, and recover from any damage that occurs.
Self-management	The autonomic computing systems must free system administrators from the details of system operation and maintenance.
Self-optimizing	The autonomic computing systems must monitor and tune resources automatically.
Self-protection	The autonomic computing systems must detect and guard itself against damage from accidents, equipment failure, or outside attacks by hackers and viruses.

Figure 4: Quality Factors of Autonomic Computing [101]

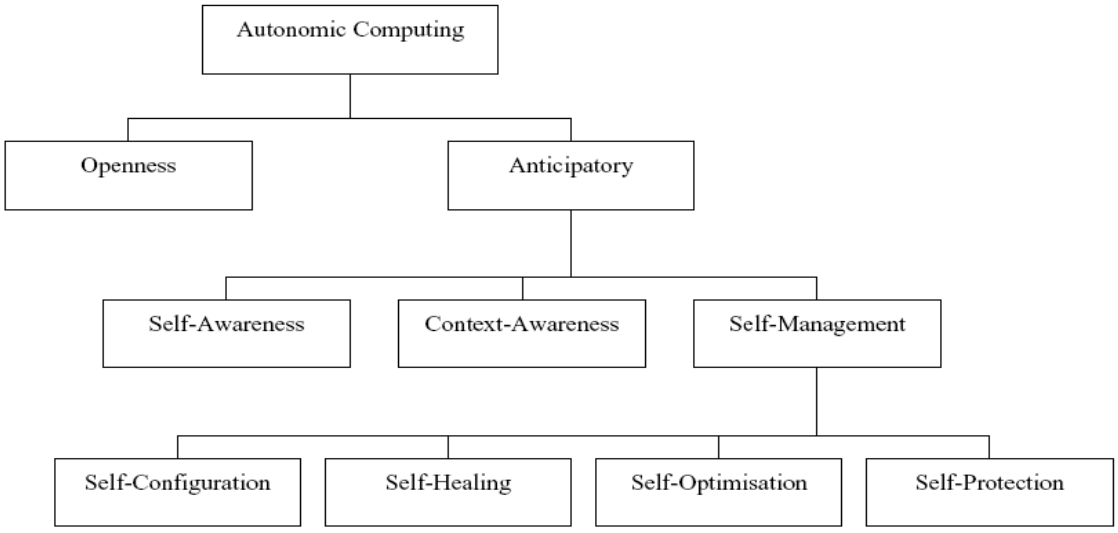


Figure 5: Quality Metrics Framework for Autonomic Computing [101]

2.2 Characteristics of Autonomic Systems

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

2.2.1 Self-Configuration

Autonomic computing systems are able to configure themselves automatically according to high level policies (business level objectives), which specify what is required instead of how they are implemented. For instance, after a new element joins, it automatically learns composition as well as configuration of the system and registers itself in terms of being used by other elements [80].

2.2.2 Self-Healing

Autonomic computing systems can detect, diagnose and repair bugs or failures in software as well as hardware. For example, a problem diagnosis element analyzes information from log files or monitors by using system knowledge, and then compares the diagnosis against system patches or alerts IT professionals. Finally, the system installs the appropriate patches followed by a regression test [80].

2.2.3 Self-Optimization

Autonomic computing systems are able to improve their operations and make themselves more efficient in performance or cost. For example, they can monitor, test and tune their parameters; they also can proactively upgrade their functions through finding, verifying, applying and validating the latest updates [80].

2.2.4 Self-Protection

Autonomic computing systems can defend the whole system against malicious attacks or cascading failures uncorrected by self-healing; they are also able to anticipate problems according to early reports from sensors and react to avoid or mitigate them [80].

2.3 Architecture of Autonomic Systems

The architecture for autonomic computing must reach the following requirements [63]:

- It should indicate external interfaces and behaviors of individual system elements.
- It must state how to integrate those elements so that they can cooperate toward system-wide self-management.
- It has to describe how to build systems by those elements in a manner that the system is autonomic as a whole.

The blueprint [63] organizes an autonomic computing system into building blocks connected by enterprise service bus patterns, which allow the elements to collaborate through standard mechanism, such as Web services. Figure 6 shows one example of composing those building blocks.

The lowest layer consists of managed resources that make up the IT infrastructure. Those resources can be hardware or software and may have embedded self-management features. More details can be found in [63].

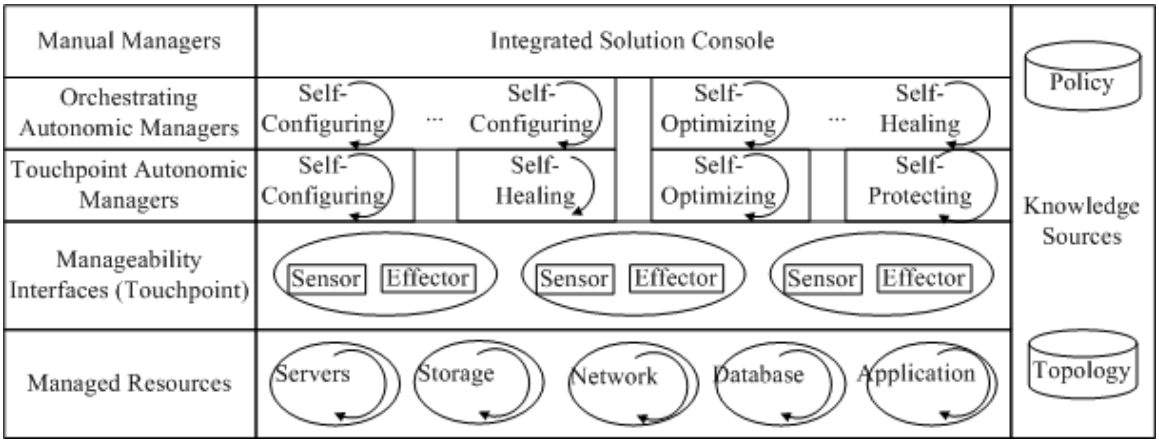


Figure 6: Autonomic Computing Reference Architecture [63]

Layer 2 contains standard manageability interfaces for accessing and controlling the managed resources in Layer 1 by the manageability endpoints.

Layer 3 and Layer 4 automate IT management processes by autonomic managers. A resource may have one or more managers in Layer 3, and each manager implements corresponding intelligent control loop (self-configuring, self-healing, self-optimizing as well as self-protecting).

Layer 4 consists of autonomic managers that orchestrate other managers in terms of delivering system-wide autonomic behavior by incorporating intelligent control loops with the perspective of overall IT infrastructure.

The top layer indicates a manual manager which provides a common system management interface for IT professional through an integrated solution console. The manual layer as well as autonomic manager layers obtain and share knowledge from knowledge sources. Therefore, resources and managers can collaborate to offer services and implement business processes.

2.3.1 Manageability Endpoint

Manageability Endpoint [63] is the component exposing states and management operations for a managed resource. It can communicate with an autonomic manager through the manageability interface. A manageability endpoint consists of a sensor for getting data from the resource and an effector for executing operations on the resource.

2.3.2 Autonomic Manager

Autonomic Manager [63] is the component which implements an intelligent control loop

as Figure 7 shows. *Monitor* is responsible for collecting information from a managed resource, such as status and metrics; *analyze* correlates those data aggregated in *monitor* and help the autonomic manager to learn IT environment and predicate future situations; *plan* provides the mechanisms for constructing actions to achieve desired goals based on policy information; *execute* controls the plan execution under the concern of dynamic environment. More details can be found in [63].

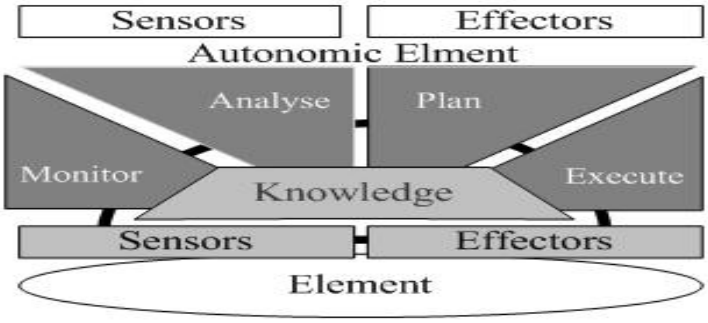


Figure 7: Intelligent Control Loop [63]

However, an IT administrator might delegate only certain parts of the intelligent control loop to an autonomic manager. Moreover, each autonomic manager also has a sensor and an effector as its interface to communicate with other autonomic managers.

2.3.3 Knowledge Source

Knowledge Source [63] is an implementation of the repository providing access to the knowledge, which consists of the management data with syntax and semantics, such as symptoms, policies, and plans.

The knowledge can be shared among autonomic managers through their sensors and effectors, and every autonomic manager is able to access the knowledge from one or

more knowledge sources. Moreover, the data used by the intelligent control loop, such as topology information, historical logs, and metrics also can be stored as knowledge.

2.3.4 Manual Manager

Manual Manager [63] is an implementation of the user interface which provides a mechanism for an IT professional to manually perform some management operations.

The manual manager collaborates with autonomic managers or other manual managers, and it involves a management console for the IT professional to delegate management operations to autonomic managers, such as configuration, monitoring and control.

2.4 Development of Autonomic Systems

There are presently seven core capabilities available for autonomic manager development [65]: 1) policy determination; 2) solution knowledge; 3) common system administration; 4) problem determination; 5) autonomic monitoring; 6) complex analysis; 7) transaction measurement.

2.4.1 Policy Determination

Policies are key part of the knowledge used by an autonomic manager to make decisions, since they contain the criteria for achieving goals or determining actions. Moreover, policies can control the planning components of an autonomic manager. Figure 8 shows the policy management in an autonomic component.

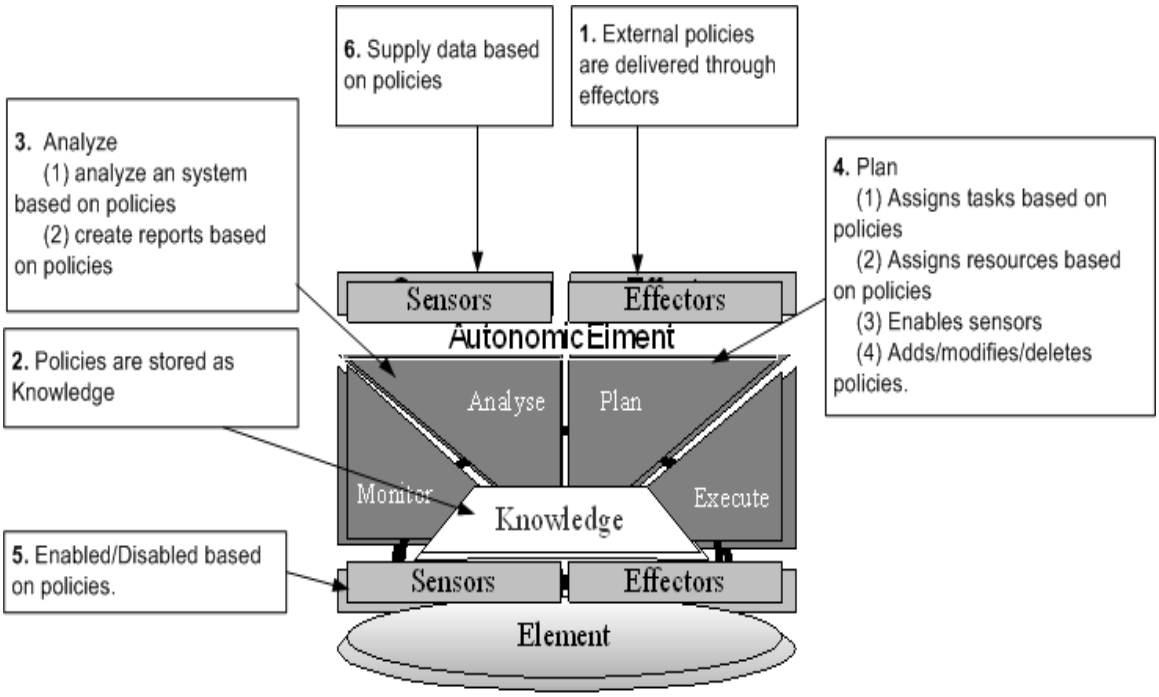


Figure 8: Policy Management in an Autonomic Element [26]

By defining policies in a standard way, they can be shared between autonomic mangers so that multiple subsystems can be managed in a similar manner.

2.4.2 Solution Knowledge

It contains many types of data coming from multiple points, such as operating systems, application languages, system utility and performance data. Common solution knowledge removes the complexity introduced by different formats and installation tools.

Moreover, the knowledge acquired in a consistent way can be used by autonomic managers in the contexts other than configuration, such as problem determination or optimization. In particular, solutions are combinations of platform capabilities (operating systems and middleware) as well as application elements. The idea is to acquire that information to support installation, configuration and maintenance at the solution level.

2.4.3 Common System Administration

It can be achieved by using a common console approach and consists of a framework for reuse as well as a consistent presentation of autonomic complex systems' properties.

According to the paper [67], the primary goal of a common console is to provide a single platform which can host all administrative operations of servers, software, and databases in a manner that allow users to manage solutions rather than managing individual systems or products. By increasing consistency of presentation and behavior across those administrative operations, the common console develops a familiar user interface which promotes reusing learned interaction skills instead of learning new and proprietary user interfaces.

2.4.4 Problem Determination

Autonomic managers take actions based on problems they find in managed elements. The first basic capability of an autonomic manager is to extract high quality data in terms of determining if a problem really exists, and the second one is to classify that problem.

2.4.5 Autonomic Monitoring

It enables an autonomic manager to filter, aggregate, and perform a complete analysis based on collected data in terms of detecting problems in systems when they happen. This capability includes [65]:

- A tool to gather information from sensors.
- A built-in data filtering mechanism.
- Pre-defined resource models and mechanisms for creating new models which enable

the description of a logical resource state.

- A tool to add policy knowledge.
- Analysis engines for basic cause analysis, server-level correlations across multiple complex systems, events and automate problem resolution.

2.4.6 Complex Analysis

Autonomic managers should be able to perform complex data analysis and reason based on large amount of data collected from managed resources by sensors. This data includes information about resource configuration, status, workload and throughput that is static or dynamic.

The tasks of common complex data analysis include classification, clustering data to characterize complex states and detect similar situations, prediction of workload and throughout based on past experience, reasoning for causal analysis as well as problem determination and optimization of resource configurations.

2.4.7 Transaction Measurement

It represents information based on the flow of interactions over an autonomic architecture. Autonomic managers need the transaction measurement capability which spans system boundaries to understand how the resources of heterogeneous systems combine into a distributed transaction environment. By monitoring that measurement, an autonomic manager can analyze and plan to change resource allocation for optimizing performance across those multiple systems based on policies; it can also determine some potential bottlenecks in the systems [65].

2.5 Multi-Agent Systems

This section states the research activity 3) in Figure 3. In the multi-agent community, agent-based approach is considered as a natural way to model autonomic systems, since the ability of an autonomous agent can be easily mapped to self-management behaviors in autonomic systems. In addition, the ability of a Multi-Agent System (MAS) to make interactions between components explicitly and control them in a flexible way supports a more distributed complexity [167].

Therefore, the MAS approach is well-suited for autonomic computing systems, and many ideas from the MAS community can be adapted to implement autonomic systems, such as self-management behavior, automatic group formation, agent coordination, evolution, agent adaptation, knowledge mining and interfacing [190].

2.5.1 Autonomous Agent

An agent is defined as a computer system which is capable of independent action on behalf of its user or owner, situated in a certain environment, and capable of autonomous actions in that environment to achieve its design objectives [191].

Agents have stronger notion of autonomy than objects in object-oriented paradigm, and they make decision for themselves whether they need to perform actions requested by another agent. Moreover, agents are able to control their internal states and own behavior; they experience environment through their sensors and act by effectors [191].

Agents also can communicate with other agents or users through certain agent communication languages. An agent is an agent with following properties [73]:

- Reactive: the agent should perceive its environment and respond in a timely way to the changes that occur in the environment;
- Proactive: the agent should not simply respond to its environment but be capable to show opportunistic along with goal-directed behavior and take the initiative where appropriate;
- Social: the agent can interact with other agents or users when appropriate to complete its problem solving and help others with their activities.

2.5.2 Definition of Multi-Agent Systems

A Multi-Agent System (MAS) is a software system possessing a number of autonomous agents which interact with one another and exchange messages through certain agent communication languages [191]. Therefore, those agents require the ability to cooperate, coordinate and negotiate with others in terms of successful interactions. The agents act on behalf of users with different goals as well as motivations, and the MAS can achieve its goals that are difficult to be reached by each individual agent. The characteristics of the MAS are [76]: 1) each agent has incomplete information or capabilities for solving problems; 2) there is no global system control; 3) data is decentralized; 3) computation is either asynchronous or synchronous.

The motivation for increasing interest in MAS research is due to their abilities such as the following [73, 159]:

- Solving problems that are too large for a centralized agent to solve because of resource limitations, performance bottlenecks or single-point of failures.

- Allowing for interconnection and interoperation of multiple existing legacy systems.
- Solving problems where data, expertise or control is distributed.
- Solving problems which can be naturally regarded as a society of autonomously interacting components or agents.

Thus, the system performance can be improved along the dimensions below [159]:

- Computational efficiency: concurrency of computation is exploited as long as communication is kept minimal, such as transmitting high level information instead of low level data.
- Reliability: by graceful recovery of component failures since agents with redundant capabilities or appropriate inter-agent coordination can be found dynamically, such as taking over the responsibilities of failed agents.
- Extensibility: the number and capabilities of the agents working on a problem can be changed.
- Robustness: by the system's ability to tolerate uncertainty.
- Maintainability: the modularity by composing a system with multiple agents.
- Responsiveness: the modularity can handle exceptions locally instead of spreading them to the whole system.
- Flexibility: agents with different capabilities can adaptively organize to solve problems.
- Usability: because functionally specific agents can be reused in different agent teams to solve various problems.

2.5.3 Agent Interactions

For designing the MAS, we need to implement micro and macro designs that are agent design as well as society design respectively [191]. In the agent design, the focus is to build agents which are capable of independent and autonomous actions; in the society design, the task is to establish interaction capabilities of those agents, such as cooperation, coordination and negotiation, particularly when certain conflicts arise between them [73].

Agent interactions are guided by cooperation strategies to improve their collective performance. The early work on distributed planning took the approach of complete planning before actions, so the agents must be able to recognize sub-goal interactions and either avoid them or resolve them [73]. For example, the authors in [39] propose a synchronizer agent to recognize and resolve those interactions; other agents send their plans to the synchronizer who examines the plans for critical aspects.

The notion of the interactions between self-interested agents has been focused on negotiation, which is the presence of some conflict forms that must be resolved in a decentralized manner by the self-interested agents through bounded rationality and incomplete information. In addition, those agents communicate and iteratively exchange proposals as well as counter-proposals [73].

The negotiation is considered as a method for coordination and conflict resolution, such as resolving goal inconsistencies while planning, resolving conflicts in resource allocation and resolving task disparities when determining organizational structures [73].

Another important aspect of successful interaction for self-interested agents is the

capability to adapt their behavior in terms of changing environment. The authors in [61] describe an agent's belief process for conjectures about the effect of their actions. A conjectural equilibrium is defined where all agents' expectations are realized, and each agent responds to its expectations optimally. The authors also present a multi-agent system in which an agent builds a model of other agents' response [73].

2.5.4 Agent Communication Languages

Agent Communication Language (ACL) is one of proposed languages for communicating agents, and most of ACL are based on the speech-act theory that is expressed by standard keywords known as performatives [179]. There are two main ACL:

- Knowledge Query and Manipulation Language (KQML): proposed by DARPA Knowledge Sharing Effort (KSE). KQML is the notion of performative keywords such as ask-if, tell, and ask-one.
- Foundation for Intelligent Physical Agents' Agent Communication Language (FIPA-ACL): FIPA-ACL message structures are defined by FIPA Agent Communication Standards as Figure 9 shows.

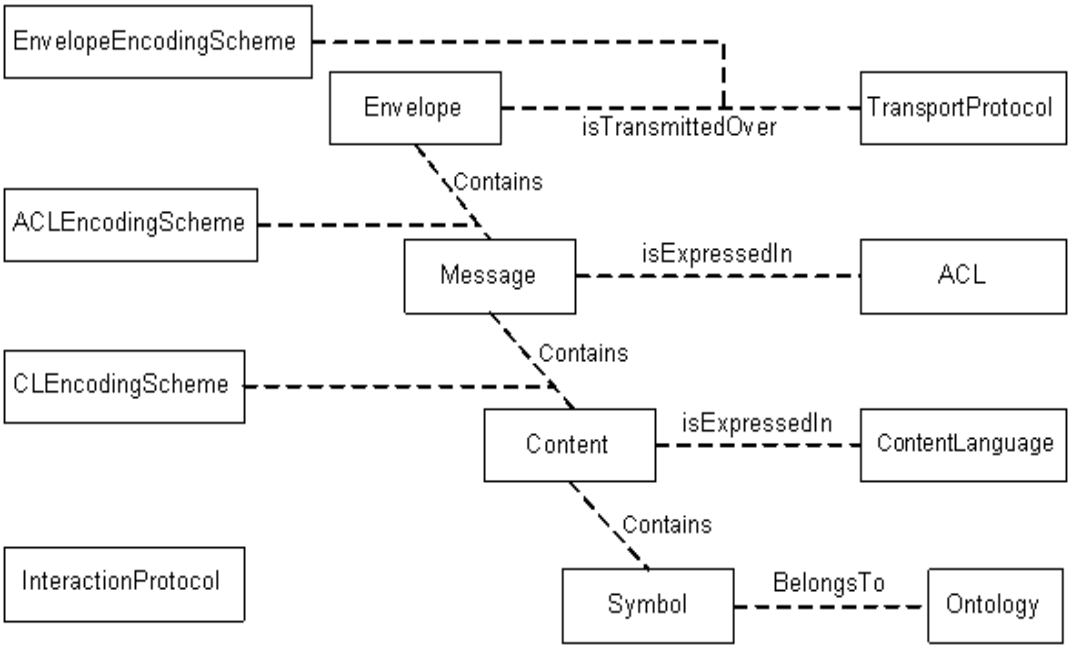


Figure 9: FIPA Standard: Components of Communication Model [181]

A FIPA-compliant message is the fundamental form of communication among agents that consists of Envelope, Payload, Message, and Content as Figure 10 shows.

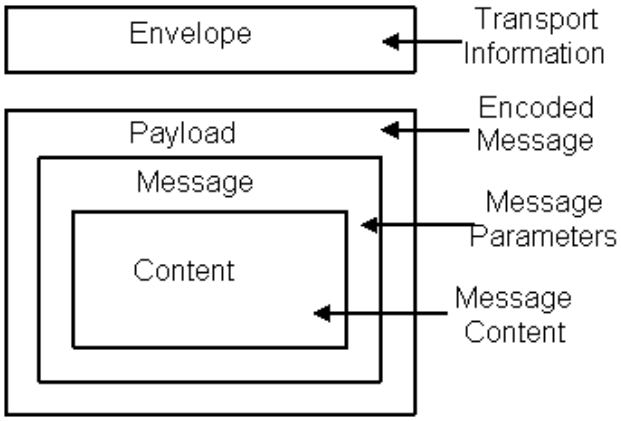


Figure 10: FIPA Message Structure [181]

2.5.5 Agent Architecture

According to the paper [108], agent architecture specifies how the agent can be decomposed into a set of component modules and how these modules communicate with

each other. Typically, the author in [191] identifies three categories for single agent architectures as the following:

- Deliberative agent architecture: an agent contains an explicitly represented symbolic model of its environment, and it makes decisions through symbolic reasoning. There are three types of reasoning agents: symbolic reasoning agents, deductive reasoning agents and practical reasoning agents. Belief-Desire-Intention (BDI) architecture is one of the main deliberative agent architectures.
- Reactive agent architecture: an agent acts based on stimulus-response rules and it does not symbolically represent its environment. In this architecture, agents are able to maintain ongoing interactions with their environment and respond to the changes in it [192]. The architecture in [22] is a good example of reactive agent architecture that considers agent properties, capabilities, and environment.
- Hybrid agent architecture: an agent can act both deliberatively and reactively. In this architecture, agent designers can build an agent out of two or more subsystems: one is the deliberative agent containing a symbolic model that can develop plans and make decisions in the way proposed by symbolic agents; another one is the reactive agent which is capable of reacting to events without complex reasoning.

The BDI architecture is a philosophical model for describing rational agents [136], and it contains specific denotation of Beliefs, Desires as well as Intentions [17]. The architecture addresses how those Beliefs, Desires and Intentions are represented, updated and processed. In the BDI architecture, agents with particular mental attitudes can choose

appropriate actions based on their capabilities and internal structures.

Beliefs indicate how agents know their surroundings that include themselves and other agents. The Beliefs also include inference rules which allow forward chaining to new beliefs, and the information is stored in a database called belief base. Unlike knowledge, the Beliefs may be not true [181].

Desires are goals that agents would like to achieve [149], such as finding the best price or becoming rich, and they are the motivational state of those agents. The difference between desires and goals is that a set of goals must be consistent, but desires may be inconsistent.

Intentions are the targets of agents, and they indicate what the agents have chosen to do, which represent the deliberative state of those agents. In an implemented system, the Intentions describe an executing plan that is a sequence of actions performed by an agent to achieve one or more intentions. Plans are only partially with details being added during their process [181]. The BDI approach consists of the following components [193]:

- A philosophical component for the theory of human rational actions.
- A software architecture component used in a number of complex applications.
- A logical component for the BDI logics.

When new information arrives, agents can update their beliefs or desires. The new beliefs or desires can trigger certain actions, but only one intended action is selected and activated. After executing that action, the intentions of those agents are updated, and the new beliefs or desires are stored. Finally, a new cycle of the BDI model execution starts.

2.5.6 Agent Oriented Programming

Agent oriented programming has revealed a great potential to develop complex computer applications by agent technologies. There are a number of approaches and methodologies for agent-based programming, such as Jason [117], 3APL [54], JADE [8], and Jadex [131].

Java Agent Development Framework (JADE) is one of the most widely used agent oriented middleware today, and it provides a FIPA-compliant agent platform as well as a package for developing Java agents. The JADE is the open-source software which has been under development since 1999 by Telecom Italia Labs. The internal architecture of the JADE fully complies with FIPA standard, and it provides a basic set of functionalities that are considered as essential for autonomous agents [181].

Jadex is also a Java-based and FIPA-compliant agent environment, but it allows modeling goal-oriented agents according to the BDI architecture. In the abstract Jadex architecture [131], an agent can receive and send messages. The received messages or goal events can trigger the internal reaction and deliberation mechanism of the agent, which dispatches those events to the plans selected from a plan base. Running plans may access and modify a belief base, exchange messages with other agents, create new goals and trigger internal events again [181].

Belief base: stores a set of beliefs that make up the knowledge of an agent. Unlike other BDI-based multi-agent systems, which beliefs are represented by certain kind of first-order predicate logic or relational models, the beliefs in Jadex is a storage of

knowledge as a database for an agent. Those beliefs cannot support any inference mechanism, and there are several advanced features on top of the belief representation. Jadex uses an Object Query Language (OQL), which is a kind of query language adopted from object-relational database world, to search the conditions that can trigger plans or goals when certain beliefs change. In addition, the beliefs also can be stored as expressions and evaluated dynamically on demand [181].

Goal structure: goals in Jadex are not just a special kind of event as those in pure BDI-based multi-agent systems but a central concept. An agent can engage into some actions for its goals until they have been achieved, unreachable or undesired. A goal lifecycle consists of the following states [131]: option, active and suspended, which can distinguish between just adopted and actively pursued goals. When a goal is adopted, it becomes an option added to the desire structure of the agent, and application specific goal deliberation mechanisms are responsible for managing the state transitions of all adopted goals. There are four types of goals that extend the general lifecycle and exhibit different behavior regarding to their processing as the following [181]:

- Achieve goal: defining a desired target state without specifying how to reach it.
- Maintain goal: specifying a state which should be kept once it is achieved.
- Perform goal: stating that something should be done but may not necessarily lead to any specific result.
- Query goal: representing a need for information.

Plan specification: plans are used to specify agents' actions to achieve their goals,

and Jadex uses a plan-library approach to represent the agents' plans, which are written in Java and predefined by developers. Those plans are instantiated in terms of handling events, achieving goals, building action libraries for the agents as well as providing all flexibilities of the Java programming language. The plans consist of two parts that are a plan head and a plan body; the plan head defines the circumstances under which the plan body is instantiated and executed. Based on the current circumstance, plans are selected automatically in response to occurring events or goals by the system [181].

Agent definition: the complete definition of an agent is captured in a XML file called Agent Definition File (ADF). The ADF consists of beliefs, goals, events, plans and other agent elements; it can be regarded as a type specification for a class of instantiated agents. Plans are declared by specifying how to instantiate them from the Java class. In addition, the initial state of an agent can be determined in a configuration tag that defines initial beliefs, goals and plans. In Jadex, the ADF is loaded first to start an agent; that agent can be initialized by the configuration tag [181].

Execution model: before incoming messages in a message queue can be forwarded to the system, it has to be assigned a capability of handling those messages. If a message belongs to an ongoing conversation, an event for the incoming message is created in the capability of executing that conversation, and the created event can be added to the global event list of an agent; otherwise, an appropriate capability has to be found. Moreover, there is a dispatcher in the execution model that is responsible for selecting applicable plans for those events from the event list. Jadex can provide flexible settings to influence

the event processing based on the different event types and instances. The messages are posted to one plan, and many plans are executed for one goal. After the plans have been selected, they are placed in a ready list and wait for execution which is performed by a scheduler [181].

2.5.7 Formal Methods for Multi-Agent Systems

When there is critical safety as well as security issues involved, informal analysis is not adequate to ensure software qualities. Instead of using natural language with inherent vagueness and ambiguity, formal notations can provide a means for precise specification, which has been concerned with the description of a software design and its properties in a mathematical logic or other formal notations [104]. In order to understand properties of the systems containing multiple actors, powerful modeling and reasoning techniques are necessary to capture potential evolutions of the systems, especially when agents or agent systems are to be modeled and analyzed computationally [105].

Formal methods for agent systems attempt to represent and understand properties of the systems by using logical formalisms to describe both mental states of the agents and possible interactions in systems. Those logics of beliefs and temporal modalities require efficient as well as rigorous theorem-proving or model-checking algorithms, which can test, debug and verify the properties of multi-agent systems before their implementation phase [105].

For the design of self-*, a programming paradigm that can support automated checking of both functional and non-functional system properties may be needed. This

would lead to certify agent components for correctness in terms of their specifications. Moreover, techniques are needed to ensure that the systems execute in an acceptable and safe manner during the adaptation processes, such as using high level contracts, invariants or dependency analysis to monitor system correctness before, during and after adaptations [105].

2.6 Related Work

2.6.1 Self-Management Properties

This subsection states the research activity 1) in Figure 3. The difficulties of dynamic software reconfiguration are examined in [184]. The authors concluded that both static structure and run time behavior must be captured in terms of defining the workable reconfiguration model.

An autonomic approach to network service deployment that scales to large and heterogeneous networks is explored in [47]. The paper introduces a two-phase intelligent network service deployment: 1) a macro-level operating in a hierarchical distributed way to query and collect the capabilities of the nodes in network; 2) a micro-level refining installation based on custom capabilities of each network component.

The authors in [16] claims that cooperative negotiation using incremental elicitation is required to perform resource allocation in a distributed autonomic system. The paper presents algorithms for computing mini-max regret and two elicitation strategies. They use an automated resource manager which can allocate resources for workload managers in order to maximize total organizational utility and then solve the resource allocation

problem.

Self-configuration is concerned with physical design, deployment and their static aspects in [129]. The project called LAMDA (Lights-out, Automated Management of Distributed Applications) is based on an adaptation of the Hierarchical Queuing Petri Nets to model the environment.

The authors in [48] present an architecture for process execution that has an autonomic controller. The controller provides self-tuning, self-configuration as well as self-healing capabilities in order to automatically configure a distributed service composition engine. The system has been designed so that its components can be dynamically replicated on several nodes of a cluster. In addition, through the controller, the engine can react to variations in workload through altering its configuration to achieve better performance. The controller can also heal the system in case of failures.

The authors in [64] describe a problem determination methodology and architecture that can standardize log format, content as well as organization. Moreover, the autonomic systems which implement self-healing are based on the common problem determination architecture to identify problems and implement solutions.

The authors in [92] propose an approach to implement self-healing according to the "resource model" concept and System Management Ontology for representing Common Information Model constructs.

The authors in [178] present a self-healing method which automates the mirroring and replications in a network of servers. Moreover, the paper describes a design based on

a self-configuration mesh of computers and a communication mechanism between those nodes that operate on a rooted spanning tree.

The authors in [197] describe adaptive components as a framework for component-based development. An adaptive component has multiple implementations, and each one is optimized for a particular request workload. The paper also claims that the dynamic switching between implementations at run-time will become a useful self-optimization tool for autonomic computing.

Statistical modeling, tracking as well as forecasting techniques borrowed from econometrics are explored in [143] to yield a predictive autonomic system that regulates its behavior in the anticipation of its needs. Moreover, the paper claims that the systems using Clockwork method can detect and forecast cyclic variations on future performance; they can use data to reconfigure themselves by anticipating their needs.

The authors in [1] describe a model-based control and optimization framework to design autonomic systems which continually optimize their performance by changing workload demands and operating conditions. The performance management problems of interest are posed as one of sequential optimization under uncertainty, and a look-ahead control approach is used to optimize the forecast system behavior over a limited prediction horizon. The basic control concepts are then extended to tackle distributed systems where multiple controllers must interact with each other to ensure overall performance goals.

The authors in [154] present a general architecture to build self-optimization services.

Their framework can support both dynamic composition of service configurations and runtime adaptation of configuration according to changes in a system or requirements. The authors design a recipe representation which can be used by developers to capture their service-specific knowledge. The recipe is used by a generic runtime infrastructure to realize initial service configuration and adaptation. The runtime infrastructure includes a synthesizer that constructs an abstract service configuration and maps it to physical nodes, an adaptation manager which monitors the service and applies adaptation strategies, and an adaptation coordinator that resolves conflicts among those strategies.

2.6.2 Autonomic Systems Modeling

IBM Tivoli Management Suite provides a jump-start toward fulfilling the ultimate goal of a fully autonomic system. Figure 11 shows the coverage of IBM Tivoli Management Suite across the IBM portfolio of products and services.

Self-Configuration: Configuration Manager can be noticed when the software on a machine is not synchronized with a reference model, and it also creates a customized deployment plan for each machine in a cluster. Identity Manager automates user life cycles with native repositories. It communicates directly with access-system to create accounts, passwords and privileges. Storage Manager can automatically identify and load drivers for the storage devices connected to servers.

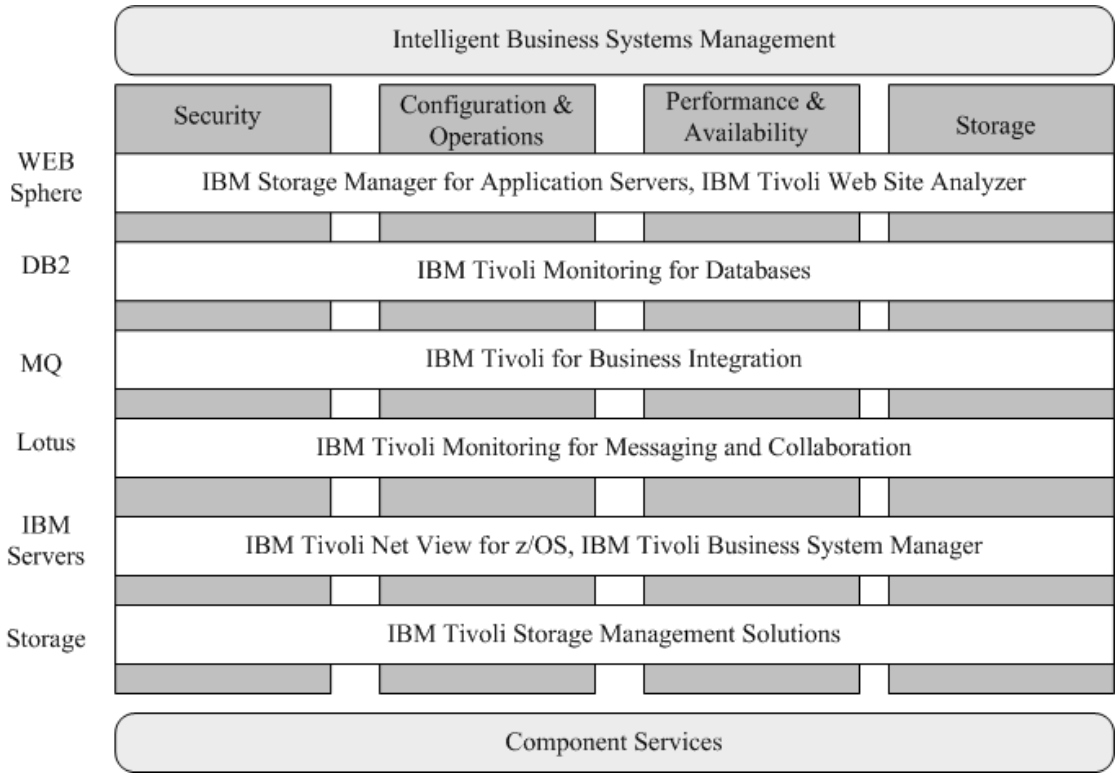


Figure 11: IBM Tivoli Management Suite across IBM Overall Architecture [118]

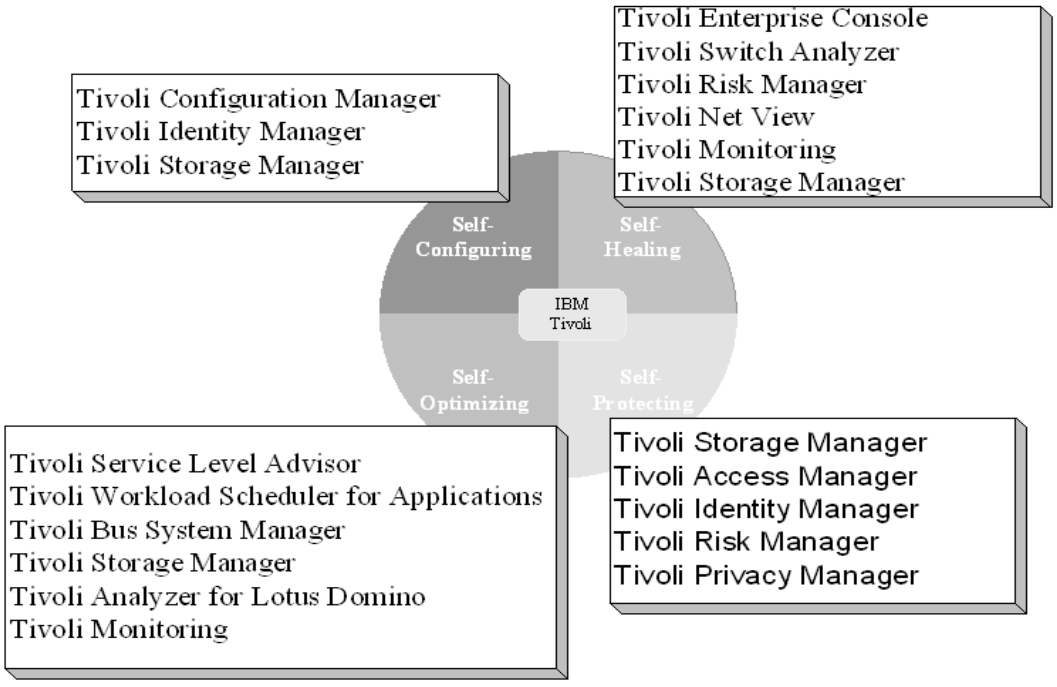


Figure 12: Tivoli Autonomic Software Products [118]

Self-Healing: Enterprise Console automatically inspects error logs, derives problem causes and initiates necessary actions. Switch Analyzer correlates network device errors to rot cases without human intervention. NetView is able to display network topologies, discover TCP/IP networks, correlate events, monitor network health and gather data. It has the router fault isolation which identifies causes of errors and consequently initiates corresponding actions. Monitoring for applications, databases as well as middleware can automatically discover, diagnose and initiate problem resolution. Risk Manager contains the self-healing technology which assesses security threats and automates responses for server reconfiguration, patch deployment as well as account revocation. Storage Resource Manager automatically notices storage problems and executes policy-based actions to solve those problems.

Self-Optimization: Service Level Advisor can perform trend analysis according to historical performance data and make predications on critical thresholds in the form of events sent to Enterprise Console. Workload Scheduler for Applications is able to monitor and automate workload executions. Monitoring for Transaction Performance enables the monitoring of performance and availability of transactions. Storage Manager supports adaptive differencing technology that can optimize resource usage for backup.

Self-Protection: Access Manager is able to prevent unauthorized access and control resources for authenticated users. Identity Manager centralizes identity management and integrates automated workflow of business processes. Risk Manager can provide system wide self-protection by assessing potential threats and automating responses.

N1 is the Sun Microsystems product for grid computing [157, 158]. It provides services to manage heterogeneous environment and removes information technology complexity by technical means. Sun Management Center in N1 grid cluster architecture is based on an intelligent agent reference model, where a manager can monitor and control managed entities by sending requests to agents residing on managed nodes, and those agents can collect management data on the behalf of the manager. Thus, Management Center uses autonomous agent technology to implement its autonomic capabilities, which includes powerful system administration tools, test and verification tools, as well as automated installation and deployment tools [158].

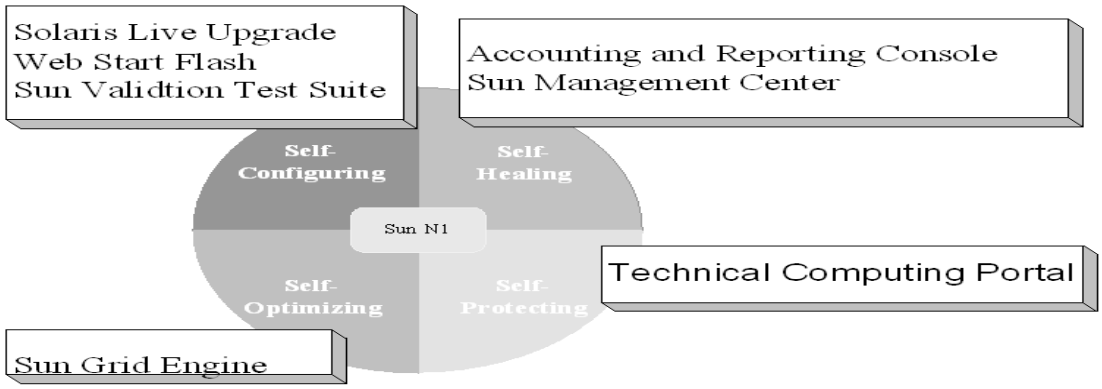


Figure 13: Sun N1 Autonomic Characteristics [158]

Self-Configuration: Solaris Live Upgrade and Web Start Flash provide automated installation and deployment technologies with which systems can be upgraded while they are running.

Self-Healing: Sun Management Center which is based on agent technologies can provide self-healing capability. Account and Reporting Console uses a comprehensive way to collect and analyzed detailed statics of usage on the Grid.

Self-Optimization: Sun Grid Engine distributed resource management software can optimize the utilization of both software and hardware resources within a heterogeneous networked environment.

Self-Protection: Technical Computing Portal provides a high-performance technical computing with a secure anytime and anywhere access to a single Web based point of delivery for services, content as well as complex applications through a standard Internet browser and a simple user interface.

Dynamic System Initiative is a Microsoft effort to incorporate into the Microsoft Windows platform a number of solutions which will ultimately implement autonomic characteristics [114]. Microsoft autonomic computing architecture is based on System Definition Model (SDM), which is used to create definitions for distributed systems, such as the definitions of resources, endpoints, relationships and subsystems. Moreover, the SDM contains deployment information, installation processes, as well as schemas for configuration, events, automation tasks, health models and operational policies.

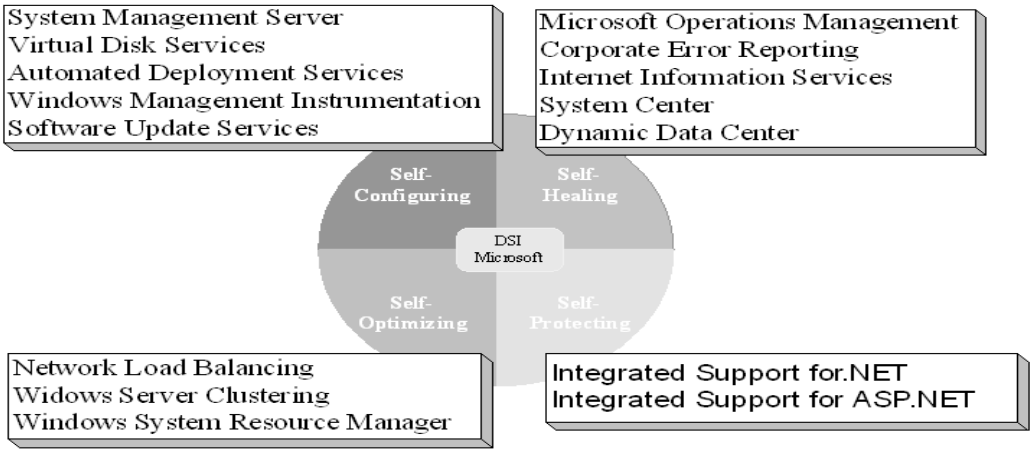


Figure 14: Microsoft DSI Autonomic Characteristics [114]

Self-Configuration: Virtual Disk Service provides a vendor independent interface to identify and configure storage devices from multiple vendors. Windows Management Instrumentation can provide direct and unified management tools locally or remotely for administrators. Software Update Services automatically deliver critical patches to target computers from a single Intranet. System Management Server can provide WAN-aware capability to reliable deployment of applications for thousands of workstations.

Self-Healing: Microsoft Operation Management incorporates event management, proactive monitoring and altering, reporting and trend analysis as well as system and application specific knowledge to improve manageability. Corporate Error Reporting can provide information about the problems in applications for vendors and developers. Internet Information Services is a Web server with self-healing that is supported by a new fault tolerant process model.

Self-Optimization: Network Load Balancing enhances scalability and availability of mission-critical, TCP/IP-based services, such as Web, Terminal Services, virtual private networking and streaming media servers. Windows System Resource Manager can help administrators to control how CPU resources are allocated to applications, servers and processes. It improves system performance, reduces interference among resources and creates a more predictable experience for users.

Self-Protection: Integrated Support for .NET as well as ASP.NET leverage a fully managed and protected application environment of Web along with XML services.

2.6.3 Real-Time Reactive Systems

This subsection states the research activity 2) in Figure 3. The authors in [147] present a formal framework for automatically recovering a class of reactive systems from run-time failures. That class of systems comprises the executions which can be divided into rounds so that each round performs a new unit of work. The paper also shows how the system recovery and repair problems can be modeled as an instance of online learning problems. The framework leverages parallelism to proactively explore the space of repairs before a failure is occurred.

The authors in [151] introduce the real-time reference architecture for autonomic computing where components implementing functions of real-time system elements or blocks, such as transducers, controllers and actuators are designed. Based on the design and implementation of that reference architecture, a self-adapting loop according to system-specific adaptation knowledge that includes the types and properties of autonomic components, behavior constraints as well as strategies for adaptation is proposed in [152]. The proposed system is an integral part of a real-time system which controls the behavior of computing environment and evaluating its global behavior through a mathematical description of time variation on the number of users in that system. According to the evaluation, the adaptive system can change the control structure of autonomic computing environment by replacing its controller with the one that matches corresponding user time variation law. Moreover, the elements of the self-adapting loop as well as the trade-off between additional overhead and autonomic computing processes are discussed in [152].

2.6.4 Multi-Agent Systems

This subsection states the research activity 3) in Figure 3. The authors in [167] present Unity, a decentralized architecture for autonomic computing based on multiple interacting agents called autonomic elements. The paper also illustrates how the Unity architecture achieves autonomic behavior, such as goal-driven self-healing and real-time self-optimization. In addition, they present a realistic prototype implementation that shows how a collection of Unity elements self-assembles, recovers from certain classes of faults and manages computational resources in a dynamic multi-application environment.

The authors in [98] introduce a peer-to-peer agent framework to support autonomic applications in a decentralized distributed environment and provide those agents to discover, compose, control elements. The framework also defines agent interaction and negotiation protocols for enabling appropriate application behavior to be dynamically negotiated and enacted. The defined protocols and agent activities are supported by a scalable decentralized and shared-space based substrate.

The authors in [122] propose a multi-agent flexible and scalable autonomic service and network management architecture. The proposed architecture is expected to reduce the time for new services and minimize the cost of operations, development as well as deployment of services in a scalable, flexible and autonomic way. The cost model show that only instantiation cost of activated services is included, which means the cost might be reduced to its minimum.

The authors in [15] describe an agent-based semantic platform where autonomic

computing principles can be applied to ensure the constant update of platform knowledge base. Self-optimization and self-management techniques are proved to be very effective for population as well as update of a semantic annotation repository. In addition, low computational requirements and a built-in along with naturally distributed architecture allow an easy deployment of the proposed platform on current Web.

The authors in [171] introduce an extension to UML 2.0 called Agent Modeling Language (AML) that addresses specific needs, such as modeling autonomy, proactivity and role-based behavior. In addition, the AML can be directly used by the designers of autonomic computing systems to visually model their architectures and behaviors.

The authors in [99] present the architecture and operation of Rudder, a rule-based adaptive multi-agent infrastructure for supporting autonomic applications in a pervasive Grid environment. Rudder enables dynamic composition and coordination of autonomic components to manage changing application requirements as well as system context.

The authors in [45] propose a self-organized model of agent-enabling autonomic computing for the Grid environment. The model adopts intelligent agents as autonomic elements and enables those agent-based elements to dynamically organize the system management without a centralized control. At the element level, each agent possesses certain capabilities as well as interests according to its managed resources and governs internal affairs to achieve elementary autonomy. At the system level, agents contribute to system management and cooperate to implement advanced autonomic behavior. That cooperation is organized by dynamically associated relationship among those autonomic

elements (agents), including acquaintance, collaboration and notification.

The authors in [14] describe a toolkit for building multi-agent autonomic systems called Agent Building and Learning Environment (ABLE), which provides a lightweight Java agent framework, a comprehensive JavaBeans library of the intelligent software components, a set of development and test tools as well as an agent platform. The paper illustrates a series of agents built by ABLE components and presents three case studies using ABLE toolkit. By using the ABLE component library to build agents running on an ABLE distributed agent platform, the authors discuss how they can incrementally add new behaviors and capabilities to autonomic systems.

The authors in [127] adopt the multi-agent approach for developing Autonomic Information System (AIS), which can adjust its processing algorithms and data sources to provide necessary information at various levels of efficiency and effectiveness. The approach is based on Organization Model for Adaptive Computational Systems.

The authors in [95] present a model of adaptive agent built from the fine-grained reusable components which can implement non-functional mechanisms, such as mobility, adaptation skills and communication. Every agent can dynamically and autonomously change its components to adapt runtime context, which improves safety and performance for open, pervasive as well as large-scale distributed applications.

The authors in [59] indicates an autonomic computing infrastructure called MAACE that can provide dynamically programmable control and management services to support development along with deployment of intelligent applications. Moreover, the MAACE

can provide an environment to manage and control software systems through multi-agent system cooperation, which has agent federation systems, agent mediate service systems and agent monitoring systems.

The authors in [132] explore a multi-agent approach for developing an autonomic architecture for telehealth systems, which involves remotely monitoring health conditions of patients in post-surgery and patients with chronic diseases or life-threatening health problems who require continuous monitoring.

The authors in [79] present the use of autonomic concepts for reflex autonomy in the development of a multi-agent system. In addition, findings are discussed with reference to the use of JADE agent platform.

The authors in [107] propose a multi-agent autonomic as well as bio-inspired based framework with the self-managing capabilities to solve complex scheduling problems by cooperative negotiation.

The authors in [173] describe two prototype agent-based systems, Lights-out Ground Operations System as well as Agent Concept Testbed, and their autonomic properties that were developed at NASA Goddard Space Flight Center to demonstrate autonomous operations of future space flight missions.

2.6.5 Formal Methods

This subsection states the research activity 5) in Figure 3. The authors in [104] use *Z* to construct a formal specification that can provide clear and precise definitions for objects, agents as well as autonomous agents; the definitions allow a better understanding of the

functionalities from different systems.

The authors in [137] provide an abstract agent architecture which can serve as an idealization of an implemented system and as a means for investigating theoretical properties first, then the paper [138] describes an alternative formalization by starting with that implemented system and formalizing semantics by an agent language, which can be viewed as an abstraction of the implemented system and allows agent programs to be written and interpreted.

The authors in [49] present a formal approach to the MAS by prototyping and simulation oriented processes. The authors use a multi-formalism approach which is the composition of Object-Z and state charts; this formalism enables the specification for both reactive and transformational aspects of the MAS as well as their prototyping by simulation. In addition, the authors use an organizational model that considers roles, interactions and organizations from requirements to detailed design.

The authors in paper [29] introduce an agent-oriented modeling technique based on Unified Modeling Language (UML) notation; graph transformation is used both on the level of modeling to capture agent-specific aspects and the underlying formal semantics. The authors also state a concurrency theory of graph transformation systems following a double-pushout approach in terms of formalizing the relation among global requirement specifications by sequence diagrams, and the implementation-oriented design models in which graph transformation rules specify the local operations of agents.

The authors in [196] propose a model-based approach to design and implement

intelligent agents for the MAS. They use formal methods at the design phase of the agent development life cycle instead of specifying agent behavior. The authors choose agent oriented G-net model based on G-net formalism, which is a type of high-level Petri net, to serve as a high-level design for intelligent agents. According to that high-level design, they further derive the agent architecture and detailed design for agent implementation; a toolkit called Agent Development Kit is developed to support the rapid development of intelligent agents for the MAS.

The authors in [37] propose a temporal logic to represent dynamic agent behavior, which is more powerful than corresponding classic logic and is useful for the description of dynamic behavior in reactive systems. The authors consider a multi-agent system as a system consisting of concurrently executing objects.

The authors in [195] use Predicate/Transition (PrT) nets, a high-level formalism of Petri net, to model and verify multi-agent behaviors. According to the PrT model, certain properties like parallel execution of multi-plans and their guarantee for the achievement of goals, can be verified by analyzing the dependency relations between transitions.

The authors in [200] provide a formal Specification Language for Agent-Based Systems (SLABS) to specify agent behaviors, which enable software engineers to analyze agent-based systems before their implementation.

The authors in [106] describe a Constraint-Based Agent (CBA) design approach that includes two formal models: 1) Constraint Nets; 2) Timed $\forall$-automata. A constraint net can model agents and their environment symmetrically as dynamical systems; a timed

$\forall$ -automata can specify desired real-time dynamic behaviors of those situated agents.

The authors in [77] study the adaptation of multi-agent systems from system level and describe a formal framework for the multi-agent systems with adaptation capabilities. The framework uses the polyadic pi-calculus that is suitable for specifying the software with dynamic configurations.

The authors in [128] develop a formal multi-agent system based on a modal algebra to preserve essential characteristics of its autonomous software agents; it can also explore the formal properties and management of cooperation (non-hierarchical or flat structures) as well as the coordination (hierarchical structures) between those agents, which can be constructed by the operations of the model.

The authors in [32] introduce a formal-language model to explicitly formalize agent-environment interaction in a multi-agent systems framework called Conversational Grammar Systems (CGS). The CGS provides a model with a high degree of flexibility, since it can accept new concepts and modify rules, protocols as well as settings during computation. The formal model used in this paper is based on eco-grammar systems, which can be defined as an evolutionary multi-agent system where different components interact with a special component called environment. Therefore, there are two types of components that are agents and the environment in an eco-grammar system; both of them can be represented by a string of symbols that identifies current state of the components. Those strings can change based on the sets of evolution rules (*L systems*); the interactions between agents and the environment are executed by agents' actions on the environment

state through certain productions from the sets of action rules.

The authors in [34] present a formal analysis of social interactions in multi-agent systems. The fundamental building blocks are social agents which may be individuals or group of agents whose structures can be formally characterized in terms of roles and relationships among them. The work presented in this paper is formulated under the BDI paradigm. A logical language *L* includes three modal operators *B*, *D* and *I* to express beliefs, desires as well as intentions respectively.

The authors in [139] propose a formal approach that adopts a formal specification language *Temporal Z* to cover the individual agent aspects and collective aspects of a multi-agent application in terms of coordination protocols, organization structures and planning activities. This paper also presents a methodology according to the stepwise refinements that allow developing a design specification from an abstract requirement specification.

2.7 Summary

In this chapter, we have briefly reviewed some concepts of the autonomic computing technology that can be applied to the RASF.

An autonomic system has the characteristics of self-configuration, self-healing, self-optimization and self-protection; the autonomic computing control loop makes a foundation of autonomic systems. In order to implement those characteristics, autonomic managers should have the following capabilities: 1) policy determination; 2) solution knowledge; 3) common system administration; 4) problem determination; 5) autonomic

monitoring; 6) complex analysis; 7) transaction measurement.

We also described the architectures, open standards, development and related work of the autonomic computing by which we can conclude that agent-based approach is a natural way to model autonomic systems, so many ideas from the MAS community can be adapted to implement autonomic systems. Thus, we gave an introduction of agent-based computing technology, which included the definitions, interactions, communication language, architecture, programming and formal methods of multi-agent systems.

Finally, we discussed the related work on autonomic systems modeling, real-time reactive systems, multi-agent systems and potential formal methods used for specifying reactive autonomic systems. We concluded that most of current formal approaches do not have appropriate mechanisms to specify RAS and have not addressed well on verifying emergent behavior (see Page 3 and Figure 1). Thus, we will give an introduction of category theory in the next chapter as a formal framework to specify autonomic and reactive behavior of the RAS modeled by RASF.

Chapter 3: Background: Category Theory

This chapter states the research activity 6) in Figure 3, which is the literature review on Category Theory. Structure is crucial in large specifications and programs. A well-chosen structure may greatly improve understanding, validation as well as modification of a specification. In the RAS where self-management behavior is one of the most important characteristics, the management of evolving specifications and analysis of changes require a specification structure, which can isolate those changes in a small number of components and analyze the impacts of a change on interconnected components [185].

Category theory has been proposed as a framework to offer that structure; it also has been successfully used to provide composition primitives in both algebraic [189] and temporal logic [35] specification languages. Category theory has a rich body of theory to reason about objects and their relations (specifications as well as their interactions); it is abstract enough for a wide range of different specification languages. Furthermore, category theory for software specification has adopted a *correct by construction* approach by which components are specified, proved and composed in the way of preserving their properties [185].

Complex systems may be identified with diagrams (semi-formal), in which system components along with connectors and their interconnections represent nodes as well as edges respectively. However, the word *diagram* in category theory has a formal meaning and carries all the intuitions that come from practice. Comparing to other formalization of the software architecture concept, category theory is semantic framework to formalize

interconnection, configuration, instantiation and composition which are important aspects of modeling the RAS with both autonomous and autonomic behavior. This can be achieved at a very abstract level, since category theory proposes a toolbox applied to whatever formalism for capturing components' behavior, as long as that formalism satisfies certain structure properties [36].

Category theory focuses on the relationships (morphisms) between objects instead of their representations; the morphisms can determine the nature of interactions established between the objects. Thus, a particular category may reflect a corresponding architectural style. In addition, category theory provides techniques to manipulate and reason about diagrams for building hierarchies of the system complexity, allowing systems to be used as components of more complex systems, and inferring the properties of systems from their configurations [36].

3.1 Definition of Category [6]

Definition 3.1.1: A *category* consists of the following data:

- *Objects*: A, B, C, etc.
- *Arrows* (*Morphisms*): f, g, h, etc.
- For each arrow f, there are given objects: dom(f), cod(f) called *domain* as well as *codomain* of f, and $f : A \to B$ indicates that A = dom(f), B = cod(f).
- Given arrows $f : A \to B$ and $g : B \to C$ with cod(f) = dom(g), there is an given arrow: $g \circ f : A \to C$ called *composite* of f and g.

- For each object A, there is an given arrow: $1_A : A \to A$ called *identity arrow* of A.

These data need to satisfy the following laws:

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

Definition 3.1.2: A *functor* F: **C** → **D** between categories **C** and **D** is a mapping of objects to objects and arrows to arrows in the way of: 1) $F(f: A \to B) = F(f) : F(A) \to F(B)$; 2) $F(g \circ f) = F(g) \circ F(f)$; 3) $F(1_A) = 1_{F()A}$.

Definition 3.1.3: in any category **C**, an arrow $f : A \to B$ is called an *isomorphism* if there is an arrow $g : B \to A$ in **C** such that $g \circ f = 1_A$ and $f \circ g = 1_B$. Since identities are unique, $g = f^{-1}$. A is *isomorphic* to B: $A \cong B$ if there exists an isomorphism between them.

Definition 3.1.4: in any category **C**, an object is called *initial object* I if for any object X in **C**, there is a unique morphism $I \to X$; an object is called *terminal object* T if for any object X in **C**, there is a unique morphism $X \to T$.

Definition 3.1.5: *discrete category* is a category where all morphisms are identity morphisms.

Definition 3.1.6: *category of sets* is the category in which objects are sets. The morphism between sets A and B are all functions from A to B.

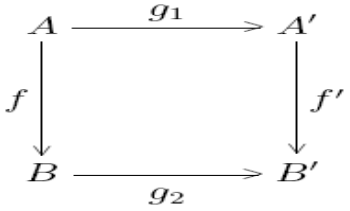
3.2 Constructions on Category [6]

Definition 3.2.1: The *product* of two categories **C** and **D**: $\mathbf{C} \times \mathbf{D}$ has objects of the form (C, D) for $C \in \mathbf{C}$, $D \in \mathbf{D}$ and arrows of the form $(f, g) : (C, D) \to (C', D')$ for $f : C$

$\to C' \in \mathbf{C}$ and $g : D \to D' \in \mathbf{D}$. Composition and units are defined as: $(f', g') \circ (f, g) = (f' \circ f, g' \circ g)$, $1_{(C,D)} = (1_C, 1_D)$; there are two *projection functors*: $\mathbf{C} \xleftarrow{\pi_1} \mathbf{C} \times \mathbf{D} \xrightarrow{\pi_2} \mathbf{D}$ defined by $\pi_1(C, D) = C$, $\pi_1(f, g) = f$ and similarly for π_2.

Definition 3.2.2: the *opposite* or *dual* category $\mathbf{C}^{\mathrm{op}}$ of a category **C** has the same objects as **C**, and an arrow $f : C \to D$ in $\mathbf{C}^{\mathrm{op}}$ is an arrow $f : D \to C$ in **C**. Thus, $\mathbf{C}^{\mathrm{op}}$ is just **C** with all of the arrows being formally inversed. It is convenient to have a notation for distinguishing objects and arrows in **C** ($f : C \to D$) from the same ones in $\mathbf{C}^{\mathrm{op}}$ ($\overline{f} : \overline{D} \to \overline{C}$). With this notation, the composition and units in $\mathbf{C}^{\mathrm{op}}$ can be defined in terms of corresponding operations in **C** as $1_{\overline{C}} = \overline{1_C}$ and $\overline{f} \circ \overline{g} = \overline{g \circ f}$.

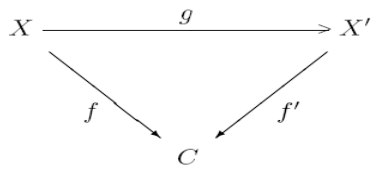
Definition 3.2.3: the *arrow category* $\mathbf{C}^{\to}$ of a category **C** has the arrow of **C** as objects, and an arrow g from $f : A \to B$ to $f' : A' \to B'$ in $\mathbf{C}^{\to}$ is a *commutative square* as the following diagram, where g_1 and g_2 are arrows in **C**. Such an arrow is a pair of arrows $g = (g_1, g_2)$ in **C** that $g_2 \circ f = f' \circ g_1$, and the identity of arrow 1_f on an object $f : A \to B$ is the pair $(1_A, 1_B)$. The composition of arrows is $(h_1, h_2) \circ (g_1, g_2) = (h_1 \circ g_1, h_2 \circ g_2)$, and there are two functors: $\mathbf{C} \xleftarrow{\mathrm{dom}} \mathbf{C}^{\to} \xrightarrow{\mathrm{cod}} \mathbf{D}$.



Definition 3.2.4: the *slice category* **C**/C of a category **C** over an object $C \in \mathbf{C}$ has:

- objects: all arrows $f \in \mathbf{C}$ such that cod(f) = C.

- arrows: g from $f : X \to C$ to $f' : X' \to C$ is an arrow $g : X \to X'$ in **C** such that $f' \circ g = f$ as the following diagram shows:



Definition 3.2.5: the *co-slice* category C/**C** of a category **C** under an object $C \in \mathbf{C}$ has:

- objects: all arrows $f \in \mathbf{C}$ such that dom(f) = C.
- arrows: h from $f : C \to X$ to $f' : C \to X'$ is an arrow $h : X \to X'$ such that $h \circ f = f'$

3.3 Abstract Structures in Category [6]

Definition 3.3.1: in any category **C**, an arrow $f : A \to B$ is called a:

- *monomorphism* if given any $g, h : C \to A$, $f \circ g = f \circ h$ implies $g = h$; it can be represented as $f : A \rightarrowtail B$

$$C \underset{h}{\overset{g}{\rightrightarrows}} A \xrightarrow{f} B$$

- *epimorphism* if given any $i, j : B \to D$, $i \circ f = j \circ f$ implies $i = j$; it can be represented as $f : A \twoheadrightarrow B$

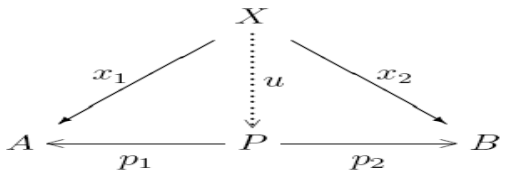
$$A \xrightarrow{f} B \underset{j}{\overset{i}{\rightrightarrows}} D$$

Definition 3.3.2: in any category **C**, an object: 1) 0 is *initial* if for any object C, there is a unique morphism $0 \to C$; 2) 1 is *terminal* if for any object C, there is a unique morphism $C \to 1$. A terminal object in **C** is exactly an initial object in $\mathbf{C}^{op}$.

Definition 3.3.3: a *split* monomorphism (epimorphism) is an arrow with a left (right)

inverse. Given arrows $e : X \to A$ and $s : A \to X$ such that $e \circ s = 1_A$, then s is called a *section* or *splitting* of e; e is called a *retraction* of s; A is called a *retract* of X.

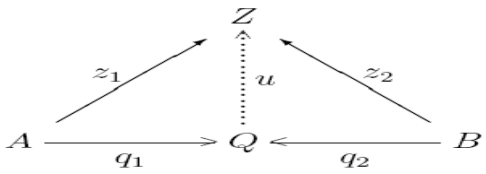
Definition 3.3.4: in any category **C**, a *product diagram* for the object A and B consists of an object P and arrows $A \xleftarrow{p_1} P \xrightarrow{p_2} B$ satisfying: given any diagram of the form $A \xleftarrow{x_1} X \xrightarrow{x_2} B$, there exists a unique $u : X \to P$ making the following diagram commute, and a pair of objects may have many different products in a category.



3.4 Duality in Category [6]

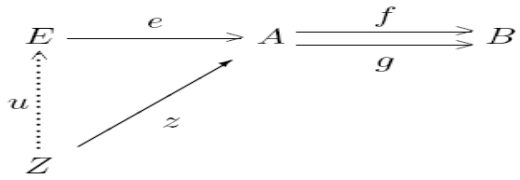
Proposition 3.4.1: for any statement $\sum$ about categories, if $\sum$ holds for all categories, then so does the dual statement $\sum^*$: $\sum$ implies $\sum^*$.

Definition 3.4.2: a diagram $A \xrightarrow{q_1} Q \xleftarrow{q_2} B$ is a *coproduct* of A, B if for any Z and $A \xrightarrow{z_1} Z \xleftarrow{z_2} B$, there is a unique $u : Q \to Z$ with $u \circ q_i = z_i$ as indicated in:

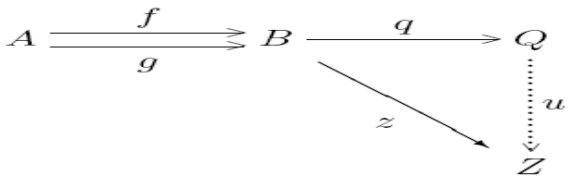


The coproduct is usually represented as $A \xrightarrow{i_1} A + B \xleftarrow{i_2} B$, and [$f$, g] represents uniquely determined arrow $u : A + B \to Z$. The *coprojection* $i_1 : A \to A + B$ and $i_2 : B \to A + B$ are usually called *injections*, and a coproduct of two objects is exactly their product in the opposite category.

Definition 3.4.3: in any category **C**, given parallel arrows $A \overset{f}{\underset{g}{\rightrightarrows}} B$, an *equalizer* of f and g consists of E and $e : E \to A$ such that: $f \circ e = g \circ e$, which means given $z : Z \to A$ with $f \circ z = g \circ z$ there is a unique $u : Z \to E$ with $e \circ u = z$ as indicated in the following diagram:



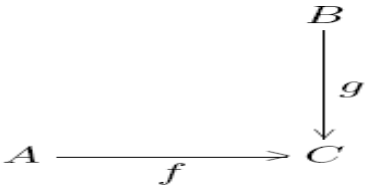
Definition 3.4.4: for any parallel arrows $f, g : A \to B$ in a category **C**, a *coequalizer* consists of Q and $q : B \to Q$ with the property $q \circ f = q \circ g$ as indicated in the following diagram:



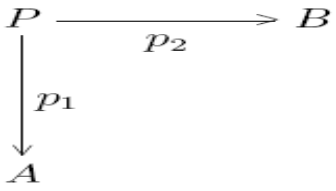
That is, given any Z and $z : B \to Z$, if $z \circ f = z \circ g$, then exists a unique $u : Q \to Z$ such that $u \circ q = z$.

3.5 Limits and Colimits [6]

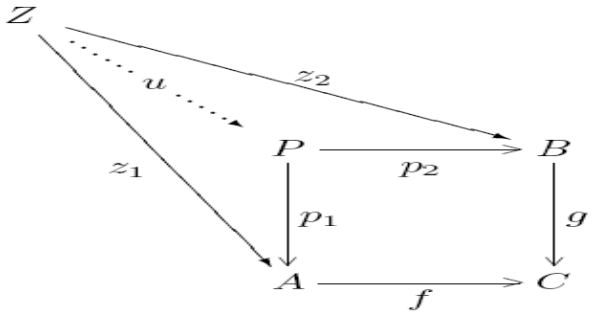
Definition 3.5.1: in any category **C**, a *pullback* of arrows f, g with cod(f) = cod (g):



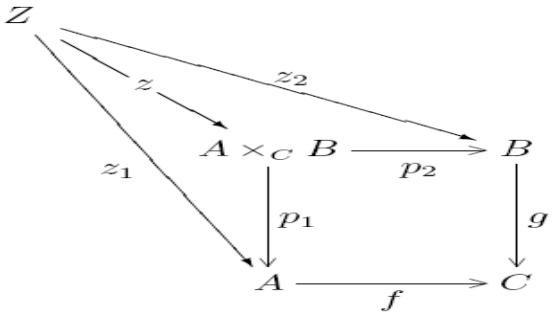
consists of the following arrows such that $f \circ p_1 = g \circ p_2$.



For example, given any $z_1 : Z \to A$ and $z_2 : Z \to B$ with $f \circ z_1 = g \circ z_2$, there exists a unique $u : Z \to P$ with $z_1 = p_1 \circ u$ and $z_2 = p_2 \circ u$.

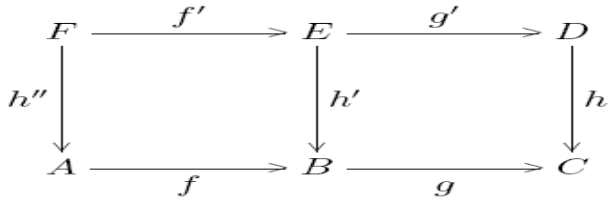


Pullbacks may use product-style notation as the following diagram:



Lemma 3.5.2: consider the commutative diagram in a category with pullbacks:

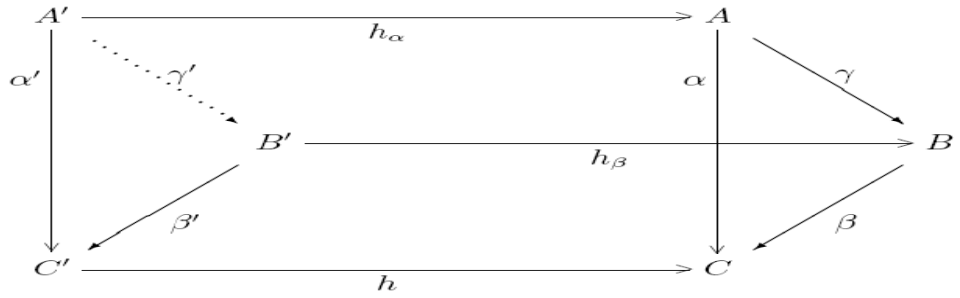
- If the two squares are pullbacks, so is the outer rectangle: $A \times_B (B \times_C D) \cong A \times_C D$
- If the right square and the outer rectangle are pullbacks, so is the left square



Corollary 3.5.3: the pullback of a commutative triangle is a commutative triangle.

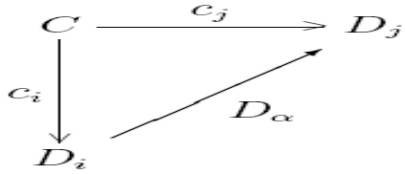
Specially, given a commutative triangle as on the right end of the *prism diagram* below,

for any $h : C' \to C$ if one can form the pullbacks α' and β' as on the left end, then there exists a unique γ' making the left end a commutative triangle as well as the upper a commutative rectangle.

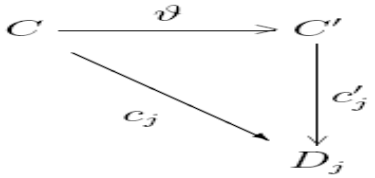


Definition 3.5.4: Dualize the definition f a pullback o define the "copullback" (usually called the "pushout") of two arrows with common domain.

Definition 3.5.5: let **J** and **C** be categories. A *diagram* of *type* **J** in **C** is a functor D : **J** → **C**. The objects in the *index category* **J** are represented as $i, j, \ldots$ and the values of the functor D : **J** → **C** are in the form D_i, D_j, …. A *cone* to a diagram D consists of an object C and a family of arrows in **C**, $c_j : \mathrm{C} \to D_j$ for each object $j \in \mathbf{J}$ such that for each arrow $\alpha : i \to j$ in **J**, the following triangle commutes.



A *morphism* of cones $\upsilon : (C, c_j) \to (C', c_j')$ is an arrow υ in **C** making each triangle commute.



Definition 3.5.6: a *limit* for a diagram $D : \mathbf{J} \to \mathbf{C}$ is a terminal object in the category **Cone**(D) represented as $p_i : \varprojlim_{j} D_j \to D_i$. A *finite limit* is a limit for a diagram on finite index category **J**. Given any cone (C, c_j) to D, there is a unique arrow $u : C \to \varprojlim_{j} D_j$ such that for all j, $p_j \circ u = c_j$.

Definition 3.5.7: a functor $F : \mathbf{C} \to \mathbf{D}$ is said to *preserve limits of type* **J** if whenever $p_j : L \to D_j$ is a limit for a diagram $D : \mathbf{J} \to \mathbf{C}$, the cone $F p_j : FL \to F D_j$ is then a limit for diagram $FD : \mathbf{J} \to \mathbf{D}$, $F(\varprojlim_{\leftarrow} D_j) \cong \varprojlim_{\leftarrow} F(D_j)$. A functor that preserves all limits is said to be continuous.

Definition 3.5.8: a functor of the form $F : \mathbf{C}^{\mathrm{op}} \to \mathbf{D}$ is called a *contravariant functor* on **C**, which takes $f : A \to B$ to $F(\mathrm{f}) : F(B) \to F(A)$ and $F(g \circ f) = F(f) \circ F(g)$.

Definition 3.5.9: a *colimit* for a diagram $D : \mathbf{J} \to \mathbf{C}$ is an initial object in the category of *cocones* from the *base* D, which consists of an object C (the vertex) and arrow $c_j : D_j \to C$ for each $j \in \mathbf{J}$, such that for all $\alpha : i \to j$ in **J**, $c_j \circ D(\alpha) = c_i$. A morphism of cocones $f : (C, (c_j)) \to (C', (c_j'))$ is an arrow $f : C \to C'$ in **C** such that $f \circ c_j = c_j'$ for all $j \in \mathbf{J}$; an initial cocone maps uniquely to any other cocone from D, and a colimit can be represented as $\varinjlim_{j \in \mathbf{J}} D_j$.

Definition 3.5.10: a functor $F : \mathbf{C} \to \mathbf{D}$ is said to *create limits of type* **J** if for every diagram $C : \mathbf{J} \to \mathbf{C}$ and limit $p_j : L \to F C_j$ in **D**, there is a unique cone $\overline{p_j} : \overline{L} \to C_j$ in **C** with $F(\overline{p_j}) = p_j$ and $F(\overline{L}) = L$ which is a limit for C; every limit in **D** is the image of a unique cone in **C**. The notation of *creating colimits* is defined analogously.

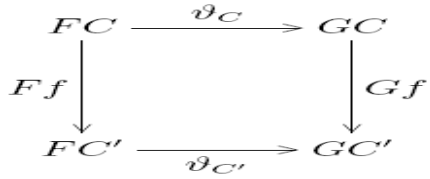
3.6 Functors and Naturality [6]

Definition 3.6.1: a functor $F : \mathbf{C} \to \mathbf{D}$ is said to be:

- *injective on objects* if the object part $F_0 : \mathbf{C}_0 \to \mathbf{D}_0$ is injective, and it is *surjective on objects* if F_0 is surjective.
- *injective on arrows* if the arrow part $F_1 : \mathbf{C}_1 \to \mathbf{D}_1$ is injective, and it is *surjective on arrows* if F_1 is surjective.

Proposition 3.6.2: a *full subcategory* $\mathrm{U} \rightarrowtail \mathrm{C}$ consists of some objects in **C** and all the arrows between them, thus satisfying the closure conditions for a subcategory.

Definition 3.6.3: for categories **C**, **D** as well as functors F, $G : \mathbf{C} \to \mathbf{D}$, a *natural transformation* ($\upsilon : F \to G$) is a family of arrows in **D**, $(\upsilon_C \quad FC \to GC)_{C \in \mathbf{C}_0}$, such that for any $f : C \to C'$ in **C**, there exists $\upsilon_{C'} \circ F(f) = G(f) \circ \upsilon_C$ as indicated in the following diagram. Given such a natural transformation $\upsilon : F \to G$, the **D**-arrow $\upsilon_C : FC \to GC$ is called the *component* of υ at C.



Definition 3.6.4: the *functor category* Fun(**C**, **D**) has: 1) objects: functors $F : \mathbf{C} \to \mathbf{D}$; 2) arrows: natural transformations $\upsilon : F \to G$. For each object F, $(1_F)_C = 1_{FC} : FC \to FC$ and the components of $F \xrightarrow{\upsilon} G \xrightarrow{\phi} H$ has components $(\phi \circ \upsilon)_C = \phi_C \circ \upsilon_C$.

Definition 3.6.5: a *natural isomorphism* is a natural transformation $\upsilon : F \to G$ which is an isomorphism in the functor category Fun(**C**, **D**).

Lemma 3.6.6: a natural transformation $\upsilon : F \rightarrow G$ is a natural isomorphism if each component $\upsilon_C : FC \rightarrow GC$ is an isomorphism.

Definition 3.6.7: an *equivalence of categories* consists of functors $E : \mathbf{C} \rightarrow \mathbf{D}$, $F : \mathbf{D} \rightarrow \mathbf{C}$ and natural isomorphisms $\alpha : 1_{\mathbf{C}} \simeq F \circ E$ in $\mathbf{C}^{\mathbf{C}}$, $\beta : 1_{\mathbf{D}} \simeq E \circ F$ in $\mathbf{D}^{\mathbf{D}}$; the functor F is called a *pseudo-inverse* of E; the categories $\mathbf{C}$ and $\mathbf{D}$ are said to be *equivalent*: $\mathbf{C} \simeq \mathbf{D}$. The equivalence of categories is a generation of isomorphism, and two categories $\mathbf{C}$, $\mathbf{D}$ are isomorphic if there are functors $E : \mathbf{C} \rightarrow \mathbf{D}$, $F : \mathbf{D} \rightarrow \mathbf{C}$ such that $1_{\mathbf{C}} = GF$, $1_{\mathbf{D}} = FG$. In the case of equivalence $\mathbf{C} \simeq \mathbf{D}$, the identity natural transformations are replaced by natural isomorphisms; the equivalence of categories may be considered as *isomorphism up to isomorphism*.

Property 3.6.8: If $\boldsymbol{C}$ is a full subcategory of $\boldsymbol{D}$ and every $Y \in \boldsymbol{D}$ is isomorphic to some object X in $\boldsymbol{C}$, then the inclusion functor F: $\boldsymbol{C} \rightarrow \boldsymbol{D}$ is an equivalence of categories.

3.7 Related Work

There has been an increasing interest on applying category theory to various areas of computer science. Particularly, it has been used to: 1) study different approaches for the mathematical semantics of programming languages; 2) define semantics for parallelism and synchronization; 3) provide a generalized concept of automata; 4) specify problems, clarify concepts, formulate consistent definitions, analyze and help in understanding computational phenomenon [94].

Category theory has played a role in studying initial algebra [41] and many sorted algebraic theories [42] have formed a basis for current algebraic semantics of abstract

data types. Category theory has also been applied to relate different theories. For example, the author in [186, 187, 188] describes an application of category theory on the Petri-Net Model for parallel computation and relates it to models, such as trees, state machines, event structures and nets.

The authors in [163, 164] apply a category theory framework to general systems. A symbolic category method is introduced to categorize various system classes and their concepts. The authors also indicate how the concept of states as well as their space representation can be derived in the framework of category theory.

The authors in [43] apply the category theory as a conceptual tool to model general systems through the abstract representation of systems, which take objects, systems, interconnection and behavior as a basis. The authors present a Behavioral Theorem, stating that the behavior of an interconnection between objects can be considered as the behavior of individual objects; they also indicate that the notion of autonomy, interaction, cooperation and self-organization are relevant to their study.

The author in [46] presents a unifying framework based on category theory for the component dependencies modeling techniques. The authors in [91] provide a universal categorical model of synchronization between computing processes. The authors in [146] define the synchronization on a formula of two consequence systems and provide the categorical characterization for construction.

The authors in [71] present the modular composition of a transaction processing protocol, namely three-phase commit (3PC) protocol utilizing some concepts of category

theory; they illustrate how the overall global properties of the protocol can be proved by utilizing constructs for local sub-properties from the inherent building blocks in the 3PC protocol.

The authors in [5] state a general definition of machines in an arbitrary category, which unifies the theories of sequential machines, linear control systems, tree automata and stochastic automata.

The authors in [60] provide a precise semantics for both components structuring and models mapping by using category theory. In this paper, morphism composition is used to trace the interconnections and mapping relations among component-based models, while consistency between the sorts/operations of those models at different abstract levels is maintained by functors.

The authors in [168] abstract and describe the process of multi-sensor data fusion as well as its taxonomy by a language of category theory. Categories are developed for sensors, data sets, processors, feature sets, classifiers as well as label sets. Fusion rules are defined and shown to hold a unique role in various categories; fusion processes can be described as an optimization of fusion rules in an appropriate category.

The author in [50] provides the architecture for system configuration, which is independent of various approaches for the specification, design and coding of systems. The key idea is to focus on configuring those systems from reusable modules at any stage during system development. The module is precisely defined as an instance of a textual specification; the configuration takes place in a mathematical framework that is based on

the category theory.

The authors in [182] report on the application of category theory to design a simulated robot control system, where a neural network controller is constructed based on a desired conceptual ontology.

3.8 Summary

In this chapter, we have presented some definitions, propositions and theorems of the category theory, which may be applied to specify autonomic and reactive behavior of the RAS modeled by RASF in Chapter 6.

Category theory has a rich body of theory to reason about objects as well as their relations, and it is abstract enough for a wide range of different specification languages. Category theory for the software specification has adopted a *correct by construction* approach by which components are specified, proved and composed in the way of preserving their properties. Moreover, category theory can provide techniques to manipulate and reason diagrams for building hierarchies of system complexity, allowing systems to be used as components of more complex systems and inferring properties of the systems from their configurations.

Finally, we have introduced the concept of constructions, duality, limits, naturality and adjoints in the category theory. We will start to describe our case studies in terms of illustrating the RASF approach in the next chapter.

Chapter 4: Background: Case Studies

This chapter states the research activity 4) in Figure 3, which is the literature review on case studies. I had one publication [126] during this stage.

4.1 Mars-World

The Mars-world case study [201] is mainly used in the rest of this thesis to illustrate our approach. In this case study, a group of robots accomplish ore exploitation on the planet Mars. To achieve this goal, these robots must locate ore resources (the squares in Figure 15) in the area, mine them, and transport produced the ore to a base (the pentagon in Figure 15) in terms of storage. This process is completed by three types of robots, and all the robots have a sensor range to detect presence of ore. There is a *sentry robot* whose responsibility is to analyze suspicious spots to evaluate if there is enough ore to be mined. This type of robot has a wider sensor range to better verify candidate locations. When the sentry robot evaluates a mine to be exploited, it sends its location to a second robot type known as *production robot*. This robot has devices to dig and mine ore. After finishing its job, the production robot calls a *carry robot* to transport the produced ore to the home base. The carry robot has necessary equipments to carry ore and ability to move faster than the other types of robots.

To better illustrate our approach, we have added two more types of robots to this case study. These two robots are more involved in administration and coordination tasks at the autonomic group as well as system levels. A *group supervisor robot* can form a group of robots to find and exploit ore in the areas requested by a *system manager robot*. It can

register specialist robots, supervise them and resolve conflicts between them. A *system manager robot* may receive requests from the ground station on Earth and transfer required data back. It can manage *group supervisor robots*, assign tasks to them and collect requested data from them. Moreover, each *supervisor robot* and *manager robot* has its backup robots used to ensure fault-tolerance, since they serve as critical roles and store important data, such as repositories and work outcomes.

Figure 15 depicts a sample scenario of the Mars-world. When the *group supervisor robot* receives an order with the subarea coordinate of the ole, it forms an exploration group with *sentry robots*, *production robots* and *carry robots*. The size and composition of the group is dynamic based on how much ore is found and left in its subarea. For instance, the *supervisor robot* in subarea1 can ask the *supervisor robot* in subarea2 or any other *supervisor robot* for more *production robots* and *carry robots* because of new detected ore. The amount of remaining ore of each spot is indicated as a percentage number and reported to the *supervisor robot* by the *sentry robots*.

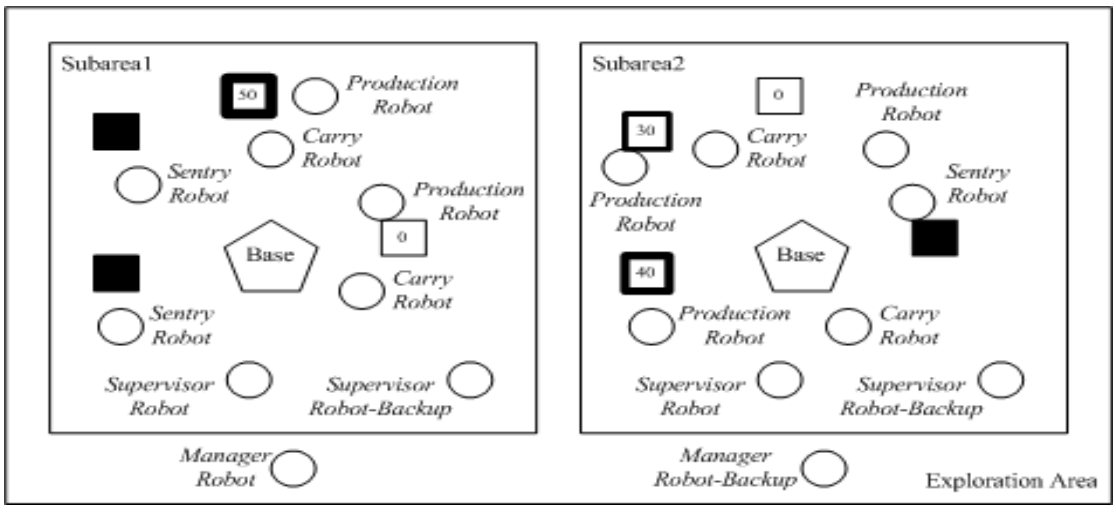


Figure 15: A sample scenario of Mars-world

4.2 Prospecting Asteroid Mission

In order to support the versatility and flexibility of applying our research outcome, we also select the Prospecting Asteroid Mission (PAM) case study as another application modeled by the RASF.

The PAM is an application of Autonomous Nano Technology Swarm (ANTS) mission architecture from NASA. The PAM consists of 1000 pico-spacecraft organized into 10 specialist classes with highly maneuverable as well as configurable solar sails. The basic design elements are low-power, low-weight components and individual systems that are capable of operating as fully autonomous and adaptable units for swarm demands as well as environmental needs. Through 10 to 20 sub-swarms operating simultaneously, hundreds of asteroids could be explored during a mission traverse for an asteroid belt [25].

The PAM must fulfill the following asteroid survey requirements: 1) optimal science operations at every asteroid such as the search of appropriate spacecraft trajectories that can enable efficient operation of workers' instruments and concurrent operations between multiple asteroids such as asteroid detection and tracking; 2) ongoing evolution of strategies as a function of asteroid characteristics; 3) no single point failure and robustness with respect to minor or critical loss; 4) a high level of autonomy as a group of specialized workers. The PAM is designed for a systematic study of an entire population of elements and involves not only a smart spacecraft, but also an autonomic and distributed network of sensors or spacecraft with the specialized device capabilities, for

instance, computing, imaging and spectrometry, as well as adaptable and evolvable heuristic systems. Furthermore, the sub-swarms of spacecraft can operate autonomously to enable the optimal gathering of complementary measurements for selected targets and can also simultaneously operate in a broadly defined framework of goals to select targets from candidate asteroids [25].

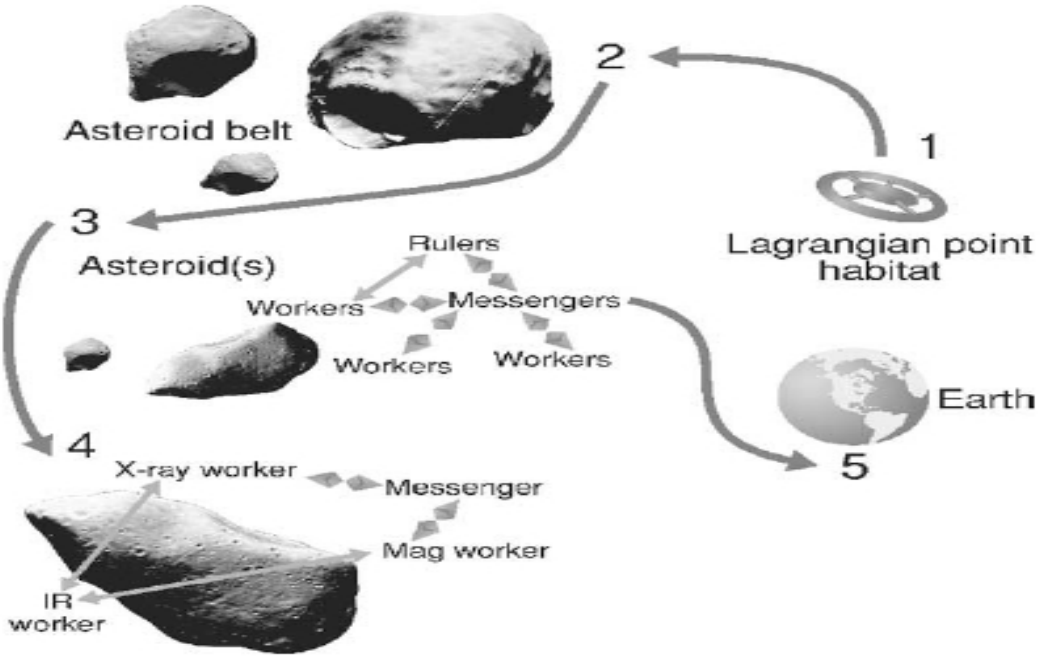


Figure 16: A Sample PAM Scenario [174]

The PAM spacecraft explore a selected asteroid through offering the highest quality and coverage of measurement by particular classes of measurers that are called virtual teams. A virtual *instrument* team consists of members from each spacecraft type to optimize data collection. Another strategy involves providing the comprehensive measurement to solve particular scientific problems by forming virtual *experiment* teams made up of members of multiple specialist spacecraft. The social structure of the PAM swarm can be determined by a particular set of scientific and mission requirements, as

well as representative system elements may include [27]: 1) a *general*, for distributed intelligence operations, resource management, mission conflict resolution, navigation, mission objectives and collision avoidance; 2) *rulers*, for heuristic operation planning, local conflict resolution, local resource management, scientific discovery data sharing and task assignment; 3) *workers*, for local heuristic operation planning and scientific data collection [27].

The PAM can therefore be regarded as an RAS with autonomic properties [174]. The resources can be configured and reconfigured to support parallel operations at hundreds of asteroids over a given period (self-configuration). For example, a sub-swarm may be organized for scientific operations at an asteroid, and this sub-swarm can be reorganized at another asteroid. The rulers may maintain data on different types of asteroids and determine their characteristics over time. Therefore, the whole system can be optimized because time will not be wasted on the asteroids that are not of interest or are difficult to observe (self-optimization). The messengers provide communication between the rulers, workers and Earth, so they can adjust their positions to balance the communication (self-adaptation). The PAM individuals should be capable of coordinating their orbits and trajectories to avoid collisions with other individuals in a reactive way. Moreover, the plans of the rulers should incorporate the constraints necessary for acceptable collision risk between the spacecraft when they perform observation tasks (self-protection and reactive). The rulers capable of sensing solar storms should invoke the goal of protecting their missions when they recognize a threat of such storms. In addition, the rulers can

inform the workers of the potential for these events to occur, so that they can orient their solar panels and sails to minimize the impact of solar wind. The rulers can also power down the workers' subsystems to minimize the disruption from charged particles (self-protection and self-adaptation).

4.3 End-to-End iFix Tool

In order to support the feasibility of applying our research outcome on industrial projects, I select the End-to-End iFix Tool (E2E) case study as an industrial application modeled by the RASF during my research internship at IBM Canada.

The E2E is a fully automated fix (patch) generation tool for the IBM WebSphere Application Server (WAS). It is a web-based application to process official fix creation requests from the IBM support teams. The tool implements an automated and autonomic process to build and test iFixes with minimal user input as well as intervention based on source code for a fix being available and identifiable in a source code repository system. The tool consists of three main components: 1) Web-based GUI front-end; 2) Fix creation engine; 3) Fix validation engine. The information obtained from the front-end is used to obtain an installable iFix from the creation engine; the validation engine is then used to install, test and uninstall the iFix to validate correctness as well as completeness. The E2E interacts with a repository tool to store source code and a build tool to compile the source code into object code and a packaging tool to package the object code into an installable fix.

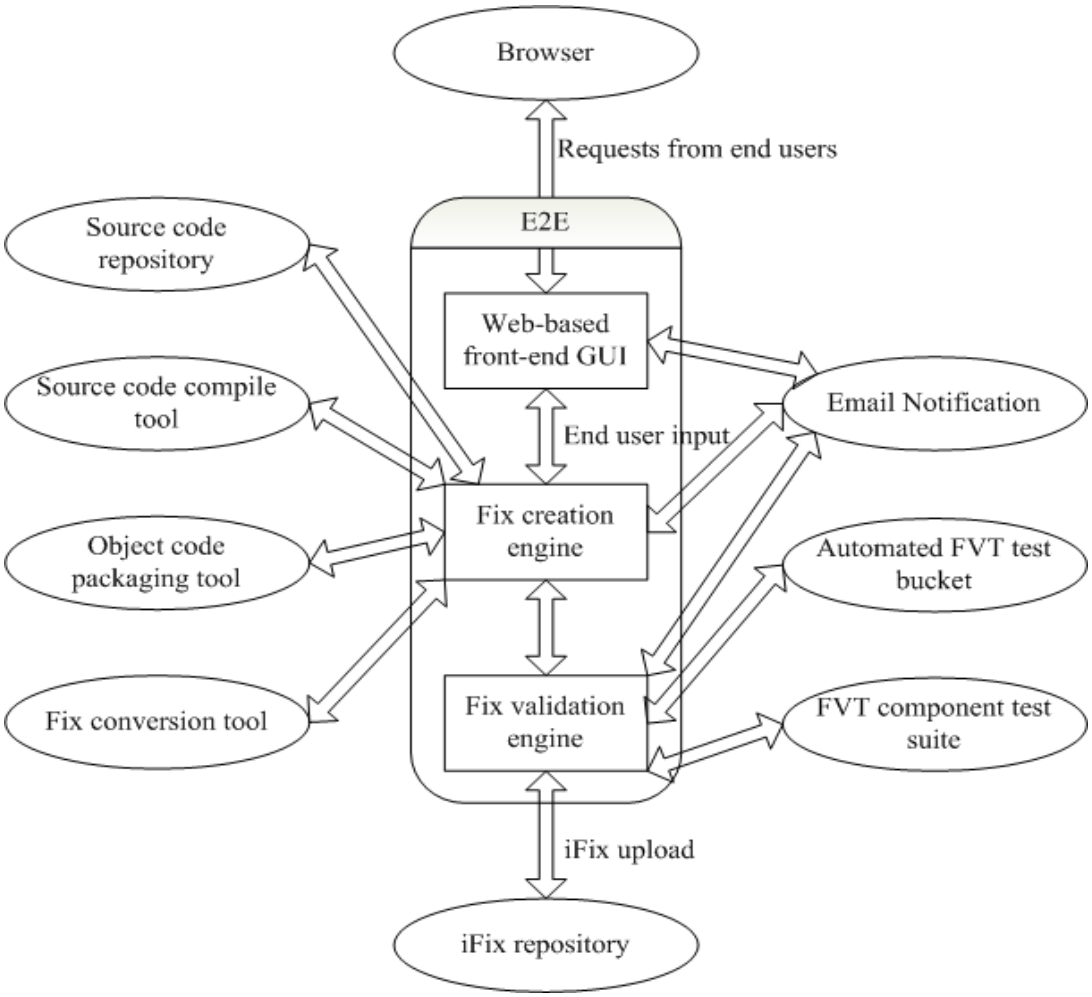


Figure 17: A Perspective of E2E

The fix creation engine leverage source code repository, source code compile tool and object code packaging tool to create an ifix. The actions below are executed based on the user input: 1) defect information is extracted from the source code repository; that information is then displayed to the end users for confirmation and modifications, which can be checked for consistency with the source code repository; 2) the engine then wraps up the information and forwards a build request to the source code compile tool in terms of generating binary classes, which can be packaged by the object code packaging tool, validate by the FVT test bucket and uploaded to the iFix repository.

The fix validation engine provides a mechanism to verify the integrity (correctness, completeness, installability and uninstallability) of an iFix reactively in the run time, which leverages two services: 1) GIT automation framework to validate iFix installation; 2) FAT bucket hosted by the Continuous Testing Framework server. If both testing results are positive, the iFix will then be uploaded to public iFix repository and an email notification will be sent the end users with the deliverable iFix.

4.4 Related Work

The authors in [53] present a model-driven autonomic computing technology for the ANTS missions. Comparing to other models, the new hierarchical model can overcome challenges of largeness, complexity, dynamicity and unexpectedness in the ANTS system. The paper also describes the structure and functions of virtual neuron, which is a basic unit together with the model for the model-driven autonomic technology in the ANTS missions.

The authors in [18] introduce an Agent Modeling Language (AML) and demonstrate how AML can be applied to efficiently, accurately and comprehensively model the PAM system. A selection of the AML models that specify the PAM domain, goals, architecture as well as behaviors are also presented in this paper.

The authors in [175] describe a formal task-scheduling approach and model a self-scheduling behavior of the ANTS by an autonomic system specification language, where both group and individual tasks are structured in the form of time aware fault-tolerant which applies tolerance to timing violations.

4.5 Summary

In this chapter, we have described three case studies in terms of illustrating the RASF approach, which includes Mars-world, PAM and End-to-End iFix Tool.

In Mars-world, the objective for a group of robots is to mine ore; the mining process is composed of locating the ore, mining it, and transporting the mined ore to a home base. The sensed occurrences of ore are reported to a *sentry robot* that has a wider sensor range in terms of verifying whether a suspicious spot actually has ore. When ore is found, the location is sent to a randomly selected *production robot* with the mining device. After mining is finished, a group of *carry robots* is requested to transport ore to the home base.

The PAM spacecraft study a selected target by particular classes of measurers called virtual teams. For example, an *experiment* team consists of the specialist classes to solve particular scientific problems, such as Petrologist team. The system elements include *generals*, *rulers*, *workers*, and *messengers*.

The E2E is a web-based application to process official fix creation requests from the IBM support teams. The tool implements an automated and autonomic process to build and test iFixes with minimal user input as well as intervention based on source code for a fix being available and identifiable in a source code repository system.

After introducing all necessary background on autonomic computing, multi-agent systems, real-time reactive systems, formal methods, category theory and case studies, we will illustrate the RASF methodology in the next chapter.

Chapter 5: Methodology of RASF

This chapter states the research activity 7) in Figure 3, which is the prototype design of the RASF model, and describes the contribution of the author (Section 5.1 & 5.2) as well as the collaboration and supervision to the master students involved in the RASF project (Section 5.3 & 5.4).

Real-time reactive systems are some of the most complex systems, and they have become increasingly heterogeneous and intelligent. However, current formal approaches do not have an appropriate mechanism to specify autonomic reactive systems, which are able to simplify and enhance the experience of end-users by anticipating their needs in a complex, dynamic and uncertain environment. With autonomic behavior, the real-time reactive systems can be more self-managed to themselves and more adaptive to their environment. Therefore, our goal is to build a formal framework, the RASF, which can leverage modeling, specification and development of the RAS.

5.1 RAS Model in RASF

The RAS architecture model (see Figure 18) is a four-layer architecture that consists of Reactive Autonomic Objects (RAO), Reactive Autonomic Components (RAC), Reactive Autonomic Component Groups (RACG) as well as the RAS. The autonomic features are implemented by RAO Leaders (RAOL), RAC Supervisors (RACS) and RACG Managers (RACGM) at the RAC, RACG as well as RAS layer respectively [90]. In this layered architecture model, each tier communicates only with the tier immediately above or

below it. Thus, the independence of those tiers makes their modularity, encapsulation, hierarchical decomposition and reuse possible.

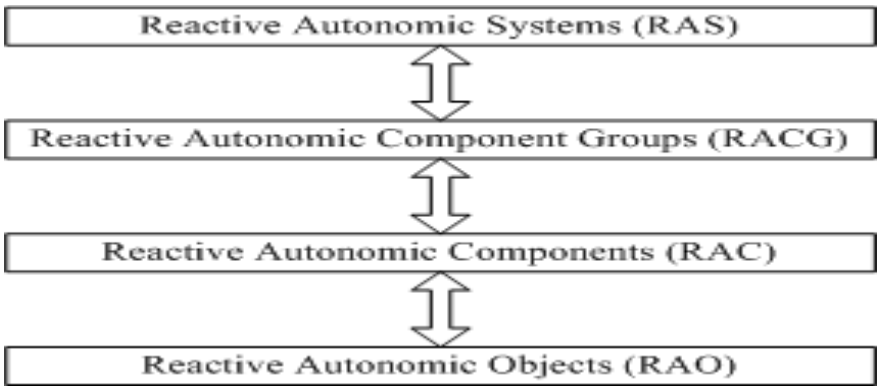


Figure 18: RASF Architecture Model

5.1.1 RAO

The reactive behavior of RAO is modeled as a labeled transition system augmented with ports, resources, attributes and the logical assertion on those attributes as well as time constraints [123]. More specifically, it is modeled as a 9-tuple $(P, \Sigma, \Theta, X, \mathcal{L}, \Phi, \Lambda, Y, R)$ where $P, \Sigma, \Theta, X, \mathcal{L}, \Phi, \Lambda, Y$ are specified as in [134]:

- P is a finite set of ports associated with each port-type and the null-type P_o whose only port is the null port p_o.
- ε is a finite set of events and includes the silent-event tick.
- Θ is a finite set of states where $\theta_0 : \Theta$, is the initial state; there is no final state.
- X is a finite set of typed attributes: abstract data types and port reference types.
- $\mathcal{L}$ is a finite set of Larch traits for the abstract data type used in X.
- Φ is a function-vector (Φ_s, Φ_{at}) which Φ_s associates with each state θ a set of sub states and Φ_{at} associates with each state θ a set of attributes.

- Λ is a finite set of transition specifications between the states.
- Υ is a finite set of time-constraints over the transitions.
- ***R*** models the set of resources available locally for the object to support its functionality.

5.1.2 RAC

RAC is a homogenous set of communicating RAO, where one of the RAO is assigned as a leader (RAOL) of the rest (workers). The workers are responsible for reactive tasks, while the RAOL works on autonomic tasks such as coordinating the self-monitoring at component level. Thus, the RAOL has a different set of states from the workers, which states are autonomic behavior related besides the reactive behavior. The reactive and autonomic natures of formal specifications for the RAOL enable them to implement autonomic functionalities in a real-time reactive system. In order to coordinate the work as well as communication between the RAO, a RAC specification consists of *Members*, *Configure*, *Leader*, *Supervisor*, *Neighbors* and *Repository* (see Figure 19). The RAC is the minimum centralized Reactive Autonomic Element (RAE) that has the ability of self-management in RASF [84].

RAC <name>
    Members: <list of the RAO's names in the RAC>
    Configure: <list of the pairs of communicating members in the RAC>
    Leader: <name of the RAO modeled as a leader for the RAC>
    Supervisor: <name of the RACG's supervisor to which the RAC belongs>
    Neighbors: <list of the RAC's names that belong to the same RACG>
    Repository: <path of the RAC's knowledge base>
End RAC

Figure 19: Specification of the RAC

Similarly to the RAO, the reactive behavior of a RAC that consists of *n* collaborating RAO is specified as a 9-tuple $(P^{syn}, \mathcal{E}^{syn}, \Theta^{syn}, X^{syn}, \mathcal{L}^{syn}, \Phi^{syn}, \Lambda^{syn}, \Upsilon^{syn}, R^{syn})$ [123]:

- P^{syn} is a set of port-types allowing for a synchronous communication between the RAO.
- $\mathcal{E}^{syn}$ is a union of all $\mathcal{E}_i$.
- Θ^{syn} is a finite set of reachable and valid Synchronous Production Machine (SPM) states.
- X^{syn} is a union of the finite sets $X_1^{syn}, \ldots, X_n^{syn}$.
- $\mathcal{L}^{syn}$ is a union of the finite sets of Larch Specification Language (LSL) traits for Abstract Data Type (ADT) used in the RAO.
- Φ^{syn} is a function-vector $(\Phi_s^{syn}, \Phi_{at}^{syn}, \Phi_\gamma^{syn})$ that Φ_s^{syn}associates with each SPM state θ^{syn} a set of sub states; Φ_{at}^{syn} associates with each SPM state θ^{syn} the union of the set of attributes a set of attributes $\Phi_{at\,1}(\theta_1^{syn}), \ldots, \Phi_{at\,n}(\theta_n^{syn})$;
- Λ^{syn} is a finite set of transition specifications between the states.
- Υ^{syn} is a finite set of time-constraints over the transitions.
- R^{syn} is a set of resources available in the RAO; it is defined as a union of all $R_i[i \ldots n]$.

5.1.3 RACG

RACG is a set of RAC that cooperate in fulfillment of group tasks by synchronous communications. The autonomic behavior at group level is coordinated by a supervisor (RACS). Figure 20 shows a RACG specification.

RACG <name>
Members: <list of the RAC's names in the RACG>
Configure: <list of the pairs of communicating members in the RACG>
Supervisor: <name of the RAO modeled as a supervisor for the RACG>
Manager: <name of the RAS's manager to which the RACG belongs>
Neighbors: <list of the RACGs' names that belong to the same RAS>
Repository: <path of the RACG's knowledge base>
End RACG

Figure 20: Specification of the RACG

5.1.4 RAS

RAS is made up of RACG with corresponding communication. It provides an integrated interface for users to delegate tasks and monitor systems. A manager (RACGM) is responsible for coordinating autonomic behavior at system level. Figure 21 illustrates a RAS specification.

RAS <name>
Members: <list of the RACG's names in the RAS>
Manager: <name of the RAO modeled as a manager for the RAS>
Repository: <path of the RAS's knowledge base>
End RAS

Figure 21: Specification of the RAS

5.2 Autonomic Behavior in RASF

In RASF, the autonomic behaviors of RAOL, RACS and RACGM are modeled as the intelligent control loops specified as labeled transition systems (Figure 22), where states specify tasks (*Monitor*, *Analyze*, *Plan*, *Execute*, *HandleException*); events specify triggers from one state to another; transitions specify state sequences under time constraints [85].

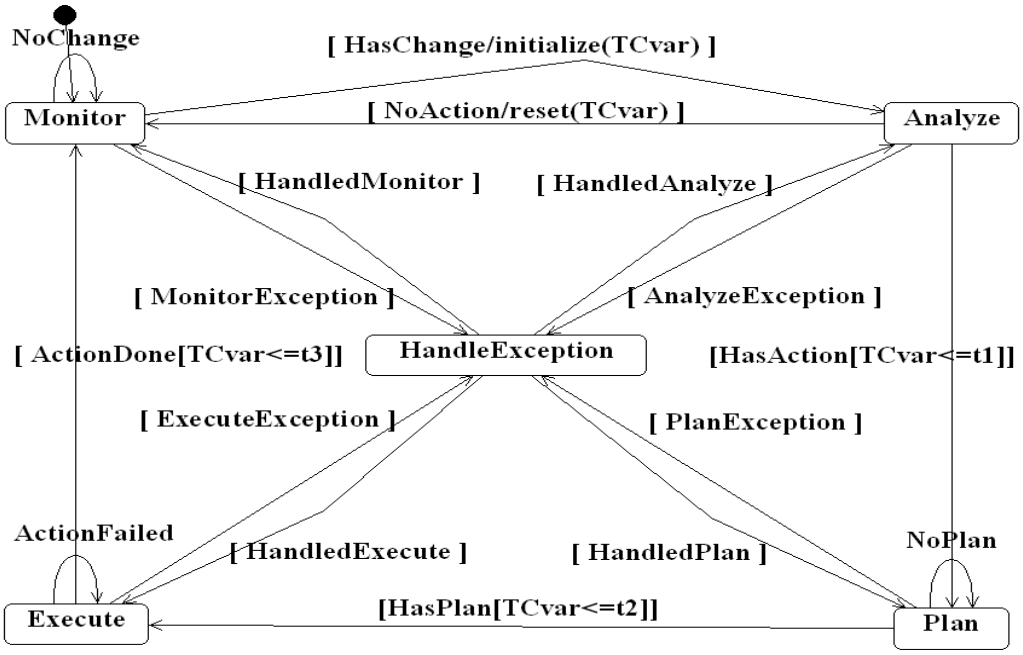


Figure 22: An Example of Intelligent Control Loop

5.2.1 Monitor

Monitor is the first state and it has three internal events which are *NoChange*, *HasChange* and *MonitorException* depending on the evaluated current states of the RAE being monitored. If those states keep the same evaluation as previously, *NoChange* event occurs and the control loop goes back to *Monitor*; otherwise, *HasChange* event occurs and the control loop transits to *Analyze*. The evaluation can be realized with heart-beating messages sent by those RAE which are connected by ports. The configuration changes caused by those RAE's new composition at runtime can also be monitored in a similar way as the state changes [126].

5.2.2 Analyze

The intelligent control loop transits from *Monitor* to *Analyze* triggered by *HasChange* event [176]. Figure 23 depicts the graphical notation of a transition.

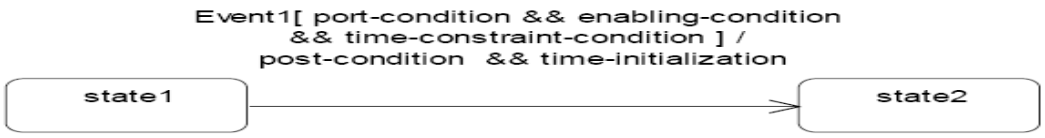


Figure 23: Specification of a Transition

- *Port condition*: a logical assertion on attributes and a port ID *pid*. If an assertion is *true*, then the *pid* can be bound to any port belonging to the port type of the *event* associated with a transition.
- *Enabling condition*: a logical assertion on attributes specifying necessary condition for a transition to take place.
- *Post-condition*: a logical assertion on attributes, and a port ID *pid*. The *post-condition* gives a data computation associated with a transition.
- *Time constraint condition*: a lower bound, an upper bound and an integer named TCvarN (N is a number) that should be initialized to 0 on the transition of a constraining event as the second *Action*. The upper bound must be specified and the lower bound is assumed to be 0 unless otherwise indicated. A constrained event may have zero, one or more disabling states where it cannot be fired. The *time constraint condition* does not apply to transitions that are not constrained events.
- *Action*: occurs when the control loop enters a state and has a format of *post-condition && time constraint initialization*. If it is empty, the *post-condition* is *true*. The *time constraint initialization* does not apply to transitions that are not constraining events.

When the intelligent control loop enters *Analyze* state, certain evaluation is processed. For instance, functionality compliance is assessed by verifying the behavioral correctness

of the RAE being analyzed and comparing their current states (collected in *Monitor* state) with their state machines. If this evaluation is positive, meaning that those RAE's behaviors follow their specifications as well as system policies, then no action is needed and the control loop is back to *Monitor* state triggered by *NoAction* event. However, if the result is negative, the defective RAE is detected and the control loop transits to *Plan* state triggered by *HasAction* event. Similarly, the reliability verification is performed in *Analyze* state by comparing current reliability value with required level when the RAE composition changes at runtime [124].

5.2.3 Plan

After the intelligent control loop enters *Plan* state, a problem solving process is activated and the control loop transits to *Execute* state triggered by *HasPlan* event. For example, if a defective RAE is found after analyzing its functional compliance, there are two options for the control loop. The first option is to provide either switching or repairing plans for users' consideration, which are made up of basic steps represented as knowledge in repositories. The second option is to develop those plans and execute them without users' intervention in *Execute* state when the RAS has enough autonomy and authorization from the users. Similarly for the reliability verification, if an assessment reaches a required level, a changing plan is developed; otherwise, a rejection to the RAE changes with explanation and suggestion is created.

5.2.4 Execute

Execute is the last state of the intelligent control loop. The plans proposed in *Plan* state

are scheduled, executed and validated. Finally, the control loop goes back to *Monitor* state triggered by *ActionDone* event. For instance, in order to replace a defective RAE, a rigorous schedule is needed to state right timing to perform every step of a switching plan, since the control loop stays in a real-time reactive environment and supports hot plugging for the cooperated RAE. After the implementation of that switching plan, a validation between the outcome and plan is processed. If the outcome passes its validation, the control loop enters the *Monitor* state; otherwise, an exception handling mechanism is activated and the control loop transits to *HandleException* state.

5.2.5 Exception Handling

In addition to the states of *Monitor*, *Analyze*, *Plan* and *Execute*, we have modeled *HandleException* state for fault-tolerance of the intelligent control loop. All exceptions to those four states will trigger a *HasException* event and transition to *HandleException* state, which has the benefit of exposing accurate behavior of the control loop when exceptions occur and having a centralized exception handling mechanism. The control loop goes back to one of those four states triggered by *ExceptionHandled* event when exceptions are processed.

Each state of the intelligent control loop can also have its sub-states for more specific behaviors in that state (see Figure 24 as an example for the sub-states of Monitor state).

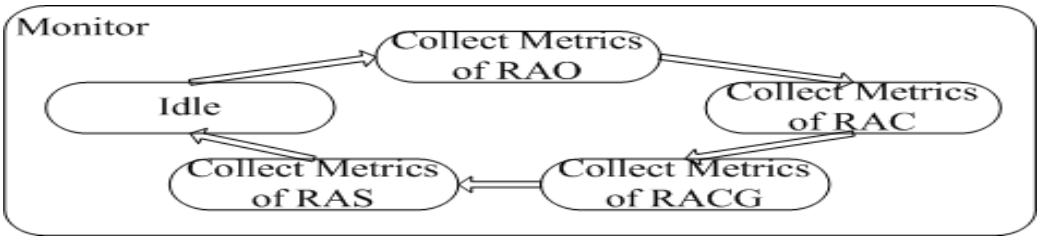


Figure 24: Sub-states of the *Monitor* State

5.3 Mapping RAS Model to MAS Model

This section states the collaboration and supervision to the master students involved in the RASF project for the research work 7) in Figure 2 [62]. In multi-agent community, agent-based approach is considered as a natural way to model autonomic systems, since the ability of an autonomous agent can be easily mapped to the self-management behaviors in autonomic systems. The ability of MAS to make the interactions between components explicitly and control them in a flexible way supports a more distributed complexity [167].

Therefore, the MAS approach is well-suited for autonomic computing systems, and many ideas from the MAS community can be adapted to implement autonomic systems, such as self-management behavior, automatic group formation, agent coordination, evolution, agent adaptation, knowledge mining and interfacing [190].

By applying the MAS approach to implement RAS, the following characteristics of the RAS can be realized [88]:

- Reactivity: agents are reactive.
- Autonomy: agents are autonomous.

- Computational efficiency: a composite computation task can be disassembled to a set of subtasks, which are implemented by several agents in a parallel and distributed way.
- Extendibility: new agents can be easily created and enrolled in the RAS according to dynamic and unpredictable running environment.
- Flexibility: the variety of agents and growing repositories make the RAS adapt to diverse legacy systems.
- Robustness: redundant agents for the same task and their ability of hot swapping greatly improve the RAS fault-tolerance as well as recoverability.

Figure 25 shows a general mapping from RAS to MAS. The elements in *MAS* are layered too; *RAS* is mapped to *MAS*; *RACG* is mapped to *sub-MAS*, which is a sub group of agents; *RAC* are mapped to agents; *RAO* are mapped to agents' plans, goals and beliefs. The *MAS* comprises centralized or distributed *sub-MAS,* which are differentiated by their responsibilities, goals or tasks. The *sub-MAS* contain agent(s), and the agents are grouped by common goals that are differentiated by their individual roles. An agent includes various plans based on agents' beliefs, goals and events. Figure 26 depicts a package diagram of *MAS* which reflects the *RAS hierarchy*. It exhibits a static global view of the overall system. The basic components for the system are *system manager agent*, *supervisor agent* and *regular agent* [62].

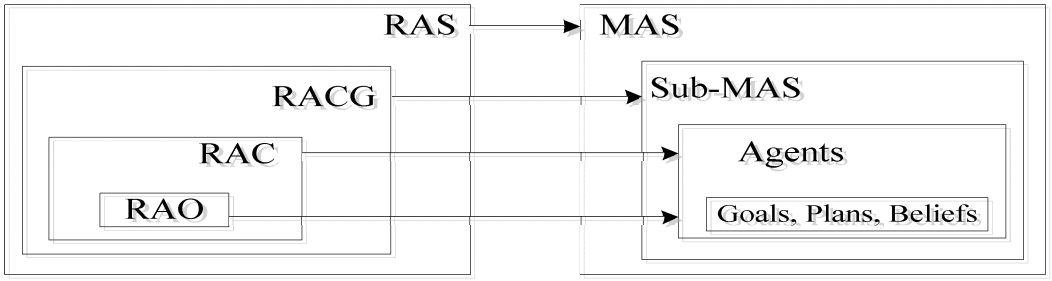


Figure 25: Mapping from RAS to MAS [62]

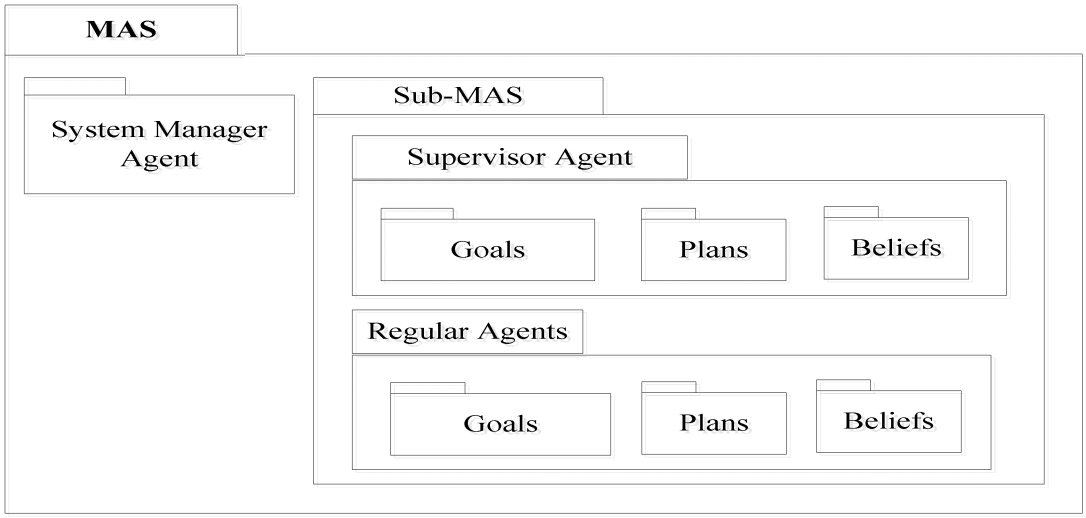


Figure 26: MAS Representation after the Mapping from RAS [62]

System manager agent is the most essential part that acts as a brain for the overall system. It governs and manages the entire system. *Supervisor agent* exists in each multi-agent group (Sub-MAS). It is the group leader that supervises the group. It plays a similar role as the system manager agent but with limited power and localized view of the entire system. *Regular agent* is the worker in the multi-agent society. Each agent in the package has goals, beliefs and plans components [62].

Those agents communicate with each other in order to work together for performing various tasks and they are hierarchical (Figure 27). *Regular agents* are on the bottom level; *system manager agent* is on the top level. All agents can only communicate with

the agents in the same level or the level directly below or above. In this case, *system manager agent* can only converse with *supervisor agents*; *regular agents* are only able to communicate with *supervisor agents*; *supervisor agents* have the ability to send messages to both *system manager agent* and *regular agents*. This design strategy reduces the coupling of agents' communication and assigns system with modularity, encapsulation, hierarchical decomposition as well as reusability [62].

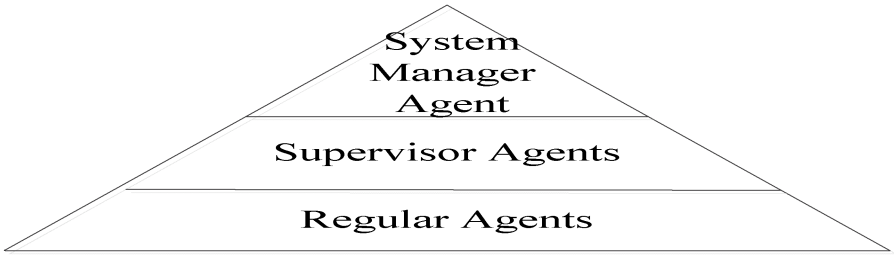


Figure 27: Agent Hierarchy in RASF [62]

There are two communication types in our MAS model: local communication and global communication. Local communication happens only in the group level (Sub-MAS). In a group, *regular agents* communicate with each other to cooperate. If communication issues happen between regular agents, error report messages will be sent to *supervisor agent* by concerned *regular agents*. Based on its beliefs, the *supervisor agent* will make a decision and send messages back to the *regular agents*. The second type is the global communication that happens between Sub-MAS. *Regular agents* cannot communicate with the agents in other groups. *Supervisor agents* can communicate with other *supervisor agents* and *system manager agent*, but they are not allowed to have contacts with the regular agents in other groups. The system manager agent has the ability

to get in touch only with supervisor agents. More details can be found in [62].

5.4 Model Transformation from RAS to MAS Implementation

This section states the collaboration and supervision to the master students involved in RASF project for the research work 8) in Figure 2 [148]. The input of the model transformation process is created using a grammar defined from the RAS model. The outcome of applying that grammar definition is an XML file which represents each type of RAE. A set of transformation rules are applied on that XML file to create the output of the model transformation in Jadex, which is a Java-based MAS-BDI compatible agent programming tool. The output model in Jadex consists of Agent Definition Files (ADF) in XML format, which defines beliefs, goals, message events, plan headers and the plan files in Java code that contain the body of executable plans [148]. Figure 28 illustrates the transformation process.

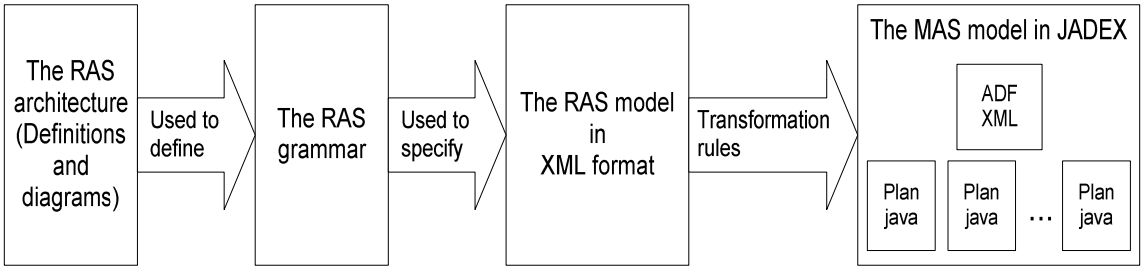


Figure 28: Model Transformation Process from RAS to MAS Implementation [148]

5.4.1 RAS Grammar

The RAS grammar defines the RAE in RASF based on Extended BNF ISO 14977 [70]. Figure 29 shows the grammar of the RAS architecture model. The behavior of RAO and RAC are illustrated by the RAS grammar shown in Figures 30 and 31. More details and

explanation of those figures can be found in [148].

```
RAOL = RAO, repository;
repository = {property}-;
property = name, type, {value}-;
RAC = RAOL, {RAO}-;
RACGM = RAC, RAS_repository;
RACS = RAC, RACG_repository;
RACG = RACS, {RAC}-;
RAS = RACGM, {RACG}-;
```

Figure 29: Grammar of the RAS Architecture Model [148]

```
RAO-behavior = {reactive-atomic}-;
reactive-atomic = reactive-trigger, message;
reactive-trigger = sender, event, receiver;
message = sender, event, receiver;
sender, receiver = RAO | RAOL | RACS | RACGM | ENVIRONMENT;
event = EO | EI | IN | timeout;
timeout = integer;
```

Figure 30: Grammar of the RAO Behavior [148]

```
RAC-behavior = {reactive | self-properties}-;
reactive = {ex-path}-;
ex-path = reactive-trigger | proactive-trigger, {message};
sender, receiver = RAO | RAOL | RACS | RACGM | ENVIRONMENT;
proactive-trigger = sender, IN, receiver;
reactive-trigger = sender, event, receiver;
message = sender, event, receiver;
self-properties = {goal}-;
goal = name, ex-path;
```

Figure 31: Grammar of the RAC Behavior [148]

5.4.2 Input Model of Transformation

The input model for the transformation is the RAS architecture model captured and represented in a XML format. RAO is the atomic element in RASF. Figure 32 depicts the XML specification of the RAO. There is no architectural definition for the RAO because

it is an atomic element, and its reactive behavior is specified by the trigger-response pairs that capture its atomic behavior. RAC is the principal element in RASF and consists of atomic elements RAO with autonomic behavior. The XML specification of RAC consists of *tags* that define its static structure and other *tags* that determine its behavior (see Figure 33). More details and explanation of those figures can be found in [148].

```
<RAO name = "rao-name">
    <REACTIVE-ATOMIC>
        <TRIGGER name= "trigger-name"/>
        <PLAN name= "plan-name"/>
        <RESPONSE name= "response-name"/>
    </REACTIVE-ATOMIC>
</RAO>
```

Figure 32: RAO Specification Template in XML Format [148]

```
<RAC name = "rac-name">
    <MEMBERS>
        <MEMBER name = "rao-name"/>
    </MEMBERS>
    <INTERACTIONS>
        <INTERACTION source = "source-rao" name = "event-name" target =
"target-rao"/>
    </INTERACTIONS>
    <REACTIVE-BEH>
        <LIST-EX-PATH>
        <EX-PATH name = "ex-path-name">
            <TRIGGER>
                <SENDER name = "environment"/>
                <EVENT name = "trigger-name"/>
                <RECEIVER name = "receiver-name"/>
            </TRIGGER>
            <MESSAGE>
                <SENDER name = "sender-name"/>
                <EVENT name = "event-name" type= "event-type">
                    <TIMEOUT min= integer max = integer/>
                </EVENT>
                <RECEIVER name = "receiver-name"/>
```

```
                </MESSAGE>
            </EX-PATH>
            </LIST-EX-PATH>
        </REACTIVE-BEH>
        <SELF-PROP>
            <GOAL name = "goal-name" path = "ex-path-name">
                <EX-PATH>
                </EX-PATH>
            </GOAL>
        </SELF-PROP>
        <LEADER name = "raol-name"/>
        <REPOSITORY>
            <PROPERTY name="property-name"
type="property-type">value</PROPERTY>
        </ REPOSITORY>
</RAC>
```

Figure 33: RAC Specification Template in XML Format [148]

```
<RACG name = "racg-name">
        <MEMBERS>
            <MEMBER name = "rac-name"/>
        </MEMBERS>
        <INTERACTIONS>
            <INTERACTION source = "source-rac" name = "event-name" target =
"target-rac"/>
        </INTERACTIONS>
        <LEADER name = "supervisor-name"/>
        <REPOSITORY>
            <PROPERTY name="property-name"
type="property-type">value</PROPERTY>
        </ REPOSITORY>
</RACG>
```

Figure 34: RACG Specification Template in XML Format [148]

5.4.3 Output Model

The output model for the transformation is the MAS framework in a BDI architecture defined and implemented in Jadex, a Java-based multi agent platform where the agents

are defined by two file formats. The definition of an agent in XML format is stored in an Agent Definition File (ADF); the body of plans in Java language format is stored in a Java source code file. The ADF file consists of different *tags* to implement various concepts of the BDI model.

5.4.4 Transformation Rules

After having the input and output of the model transformation, a set of mapping rules for the XML format of RAS model and MAS model are defined as: 1) <RAC> to <package>; 2) <MEMBERS> to <package>; 3) <RAO> to <agent>; 4) <INTERACTION> to <messageevent>; 5) <LEADER> to <beliefs>; 6) <REPOSITORY> to <beliefs>; 7) <PROPERTY> to <belief>/<beliefset>; 8) <EX-PATH> to <plan>; 9) <MESSAGE> to <plan>; 10) Asynchronous message event rule; 11) Synchronous message event without timeout rule; 12) Synchronous message event with timeout rule; 13) Empty message event rule; 14) <REACTIVE-BEHAVIOR> vs. <SELF-PROPERTY>; 15) <GOAL> to <achievegoal> [148].

5.5 RASF Process Model

In order to develop a reactive autonomic system using RASF, we can follow the RASF process model indicated below based on the Figure 2 in Section 1.3.

Phase 1 (arrow 2): Build a RAS model (box 3) based on the RAS requirements (box 1). This model includes: i) RAS architecture model (Figure 21), ii) reactive behavior of RAE (Section 5.1.1 & 5.1.2), iii) the specification of RAE (Figure 23 & 24) and iv) specification of the self-* properties self-healing (Section 7.1, 7.3 & 7.5) as well as

self-configuration (Section 8.1, 8.3 & 8.5), that includes autonomic behavior (intelligent control loop) of the RAE (Section 5.2).

Phase 2 (arrow 4): Transform the RAS model in Phase 1 to its categorical model (box 5) using the category theory. This transformation includes the categorical models of the structure (Section 6.1), the behavior (Section 6.2) and the self-* properties self-healing (Section 7.2, 7.4, 7.6 & 7.7) as well as self-configuration (Section 8.2, 8.4, 8.6 & 8.7) in the RAS model together with corresponding XML representation (Section 7.8 & 8.8).

Phase 3 (arrow 6): Transform the RAS model in Phase 1 to its MAS model (box 7). This transformation includes the mapping from the RAS architecture model to the MAS architecture model (Figure 28), from RAS specification to MAS specification (Figure 29) and from the RAS behavior to the MAS behavior (Section 5.3).

Phase 4 (arrow 12): Transform MAS model in Phase 3 to its categorical model (box 13) using category theory. This transformation includes the categorical models of plans (Section 6.5.1), goals (Section 6.5.2), beliefs (Section 6.5.3), agents (Section 6.5.4) as well as repository (Section 6.5.5) in the MAS model together with corresponding XML representation (Section 6.6).

Phase 5 (arrow 10): Visualize the categorical RAS model in Phase 2 to its graphical representation (box 11) by importing its XML representation in Phase 2 to our graphical illustration tool CATCanvas (Section 6.4.2).

Phase 6 (arrow 14): Visualize the categorical MAS model in Phase 4 to its graphical representation (box 15) by importing its XML representation in Phase 4 to our graphical

illustration tool CATCanvas (Section 6.4.2).

Phase 7 (verification line between box 5 and box 13): Verify if the categorical MAS model in Phase 4 conforms to the categorical RAS model in Phase 2. This verification is achieved through comparing their XML representations generated in Phase 4 and Phase 2 against their mapping rules extended from Phase 3 with category theory. This phase is one of our future work directions.

Phase 8 (the validation line between box 11 and box 15): Validate if the graphical illustration of categorical MAS model in Phase 6 conforms to the graphical illustration of categorical RAS model in Phase 5. This validation can be achieved by comparing the XML representations exported from the CATCanvas in Phase 6 and Phase 5 against their mapping rules extended from Phase 3 with CML (Section 6.4.1). This phase is one of our future work directions.

Phase 9 (arrow 8): Transform the MAS model in Phase 3 above to its implementation (box 9) using the Jadex framework. This transformation (Figure 31) includes the RAS grammar (Section 5.4.1), input model (Section 5.4.2), output model (Section 5.4.3) and transformation rules (Section 5.4.4).

Phase 10 (arrow 16): Transform the MAS implementation in Phase 9 to categorical model (box 17) using category theory. This phase is one of our future work directions.

Phase 11 (arrow 18): Visualize the categorical MAS implementation in Phase 10 to its graphical representation (box 19) by importing its XML representation in Phase 10 to our graphical illustration tool CATCanvas (Section 6.4.2). This phase is one of our future

work directions.

Phase 12 (the verification line between box 13 and box 17): Verify if the categorical MAS implementation in Phase 10 conforms to the categorical MAS model in Phase 4. This verification is achieved by comparing their XML representations generated in Phase 10 and Phase 4 against their mapping rules extended from Phase 9 with category theory. This phase is one of our future work directions.

Phase 13 (the validation line between box 15 and box 19): Validate if the graphical illustration of the categorical MAS implementation in Phase 11 conforms to the graphical illustration of the categorical MAS model in Phase 6. This validation can be achieved through comparing the XML representations exported from the CATCanvas in Phase 11 and Phase 6 against their mapping rules extended from Phase 9 with CML (Section 6.4.1). This phase is one of our future work directions.

5.6 RASF Tooling Support

After having the RASF process model, we developed a tool named RASF Integration Tool (RASFIT) to support the models, specifications, transformations, verifications and validations in the RASF process model we introduced earlier in Section 5.5.

RASFIT is an Eclipse [202] plug-in based solution that extends the Eclipse IDE with i) a UML design tool (Enterprise Architect [203]), ii) a framework in terms of building the multi-agent applications named Jadex [204], iii) a model transformation framework [148] to produce the multi-agent templates representing the RAS components that satisfy both reactive and autonomic properties and iv) a graphical tool for illustrating categorical

models [81].

The architecture of RASFIT (Figure 227) consists of i) an Eclipse plug-in module (Section 9.1.1) that is responsible for the interactions from end users through the Eclipse IDE by using the Eclipse API, ii) an Enterprise Architecture (EA) module (Section 9.1.2) which is responsible for modeling RAE through the integrated EA IDE in the Eclipse IDE by using the EA API, iii) a Jadex module (Section 9.1.3) that is responsible for modeling the MAS implementation from the RAS model through the integrated Jadex IDE in the Eclipse IDE by using the Jadex API, iv) a model transformation module (Section 9.1.4) which is responsible for transforming the RAS model represented in a XML format to the MAS implementation represented in the format of agent definition files, defining beliefs, goals, message events, plan headers and related plan files in Java code that contain the body of executable plans [148], and v) a CATCanvas module (Section 9.1.5) that is for graphically illustrating a RAS model represented in a XML format by importing its XML file [81].

5.7 Summary

In this chapter, we have given a conceptual view of the RASF and conveyed architectural decisions for further design as well as implementation of the RASF.

The 4-tier architecture for the RASF consists of the RAO, RAC, RACG and RAS; the autonomic behavior, such as self-monitoring or self-analyzing, is implemented by the component leaders (RAOL), group supervisors (RACS) and system managers (RACGM) at the RAC, RACG as well as RAS tier respectively. The RAO is modeled as a labeled

transition system augmented with ports, attributes, logical assertion on the attributes, time constraints and resource specification; the RAC is a set of synchronously communicating RAO, where one of the RAO is assigned as a leader of the rest (workers); the RACG is a set of centralized or distributed RAC which cooperate in fulfillment of group tasks by asynchronous communication; the RAS is made up of centralized or distributed RACG with their asynchronous communication.

Finally, we have presented the autonomic behavior (intelligent control loop) in RASF, mapping from RAS model to MAS model, model transformation from RAS to the MAS implementation and the introduction of RASF process model as well as tooling support. We will explain the categorical specification of RASF in next chapter according to the RAS model, MAS model and category theory we described in this chapter and Chapter 3 respectively.

Chapter 6: Categorical Specifications of RASF

This chapter states the research activity 8), 18), 19) in Figure 3, which are prototype design of categorical RASF model, transformation from categorical RAS model to its XML specification as well as transformation from categorical MAS model to its XML specification, and describes the contribution of the author (Section 6.1, 6.2, 6.3 and 6.6) as well as the collaboration and supervision to the master students involved in the RASF project (6.4 & 6.5). I had one publication [90] and one in preparation for this chapter.

Category theory can provide abstract hierarchical types which are useful to build complex data types for RAS. This will allow them to be modeled in isolation thus a modular treatment is possible. Moreover, their hierarchical nature allows the axioms of a subcategory to be inherited from its category, which gives a consistent approach to data abstraction [115] at each stage during the development and usage of RASF, such as requirement specifications, RAS modeling, MAS modeling, model transformation and implementation.

The RAE in the RAS can change state, be manipulated, or connected to other RAE; the categorical invariant properties, such as limit and colimit (see Definition 3.5.6 & 3.5.9), are used to characterize those RAE and their behavior. The basic building blocks for the RAS description are provided by the categorical notions of objects, morphisms, categories, functors and natural transformations; deriving and composing structures are achieved through canonical constructors, universal constructors as well as categorical invariant properties (see Section 3.5).

6.1 Categorical Model of Structure in RASF

This section states the research activity 8) in Figure 3. Based on the architecture model in RASF, the internal structure of a RAS is both hierarchical (composition of its RAE) and recursive (tiered interconnectivity of its RAE). Therefore, the behavior of a RAS can be characterized by its RAE and their interactions. A categorical framework for that behavior can be freely generated below.

Property 6.0.1: *Type* is a category where objects represent the object types denoted by *ObjType(Type)* and morphisms represent the morphism types as *MorType(Type)*. For example, *MyCategory* is a category where objects are denoted by *Obj(MyCategory)* and morphisms denoted by *Mor(MyCategory)*. There is a functor *F* from *MyCategory* to *Type* which maps each object of *MyCategory* to a type (an object of *Type*): *F(Obj(MyCategory)) = ObjType(Type)* and maps each morphism of *MyCategory* to a type (a morphism of *Type*): *F(Mor(MyCategory)) = MorType(Type)*.

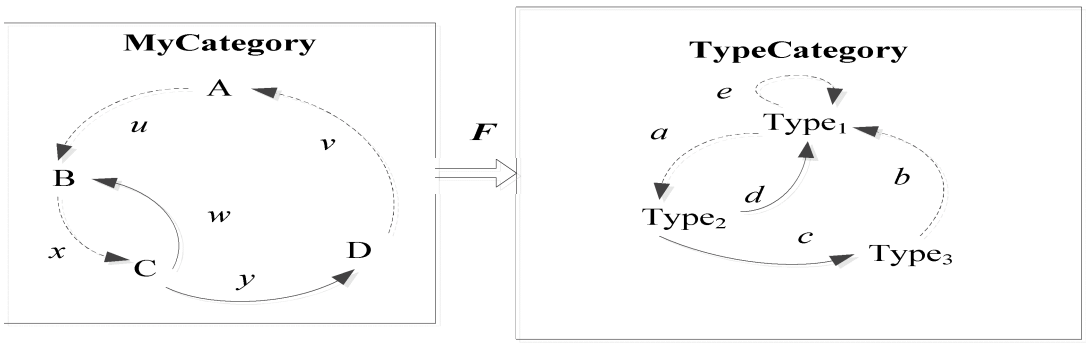


Figure 35: An Example of Type Category

In Figure 35, a *type category* called ***TypeCategory*** contains objects: $type_1$, $type_1$ and $type_3$; type *m* morphisms: *c* and *d*; type *n* morphisms: *a, b* and *e*. ***MyCategory*** contains

objects: *A, B, C* and *D*; morphisms: *u, v, w, x* and *y*. Functor ***F*** maps *MyCategory* objects and morphisms to types in *TypeCategory*: *F(A) = Type$_1$, F(B) = Type$_1$, F(C) = Type$_2$, F(D) = Type$_3$, F(u) = e (type n), F(v) = b (type m), F(w) = d (type m), F(x) = a (type n) and F(y) = c (type m).*

Property 6.0.2: the null object *Object$_{Null}$* in a category does not have any real meaning or content and it does not have any relationship with other objects. *Object$_{Null}$* and its identity morphism are useful for catching "non-useful" or "non-related" objects as well as morphisms from other categories by defined functors (relations).

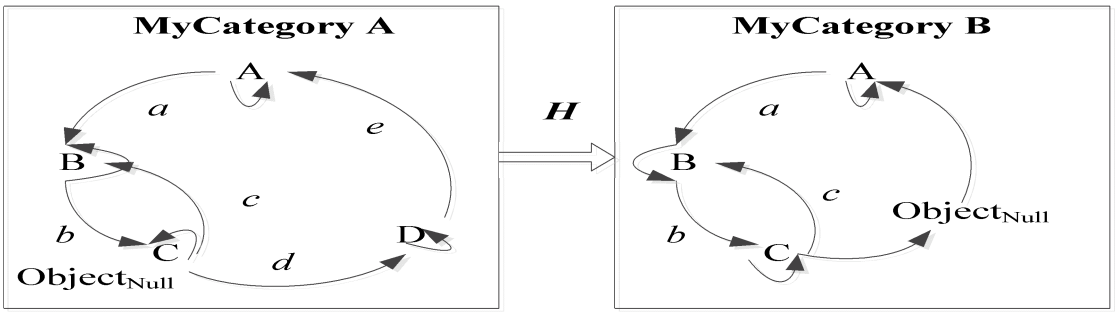


Figure 36: An Example of Null Object in Category

Figure 36 is an example of using ***Object$_{Null}$***. ***MyCategory A*** contains objects: *A, B, C*, *D* and ***Object$_{Null}$***; morphisms: *a, b, c, d* and *e*. ***MyCategory B*** has objects: *A, B*, *C* and ***Object$_{Null}$***; morphisms: *a, b* and *c*. Functor ***H*** maps ***MyCategory A*** objects and morphisms to ***MyCategory B***: *H (A) = A, H (B) = B, H (C) = C, H (D) =* ***Object$_{Null}$****, H (****Object$_{Null}$****) =* ***Object$_{Null}$****, H (a) = a, H (b) = b, H (c) = c, H (d) = id* $_{\textbf{ObjectNull}}$ and *H (e) = id* $_{\textbf{ObjectNull}}$. From this example, we can see ***MyCategory B*** contains all the objects and morphisms in ***MyCategory A*** except for object *D* and its related morphisms *d* and *e.*

Property 6.0.3: A directed graph *G* is a set *O* of objects called vertices or nodes, and

a set *A* of ordered pairs of vertices are called arrows or directed edges [93]. Every arrow diagram or directed graph can be specified as a category named **PATH** where morphisms are sequences (paths) of arrows. One can create a directed graph by drawing an arrow from *x* to *y* where *x, y* ∈ the same set *X*, which can be associated with the category denoted by **PATH (X)** or **PATH [**130**]**. The objects are elements in *X* and the morphisms are all sequences (paths) of the adjacent arrows. This naturally defines a *composition of arrows*. This viewpoint leads to a general categorical semantics for the relational structures. Vice versa, every category is a graphical structure.

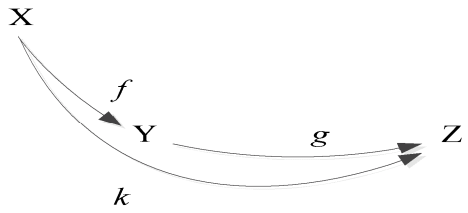
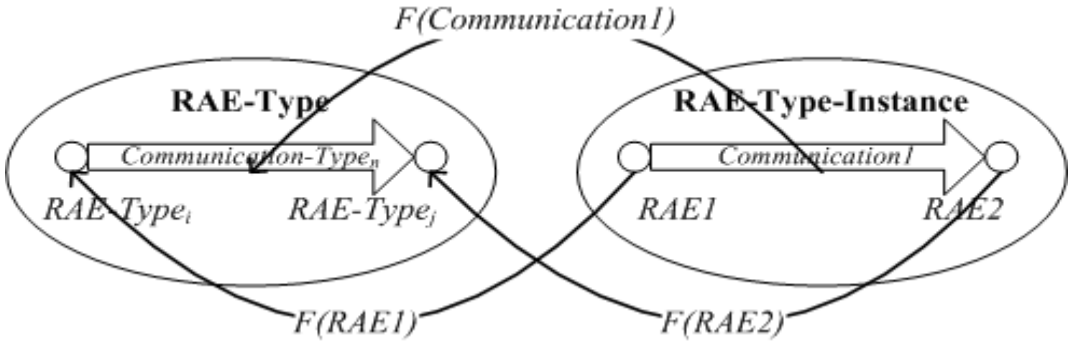


Figure 37: An Example of PATH Category

Figure 37 is an example of PATH. For morphisms (arrows) *f: x → y*, *g: y → z* and morphism *k: x → z*, if *f, g* and *k* are of the same type, then *k* is not considered as a direct arrow since *k* equals to the sequence (path) of the consecutive arrows (*f* and *g*). By the definition of PATH, the lengths of the sequences *f* and *g* are one, and the length of *k* is two. The existence of the identity arrow for each object will always be assumed by definition, and it can be interpreted as sequences of length zero.

Property 6.1.1: **RAE-Type** is an instance category of the **Type** category (see Property 6.0.1) in which objects represent the RAE types denoted by *ObjType(***RAE-Type***)* and morphisms represent the morphism types denoted by *MorType(***RAE-Type***)*.

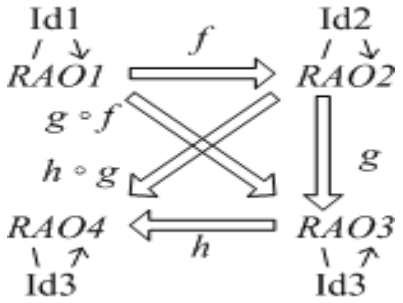
For example, **RAE-Type-Instance** is the category in which objects are denoted by *Obj(***RAE-Type-Instance***)* and morphisms denoted by *Mor(***RAE-Type-Instance***)*. There is a functor *F* from **RAE-Type-Instance** to **RAE-Type**, a structure-preserving mapping of the objects of **RAE-Type-Instance** to objects of **RAE-Type**: *F(ObjType(***RAE-Type-Instance***))* = *Obj(***RAE-Type***)*, and of the morphisms of **RAE-Type-Instance** to the morphisms of **RAE-Type**: *F(MorType(***RAE-Type-Instance***))* = *Mor(***RAE-Type***)* as the Definition 3.1.2 (see the figure below).



Property 6.1.2: RAC can be specified as a category **RAC** with a set of objects |**RAC**| and morphisms so that for each $RAO_i, RAO_j \in$ |**RAC**|, there is a set of morphisms f : **RAC**(RAO_i, RAO_j) mapping the RAO_i to RAO_j which indicate the communication between them as f : $RAO_i \rightarrow RAO_j$ (see Definition 3.1.1).

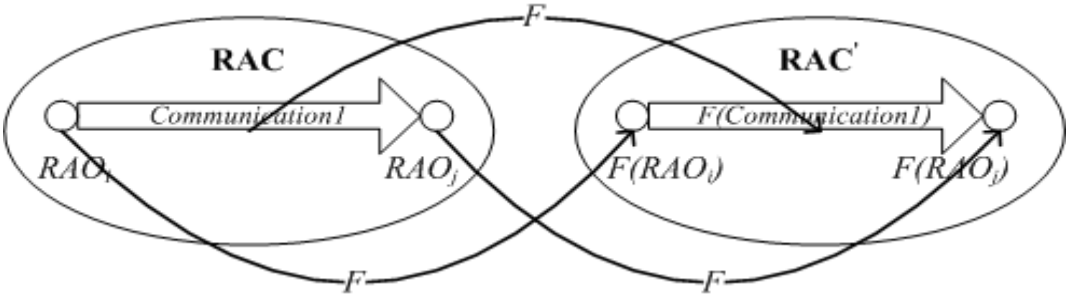
Proof. *All what we need is to prove: i) the existence of composition and identity morphism, and ii) prove associativity. Let RAO_1, RAO_2 and RAO_3 be three RAO such that RAO_1 can interact with RAO_2, which can interact with RAO_3. Then RAO_1 can interact with RAO_3 (indirectly through RAO_2), which means the existence of a composition of morphisms between RAO_1 and RAO_3. The identity morphism does exist as a natural*

representation of internal interactions. Let f, g and h be the morphisms such that f: RAO_1 → RAO_2, g: RAO_2 → RAO_3 and h: RAO_3 → RAO_4. It is clear that h ∘ (g ∘ f) = (h ∘ g) ∘ f.■



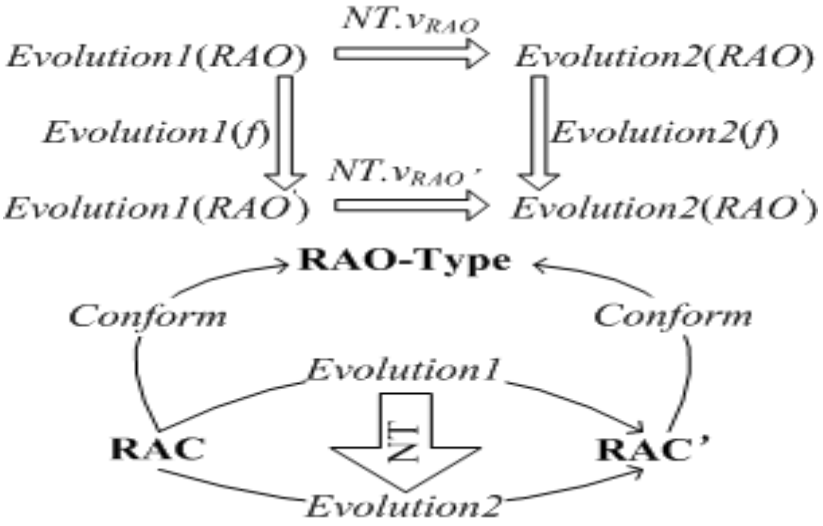
Every category ***C*** has an identity functor $1_{\boldsymbol{C}} : \boldsymbol{C} \rightarrow \boldsymbol{C}$. We can then easily demonstrate the following property.

Property 6.1.3: The evolutions of a RAC, because of self-adaptation as well as self-organization (original behavior is preserved) are modeled with functors. For instance, an evolution from RAC to RAC' is specified as a functor *F*, a structure-preserving mapping of the objects (*RAO*) in **RAC** to the objects (*RAO'*) in **RAC'** (*F*: |**RAC**| → |**RAC'**|), and of the morphisms in **RAC** to morphisms in **RAC'** (*F*: **RAC**(RAO_i, RAO_j) → **RAC'**(*F*(RAO_i), *F*(RAO_j))) as the Definition 3.1.2 (see the figure below).



In category theory, the natural transformation provides a way of transforming one functor into another while respecting the internal structure (i.e. the composition of morphisms) of the categories involved (see Definition 3.6.3).

Property 6.1.4: Because the evolutions of RAC are specified as functors from the category **RAC** to **RAC**', the mapping of those evolutions can be represented by the natural transformation: υ : *Evolution1* → *Evolution2* is a family of the arrows in **RAC**': v_{RAO}: *Evolution1*(*RAO*) → *Evolution2*(*RAO*), such that for any *f* : *RAO* → *RAO*' in **RAC**, there exists $v_{RAO} \circ$ *Evolution1*(*f*) = *Evolution2*(*f*) $\circ$ v_{RAO} as indicated in the figures below, and v_{RAO} is the component of the natural transformation υ (see Definition 3.6.3). In addition, both **RAC** and **RAC**' should have the functor *Conform* to their index (type) category **RAO-Type** in terms of structure-preserving mapping.

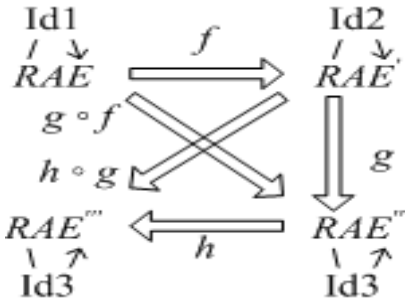


Property 6.1.5: All possible evolutions (functors) and their relationships (natural transformations) for the RAE can be specified as a functor category **Fun**(**RAE**, **RAE**'), where objects are functors *Evolution* : **RAE** → **RAE**' and morphisms are their natural transformations υ : $Evolution_i$ → $Evolution_j$ (see Definition 3.6.4).

If we regard **RAE** and **RAE**' as objects instead of categories by ignoring the object details inside **RAE** and **RAE**', all possible evolutions for the RAE can be specified as a

category **RAE-Evolution** where objects are RAE, RAE', RAE'', etc., and morphisms are those evolutions, such as *Evolution1*, *Evolution2*, etc.

Proof. *All what we need is to prove: i) the existence of composition and identity morphism, and ii) prove associativity. Let RAE' and RAE'' be two evolutions for the RAE such that RAE can evolve to RAE', which can evolve to RAE''. Then RAE can evolve to RAE'' (indirectly through RAE'), which means the existence of a composition of the morphisms between RAE and RAE''. The identity morphism does exist as a natural representation of internal evolutions. Let f, g and h be the morphisms such that f: RAE → RAE', g: RAE' → RAE'' and h: RAE'' → RAE'''. It is clear that h ∘ (g ∘ f) = (h ∘ g) ∘ f.*■

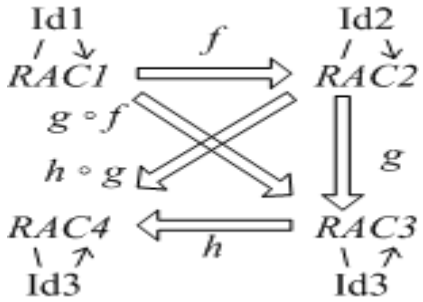


Property 6.1.6: RACG can be specified as a category **RACG** with a set of full subcategories and products such that for each $\mathbf{RAC}_m, \mathbf{RAC}_n \subseteq \mathbf{RACG}$, there is a set of products $P : \mathbf{RAC}_m \times \mathbf{RAC}_n$ has objects of the form (RAO_m, RAO_n) for $RAO_m \in |\mathbf{RAC}_m|$, $RAO_n \in |\mathbf{RAC}_n|$ and arrows of the form $(f, g) : (RAO_m, RAO_n) \rightarrow (RAO_m', RAO_n')$ for $f : RAO_m \rightarrow RAO_m' \in |\mathbf{RAC}_m|$ and $g : RAO_n \rightarrow RAO_n' \in |\mathbf{RAC}_n|$ (see Definition 3.2.1 & Proposition 3.6.2).

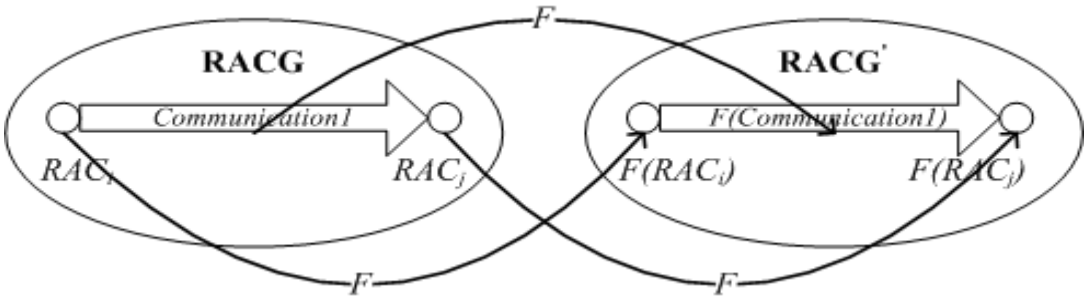
Property 6.1.7: The RACG may also be specified as a category **RACG** with a set of

objects |**RACG**| and morphisms such that for each $RAC_m, RAC_n \in$ |**RACG**|, there is a set of morphisms f : **RACG**(RAC_m, RAC_n) mapping the RAC_m to the RAC_n that indicate the communication between them as $f: RAC_m \rightarrow RAC_n$ (see Definition 3.1.1).

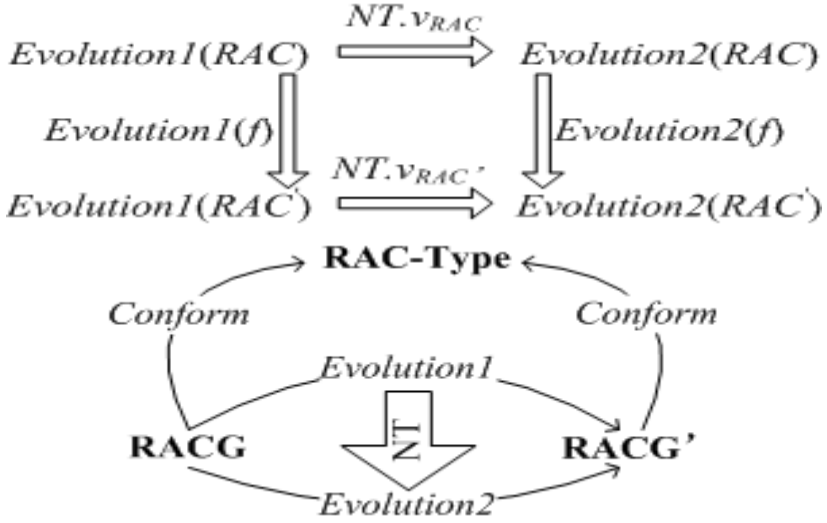
Proof. *All what we need is to prove: i) the existence of composition and identity morphism, and ii) prove associativity. Let RAC_1, RAC_2 and RAC_3 be three RAC such that RAC_1 can interact with RAC_2, which can interact with RAC_3. Then RAC_1 can interact with RAC_3 (indirectly through RAC_2), which means the existence of a composition of morphisms between RAC_1 and RAC_3. The identity morphism does exist as a natural representation of internal interactions. Let f, g and h be the morphisms such that f: RAC_1 $\rightarrow$ RAC_2, g: $RAC_2 \rightarrow RAC_3$ and h: $RAC_3 \rightarrow RAC_4$. It is clear that $h \circ (g \circ f) = (h \circ g) \circ f$.*■



Property 6.1.8: The evolutions of a RACG, because of self-adaptation as well as self-organization (original behavior is preserved) are modeled with functors. For instance, an evolution from RACG to RACG' is specified as a functor *F*, a structure-preserving mapping of the objects (*RAC*) in **RACG** to the objects (*RAC'*) in **RACG'** (*F*: |**RACG**| $\rightarrow$ |**RACG'**|), and of the morphisms in **RACG** to the morphisms in **RACG'** (*F*: **RACG** (RAC_i, RAC_j) $\rightarrow$ **RACG'**($F(RAC_i)$, $F(RAC_j)$)) as the Definition 3.1.2 (see the figure below).



Property 6.1.9: Because the evolutions of RACG are specified as functors from the category **RACG** to **RACG**', the mapping of those evolutions can be represented by the natural transformation: υ : *Evolution1* → *Evolution2* is a family of the arrows in **RACG**': υ_{RAC}: *Evolution1*(*RAC*) → *Evolution2*(*RAC*), such that for any f : $RAC \rightarrow RAC'$ in **RACG**, there exists $\upsilon_{RAC} \circ Evolution1(f) = Evolution2(f) \circ \upsilon_{RAC}$ as indicated in the figures below, and υ_{RAC} is the component of the natural transformation υ (see Definition 3.6.3). In addition, both **RACG** and **RACG**' should have the functor *Conform* to their index (type) category **RAC-Type** in terms of structure-preserving mapping.

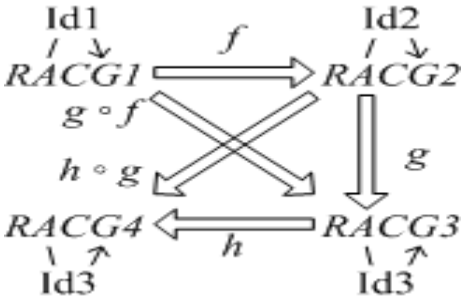


Property 6.1.10: RAS can be specified as a category **RAS** with a set of full subcategories and products such that for each $\mathbf{RACG}_m, \mathbf{RACG}_n \subseteq \mathbf{RAS}$, there is a set of

products P : $\mathbf{RACG}_m \times \mathbf{RACG}_n$ has objects of the form (RAC_m, RAC_n) for $RAC_m \in |\mathbf{RACG}_m|$, $RAC_n \in |\mathbf{RACG}_n|$ and arrows of the form (f, g) : (RAC_m, RAC_n) → (RAC_m', RAC_n') for $f : RAC_m \rightarrow RAC_m' \in |\mathbf{RACG}_m|$ and $g : RAC_n \rightarrow RAC_n' \in |\mathbf{RACG}_n|$ (see Definition 3.2.1 & Proposition 3.6.2).

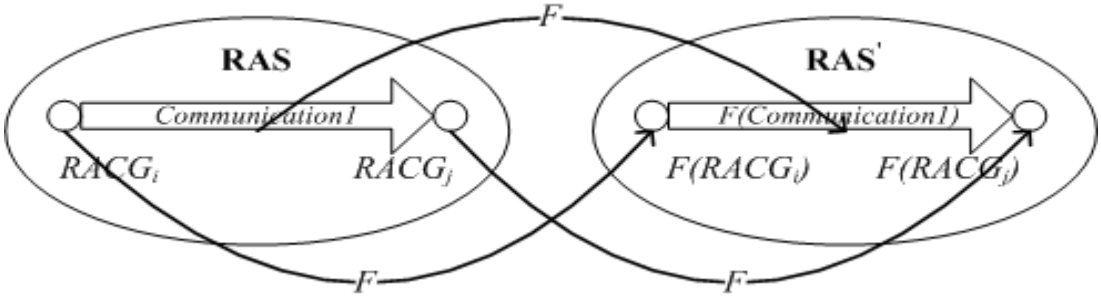
Property 6.1.11: The RAS may also be specified as a category **RAS** with a set of objects |**RAS**| and morphisms such that for each $RACG_x, RACG_y \in$|**RAS**|, there is a set of morphisms f : **RAS**($RACG_x, RACG_y$) mapping the $RACG_x$ to the $RACG_y$ that indicate their interactions as $f : RACG_x \rightarrow RACG_y$ (see Definition 3.1.1).

Proof. *All what we need is to prove: i) the existence of composition and identity morphism, and ii) prove associativity. Let $RACG_1$, $RACG_2$ and $RACG_3$ be three RACG such that $RACG_1$ can interact with $RACG_2$, which can interact with $RACG_3$. Then $RACG_1$ can interact with $RACG_3$ (indirectly through $RACG_2$), which means the existence of a composition of morphisms between $RACG_1$ and $RACG_3$. The identity morphism does exist as a natural representation of internal interactions. Let f, g and h be the morphisms such that f: $RACG_1 \rightarrow RACG_2$, g: $RACG_2 \rightarrow RACG_3$ and h: $RACG_3 \rightarrow RACG_4$. It is clear that* $h \circ (g \circ f) = (h \circ g) \circ f$.■

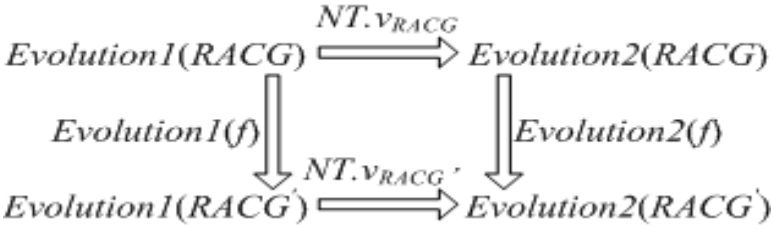


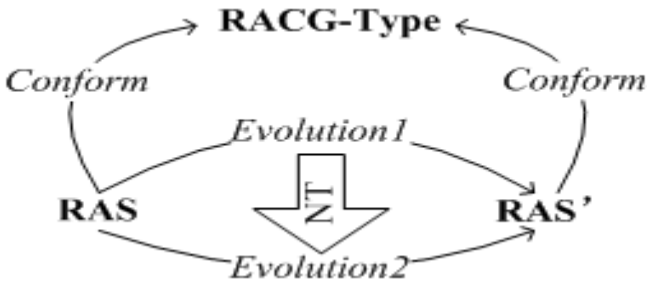
Property 6.1.12: The evolutions of a RAS, because of self-adaptation as well as self-

organization (original behavior is preserved) are modeled with functors. For instance, an evolution from RAS to RAS' is specified as a functor *F*, a structure-preserving mapping of the objects (*RACG*) in **RAS** to the objects (*RACG'*) in **RAS'** (*F*: |**RAS**| → |**RAS'**|), and of the morphisms in **RAS** to the morphisms in **RAS'** (*F*: **RAS**($RACG_i$, $RACG_j$) → **RAS'**($F(RACG_i)$, $F(RACG_j)$)) as the Definition 3.1.2 (see the figure below).

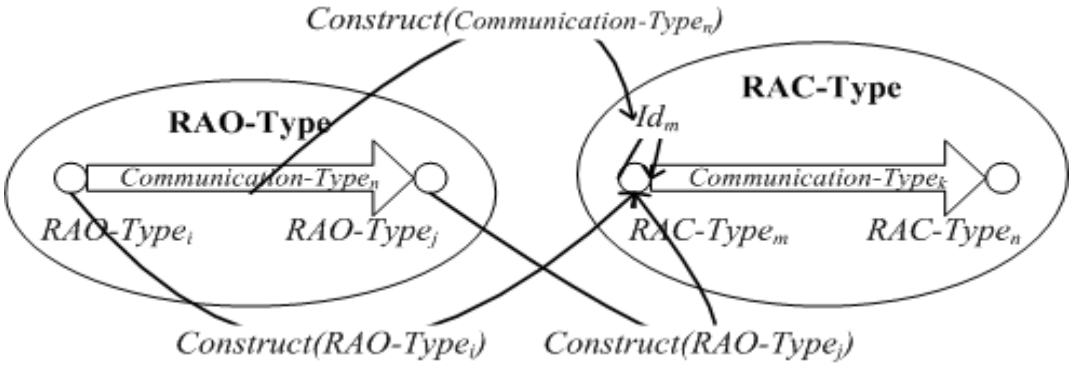


Property 6.1.13: Because the evolutions of RAS are specified as functors from the category **RAS** to **RAS'**, the mapping of those evolutions can be represented by the natural transformation: υ : *Evolution1* → *Evolution2* is a family of the arrows in **RAS'**: v_{RACG}: *Evolution1*(*RACG*) → *Evolution2*(*RACG*), such that for any *f* : *RACG* → *RACG'* in **RAS**, there exists $v_{RACG} \circ Evolution1(f) = Evolution2(f) \circ v_{RACG}$ as indicated in the figures below, and v_{RAC} is the component of the natural transformation υ (see Definition 3.6.3). In addition, both **RAS** and **RAS'** should have the functor *Conform* to their index (type) category **RACG-Type** in terms of structure-preserving mapping.



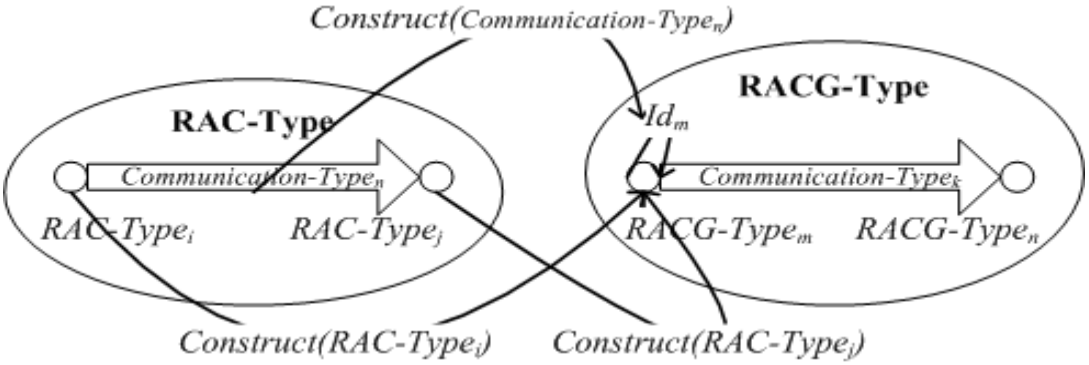


Property 6.1.14: The constraints on constructing a RAC type from a group of RAO types can be specified as a functor *Construct* from the category **RAO-Type** to category **RAC-Type**, a structure-preserving mapping of the objects (*RAO-Type*) in **RAO-Type** to the objects (*RAC-Type*) in **RAC-Type** (*F*: |**RAO-Type**| → |**RAC-Type**|), and of the morphisms in **RAO-Type** to the identity morphisms in **RAC-Type** (*F*: **RAO-Type**(*RAO-Type*$_i$, *RAO-Type*$_j$) → **RAC-Type**(*RAC-Type*$_m$, *RAC-Type*$_m$)) as Definition 3.1.2 (see the figure below).

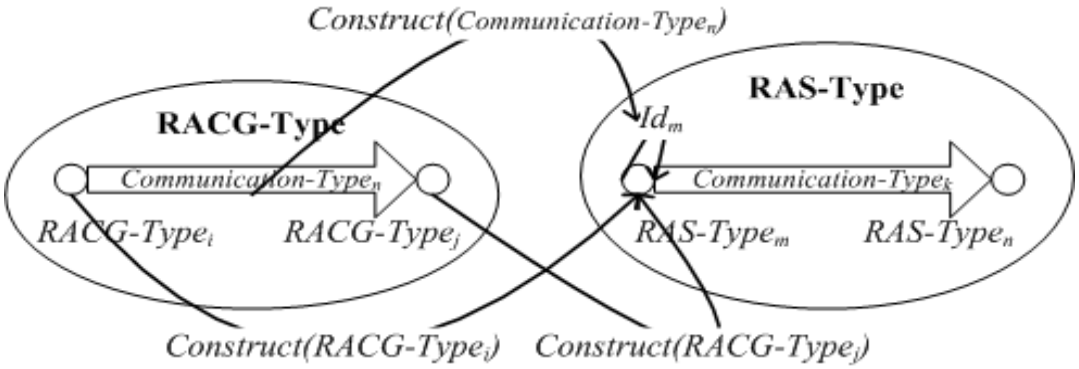


Property 6.1.15: The constraints on constructing a RACG type from a group of RAC types can be specified as a functor *Construct* from the category **RAC-Type** to category **RACG-Type**, a structure-preserving mapping of the objects (*RAC-Type*) in **RAC-Type** to the objects (*RACG-Type*) in **RACG-Type** (*F*: |**RAC-Type**| → |**RACG-Type**|), and of the morphisms in **RAC-Type** to the identity morphisms in **RACG-Type** (*F*: **RAC-Type** (*RAC-Type*$_i$, *RAC-Type*$_j$) → **RACG-Type**(*RACG-Type*$_m$, *RACG-Type*$_m$)) as Definition

3.1.2 (see the figure below).



Property 6.1.16: The constraints on constructing a RAS type from a group of RACG types can be specified as a functor *Construct* from the category **RACG-Type** to category **RAS-Type**, a structure-preserving mapping of the objects (*RACG-Type*) in **RACG-Type** to the objects (*RAS-Type*) in **RAS-Type** (*F*: |**RACG-Type**| → |**RAS-Type**|), and of the morphisms in **RACG-Type** to the identity morphisms in **RAS-Type** (*F*: **RACG-Type** (*RACG-Type*$_i$, *RACG-Type*$_j$) → **RAS-Type**(*RAS-Type*$_m$, *RAS-Type*$_m$)) as Definition 3.1.2 (see the figure below).



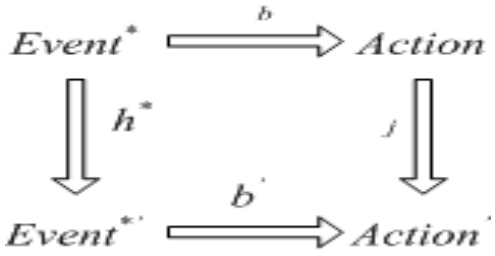
6.2 Categorical Model of Behavior in RASF

This section states the research activity 8) in Figure 3.

Definition 6.2.1: A *monoid* (sometimes called semi-group with unit) is a set M equipped with a binary operation $\cdot : M \times M \rightarrow M$ and a distinguished "unit" element

$u \in M$ such that for all $x, y, z \in M$, $x \cdot (y \cdot z) = (x \cdot y) \cdot z$ and $u \cdot x = x = x \cdot u$. Equivalently, a monoid is a category with just one object. The arrows of the category are the elements of the monoid. In particular, the identity arrow is the unit element u. Composition of arrows is the binary operation $m \cdot n$ of the monoid [6].

Property 6.2.2: The behavior of a RAE can be specified as an automation that consists of an input set *Event*, a state set *State*, an output set *Action*, a transition function $f : Event \times State \rightarrow State$, an initial state $State_0 \in State$, and an output function $g : State \rightarrow Action$. The behavior of that automata is a function $b : Event^* \rightarrow Action$ (from the monoid $Event^*$ to *Action*). A category **RAE-Behavior** of that behavior has the objects as pairs ($Event$, $b : Event^* \rightarrow Action$) and the morphisms from ($Event$, $b : Event^* \rightarrow Action$) to ($Event'$, $b' : Event^{*'} \rightarrow Action'$) as pairs ($h$, j), where $h : Event \rightarrow Event'$ and $j : Action \rightarrow Action'$ such that the following diagram commutes.

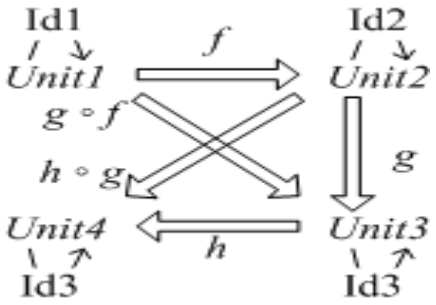


Proof. *The composition law of automata is defined to be that of functions between sets. For example, composition applies internally to each component of the morphsims* (X, $b : X^* \rightarrow Y$). *Similarly, the identity of an automation is defined to be the pairs that consists of the two identity functions over X and function* $b : X^* \rightarrow Y$. *The proof of* ***RAE-Behavior*** *is indeed obtained because the associativity of the composition law and*

the property of the identity are automatically inherited from the corresponding properties of functions.

Property 6.2.3: **Discrete-Time** is a category in which objects are abstracting time unit represented as integers, and morphisms are of type "before" denoted as "<".

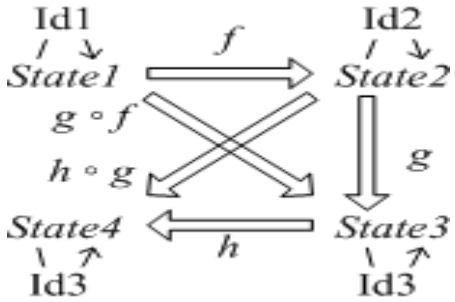
Proof. *All what we need is to prove: i) the existence of composition and identity morphism, and ii) prove associativity. Let $Unit_1$, $Unit_2$ and $Unit_3$ be three time units such that $Unit_1$ is before $Unit_2$, which is before $Unit_3$. Then $Unit_1$ is before $Unit_3$ (indirectly through $Unit_2$), which means the existence of a composition of morphisms between $Unit_1$ and $Unit_3$. The identity morphism does exist as a natural representation of interactions with atomic time unit. Let f, g and h be the morphisms such that f: $Unit_1 \rightarrow Unit_2$, g: $Unit_2 \rightarrow Unit_3$ and h: $Unit_3 \rightarrow Unit_4$. It is clear that $h \circ (g \circ f) = (h \circ g) \circ f$.*■



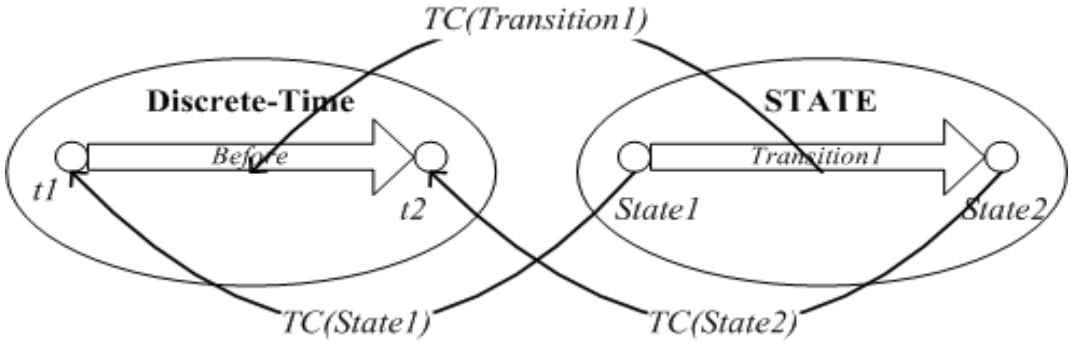
Property 6.2.4: **STATE** is a category in which objects are states denoted by $State_1$, $State_2$…, and morphisms are transitions denoted by $Transition_1$, $Transition_2$….

Proof. *All what we need is to prove: i) the existence of composition and identity morphism, and ii) prove associativity. Let $State_1$, $State_2$ and $State_3$ be three states such that $State_1$ transits to $State_2$, which transits to $State_3$. Then $State_1$ can transit to $State_3$ (indirectly through $State_2$), which means the existence of a composition of morphisms*

between State$_1$ and State$_3$. The identity morphism does exist as a natural representation of internal transitions. Let f, g and h be the morphisms such that f: State$_1$→ State$_2$, g: State$_2$ → State$_3$ and h: State$_3$ → State$_4$. It is clear that h ∘ (g ∘ f) = (h ∘ g) ∘ f.■

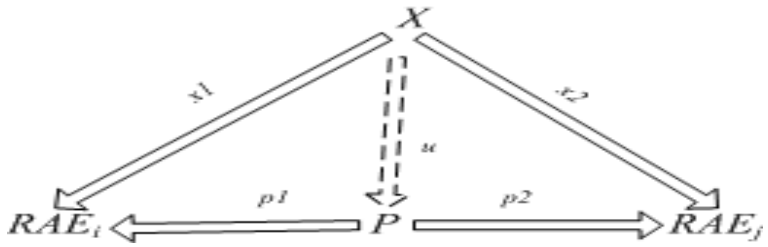


Property 6.2.5: The time constraints on the state transitions of the RAE is specified as a functor from **STATE** to **Discrete-Time** (see Property 6.2.3), a structure-preserving mapping of the objects (states) of **STATE** to the objects (time unit expressed as integers) of **Discrete-Time**, and of the morphisms of **STATE** (transition) to morphisms (before) of **Discrete-Time** (see Definition 3.1.2 and the figure below).

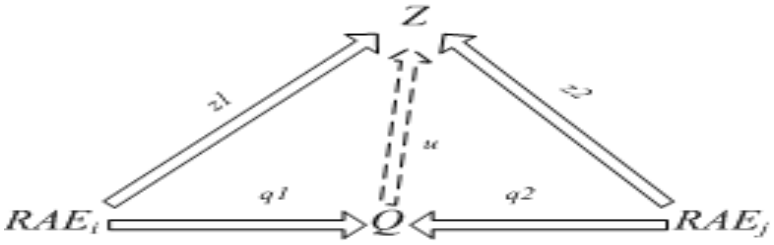


Property 6.2.6: The synchronous communication between the RAE can be specified as their product denoted by $RAE_i \times RAE_j$ consisting of an object P as well as arrows $RAE_i \xleftarrow{p_1} P \xrightarrow{p_2} RAE_j$ satisfying: any diagram of a form $RAE_i \xleftarrow{x_1} X \xrightarrow{x_2} RAE_j$, there exists a unique $u : X \rightarrow P$ making the following diagram commute (see Definition

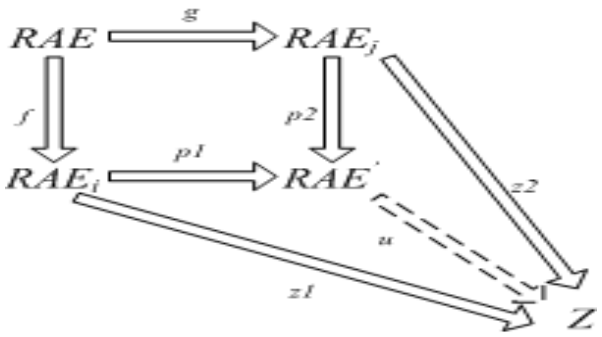
3.3.4).



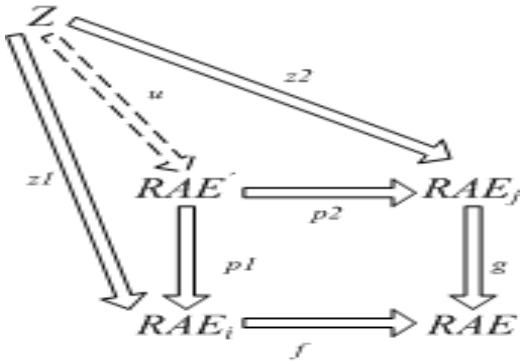
Property 6.2.7: The asynchronous communication among the RAE can be specified by their coproduct denoted by $RAE_i + RAE_j$ consisting of an object Q and arrows $RAE_i \xrightarrow{q_1} Q \xleftarrow{q_2} RAE_j$ satisfying: any diagram of the form $RAE_i \xrightarrow{z_1} Z \xleftarrow{z_2} RAE_j$, there exists a unique $u : Q \rightarrow Z$ making the following diagram commute (see Definition 3.4.2).



Property 6.2.8: The next relay of the outgoing communication from the same source object RAE to the same destination object RAE' can be specified as the pushout: $RAE' = RAE_i +_{RAE} RAE_j$, such that given any $z_1 : RAE_i \rightarrow Z$ and $z_2 : RAE_j \rightarrow Z$ with $z_1 \circ f = z_2 \circ g$, there exists a unique $u : RAE' \rightarrow Z$, $u \circ p_1 = z_1$ and $u \circ p_2 = z_2$ (see Definition 3.5.4).



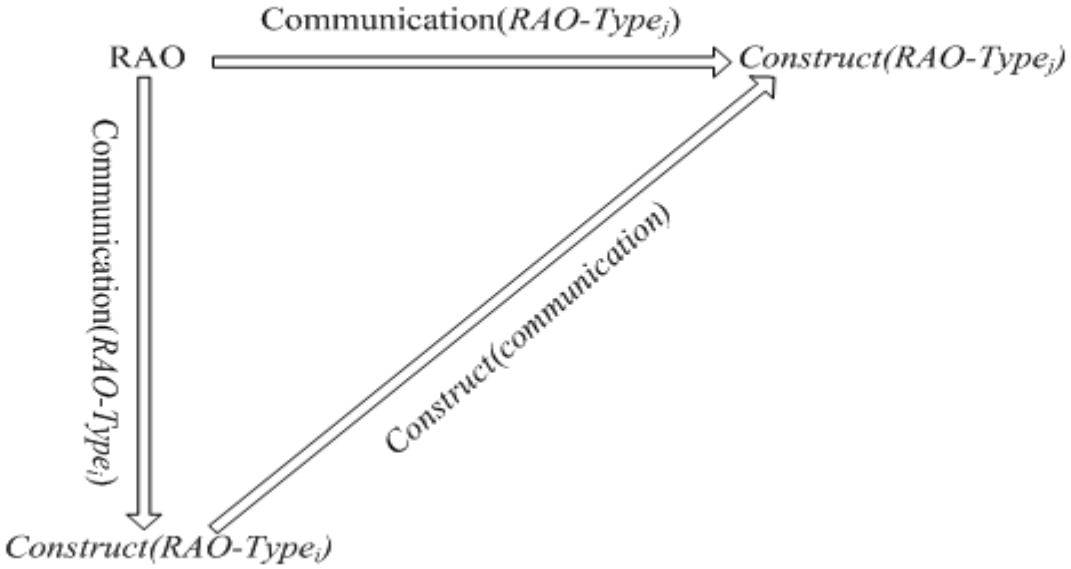
Property 6.2.9: The previous relay of the incoming communication toward the same destination *RAE* from the same source object *RAE*' can be specified as the pullback: *RAE*' = $RAE_i \times_{RAE} RAE_j$, such that given any $z_1 : Z \rightarrow RAE_i$ and $z_2 : Z \rightarrow RAE_j$ with $f \circ z_1 = g \circ z_2$, there exists a unique $u : Z \rightarrow RAE'$ with $z_1 = p_1 \circ u$ and $z_2 = p_2 \circ u$ (see Definition 3.5.3).



Property 6.2.10: If we start with a diagram of RAO, a kind of universal communicator may be introduced, and this is a higher-level object with arrow connections to each object in a base diagram. Therefore, we can model that object (limit or colimit object) and those arrow connections as a limit or colimit of the base diagram. Graphically speaking, the limit is a domain of all the arrows going to the RAO in the base diagram, and the colimit is a codomain of all the arrows coming from the RAO in the base diagram. Having the limit or colimit allows for the modeling of each specific interaction among the RAO (group behavior) by the communication path between the limit (colimit) object and those RAO. According to the definition of the limit (colimit) object, no other object in the diagram can improve the communication capability comparing to the limit (colimit) object due to the commutativity constraint in the universal properties of a limit (colimit).

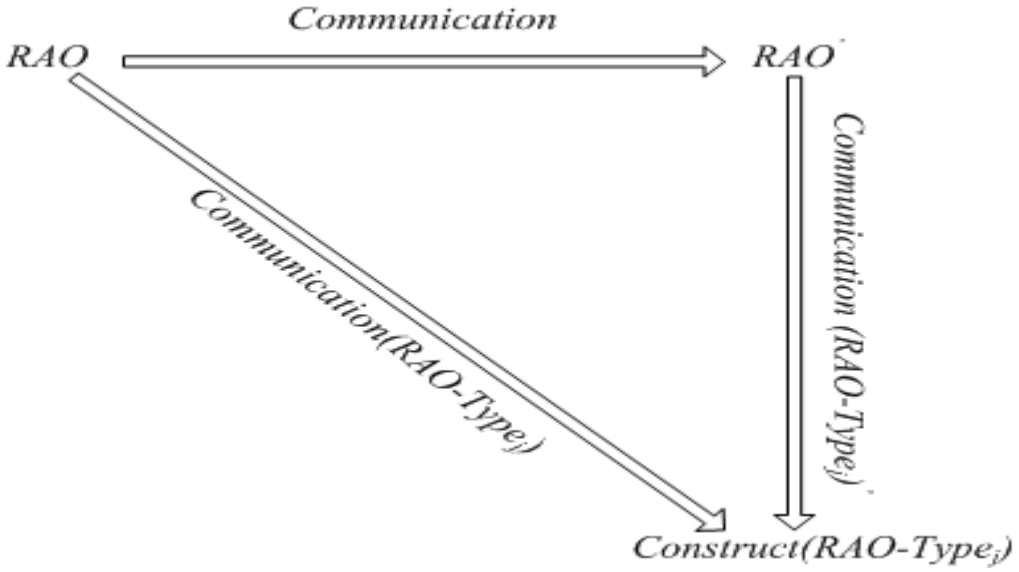
As the behavior of a RAC is aggregated from the behavior of its RAO (RAC is

aggregated from its RAO), a limit for the category (diagram) **RAC** can represent its designated group behavior. Let **RAO-Type** and **RAO-Type-Instance** be categories (see Property 6.1.1). A *diagram* of *type* **RAO-Type** in **RAO-Type-Instance** is the functor *Construct* : **RAO-Type** → **RAO-Type-Instance**. The objects in the *index category* **RAO-Type** are represented as $RAO\text{-}Type_i$, $RAO\text{-}Type_j$, … and the values of the functor *Construct* are in the form of $Construct(RAO\text{-}Type_i)$, $Construct(RAO\text{-}Type_j)$, …. A *cone* to the diagram *Construct* consists of an object *RAO* as well as a family of morphisms in **RAO-Type-Instance**, Communication($RAO\text{-}Type_j$): $RAO \rightarrow Construct(RAO\text{-}Type_j)$ for each object $RAO\text{-}Type_j \in$ **RAO-Type** such that for each morphism *Communication* : $RAO\text{-}Type_i \rightarrow RAO\text{-}Type_j$ in **RAO-Type**, the following triangle commutes.



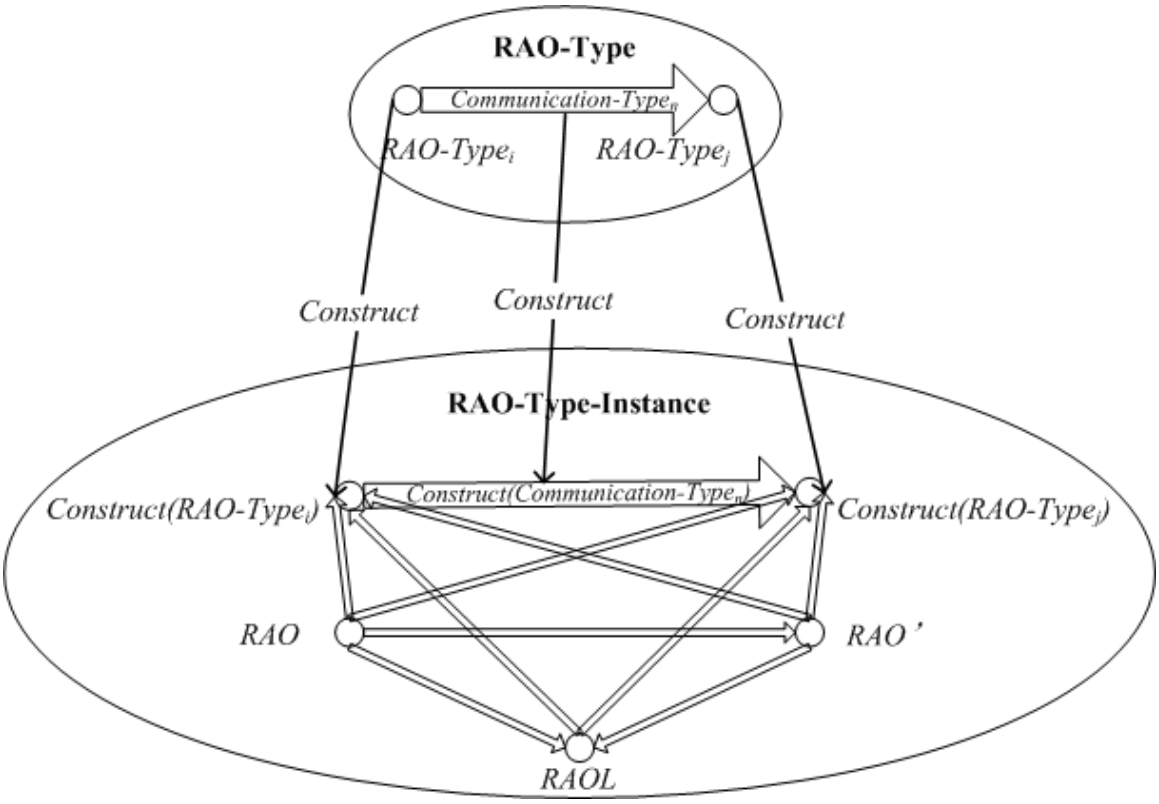
A morphism of cones *Communication* : (*RAO*, $Communication(RAO\text{-}Type_j)$) → (*RAO*', $Communication\ (RAO\text{-}Type_j)'$) is a morphism *Communication* in **RAO-Type-Instance** making each triangle commute: $Communication\ (RAO\text{-}Type_j) = Communication(RAO\text{-}Type_j)' \circ Communication$ for all $RAO\text{-}Type_j \in$ **RAO-Type**. Therefore we have a category

RAC-Behavior(*Construct*) of cones to *Construct* (see the figure below).



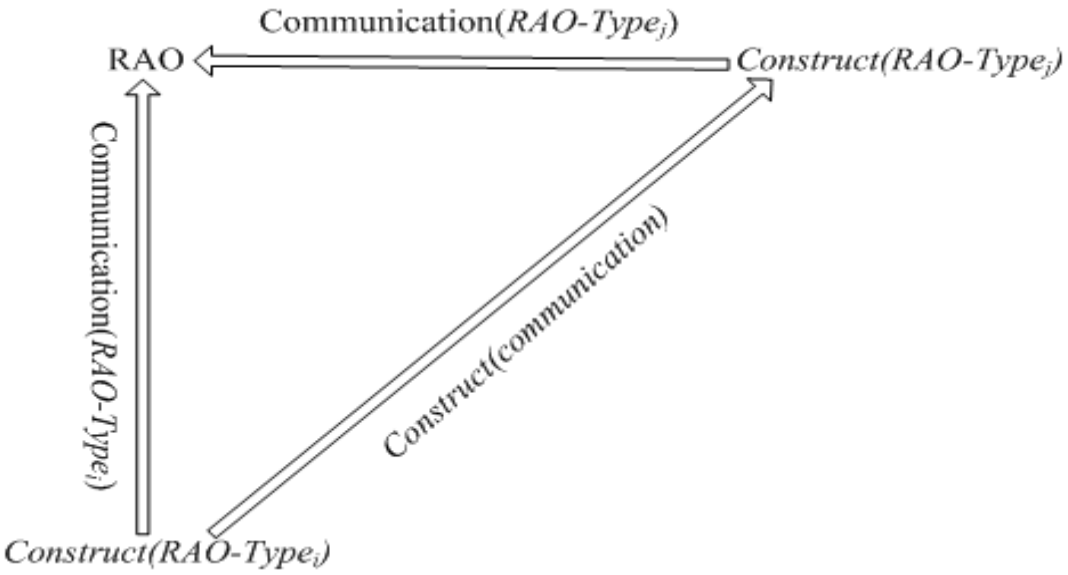
We consider the diagram *Construct* as a view of **RAO-Type** in **RAO-Type-Instance**. A cone to such a diagram *Construct* is imaged as a many-sided pyramid over the base *Construct*, and a morphism of cones is an arrow between the apexes of such pyramids.

A *limit* for the diagram *Construct* : **RAO-Type** → **RAO-Type-Instance** is a terminal object in the category **RAC-Behavior**(*Construct*) represented as *RAOL* (limit object) along with the communication from *RAOL* to *RAO*. A *finite limit* is a limit for a diagram on a finite index category **RAO-Type**. As a result, the grid-like communication among the *RAO* can be regarded as a cone-like incoming communication between those *RAO* and their *RAOL*, through converting their relationship of many-to-many to one-to-many by a categorical computation. Such model facilitates the specification of the designated behavior of those *RAO* by hiding the many-to-many relationship details (see Definition 3.5.6 and the figure below).

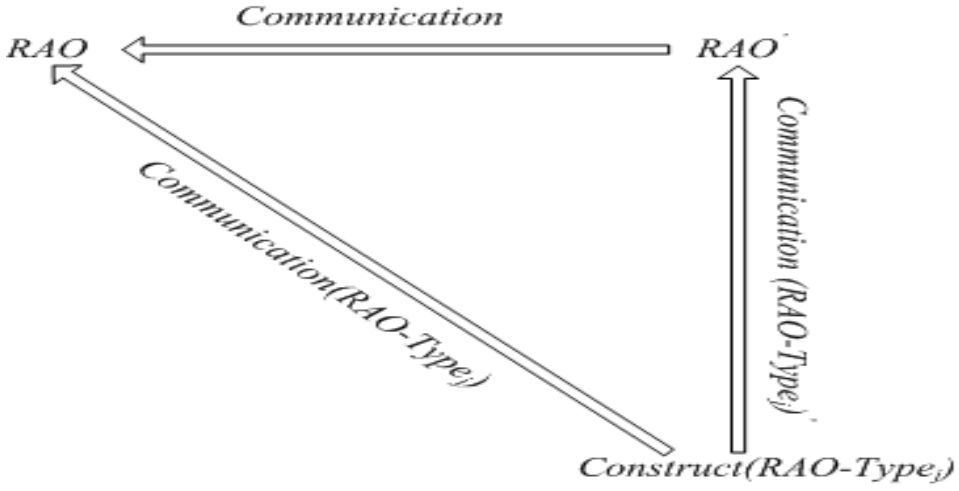


The validity of the category **RAO-Type-Instance** in the figure above is guaranteed by the functor from **RAO-Type-Instance** to **RAO-Type** (see Property 6.1.1).

Property 6.2.11: Similarly to **Property 6.2.10**, a *colimit* for the diagram *Construct*: **RAO-Type** → **RAO-Type-Instance** is an initial object in the category of *cocones* from the base *Construct* represented as *RAOL* (colimit object) along with the communication from *RAO* to *RAOL*. Each cocone consists of an object *RAO* (the vertex) and morphisms *Communication*(*RAO-Type$_j$*) : *Construct(RAO-Type$_j$)* → *RAO* for every *RAO-Type$_j$* ∈ **RAO-Type**, such that for all the *Communication* : *RAO-Type$_i$*→*RAO-Type$_j$* in **RAO-Type**, the triangle below commutes: *Communication(RAO-Type$_j$)* ∘ *Construct(Communication)* = *Communication(RAO-Type$_i$)* as the definition 3.5.9 (see the figure below).

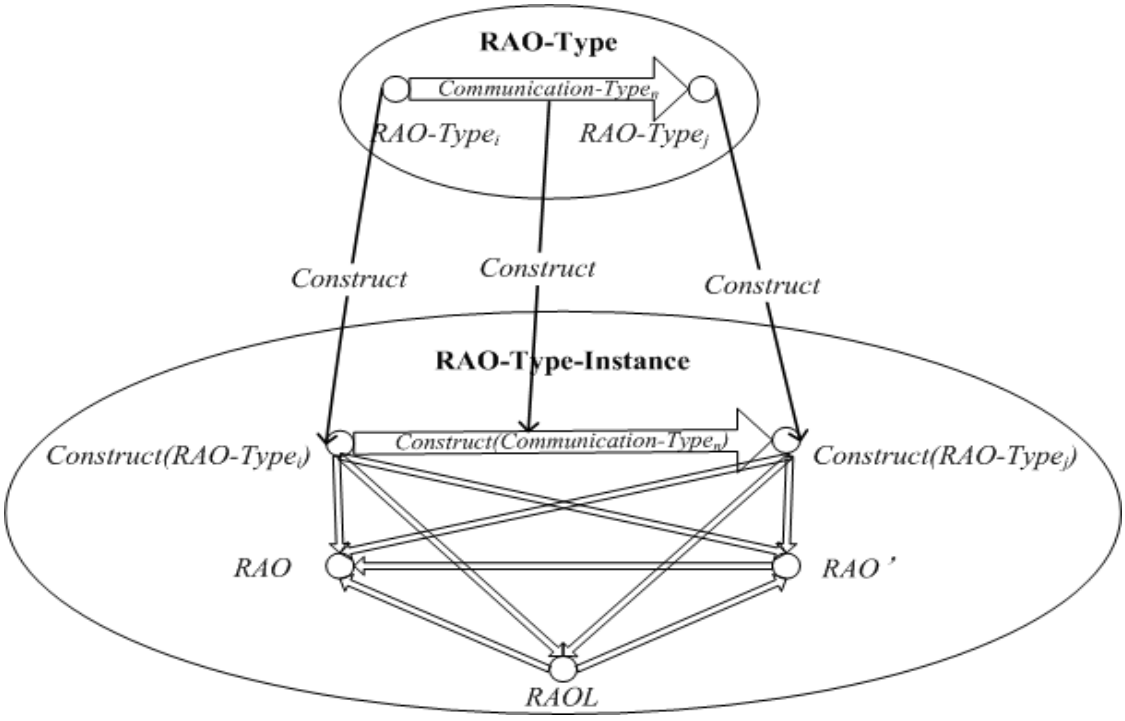


A morphism of the cocones *Communication* : (*RAO*$'$, *Communication(RAO-Type$_j$)*$'$) → (*RAO, Communication(RAO-Type$_j$)*) is a morphism *Communication*: *RAO*$'$ → *RAO* in **RAO-Type-Instance**, the triangle below commutes: *Communication* ∘ *Communication (RAO-Type$_j$)*$'$ = *Communication(RAO-Type$_j$)* for all *RAO-Type$_j$* ∈ **RAO-Type** (see the figure below).



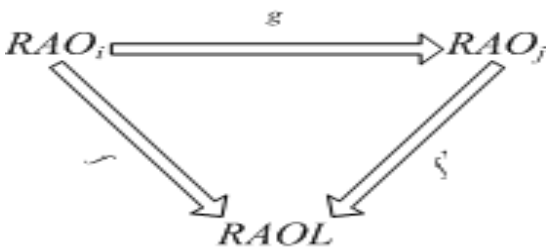
As a result, the grid-like communication among the RAO can be regarded as a cone-like outgoing communication between those RAO and their RAOL, by converting their relationship of many-to-many to many-to-one through a categorical computation.

Such model facilitates the specification of the achieved behavior of those RAO by hiding the many-to-many relationship details (see the figure below).

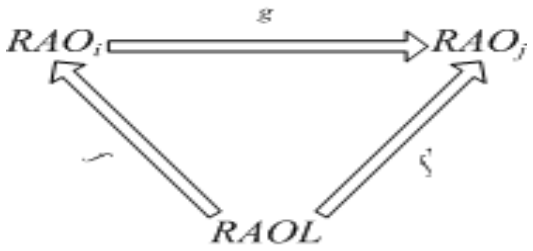


The validity of the category **RAO-Type-Instance** in the figure above is guaranteed by the functor from **RAO-Type-Instance** to **RAO-Type** (see Property 6.1.1).

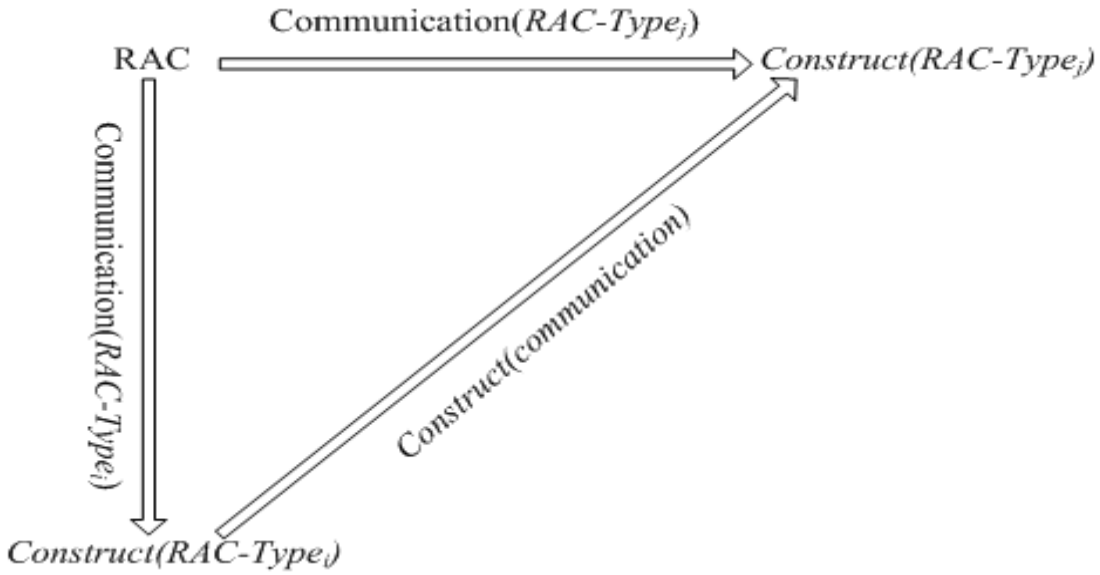
Property 6.2.12: The outgoing communication from the RAO to its RAOL in a RAC can be specified by a slice category as **RAC**/*RAOL*, where each object is the outgoing communication (f, f') and the morphism is the arrow g from f: $RAO_i \rightarrow RAOL$ to f': $RAO_j \rightarrow RAOL$ such that $f' \circ g = f$ (see Definition 3.2.4).



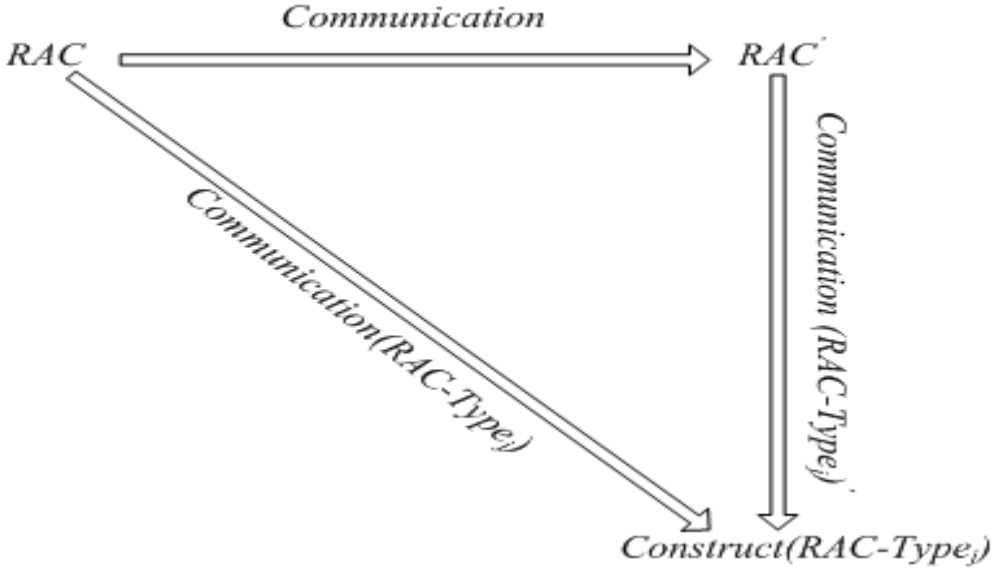
Property 6.2.13: The incoming communication from the RAOL to its RAO in a RAC can be specified by a coslice category as *RAOL*/**RAC**, where objects are incoming communication (f, f') and the morphism is an arrow g from f: $RAOL \rightarrow RAO_i$ to f': $RAOL \rightarrow RAO_j$ such that $g \circ f = f'$ (see Definition 3.2.5).



Property 6.2.14: As the behavior of a RACG is aggregated from the behavior of its RAC (RACG is aggregated from its RAC), a limit for the category (diagram) **RACG** can represent its designated group behavior. Let **RAC-Type** and **RAC-Type-Instance** be categories (see Property 6.1.1). A *diagram* of *type* **RAC-Type** in **RAC-Type-Instance** is the functor *Construct* : **RAC-Type** → **RAC-Type-Instance**. The objects in the *index category* **RAC-Type** are represented as *RAC-Type*$_i$, *RAC-Type*$_j$, … and the values of functor *Construct* are in the form of *Construct (RAC-Type*$_i$*)*, *Construct(RAC-Type*$_j$*)*, …. A *cone* to diagram *Construct* consists of an object *RAC* and a family of morphisms in **RAC-Type-Instance**, Communication (*RAC-Type*$_j$): *RAC* → *Construct(RAC-Type*$_j$*)* for each object *RAC-Type*$_j$ ∈ **RAC-Type** such that for each morphism *Communication* : *RAC-Type*$_i$ → *RAC-Type*$_j$ in **RAC-Type**, the following triangle commutes.

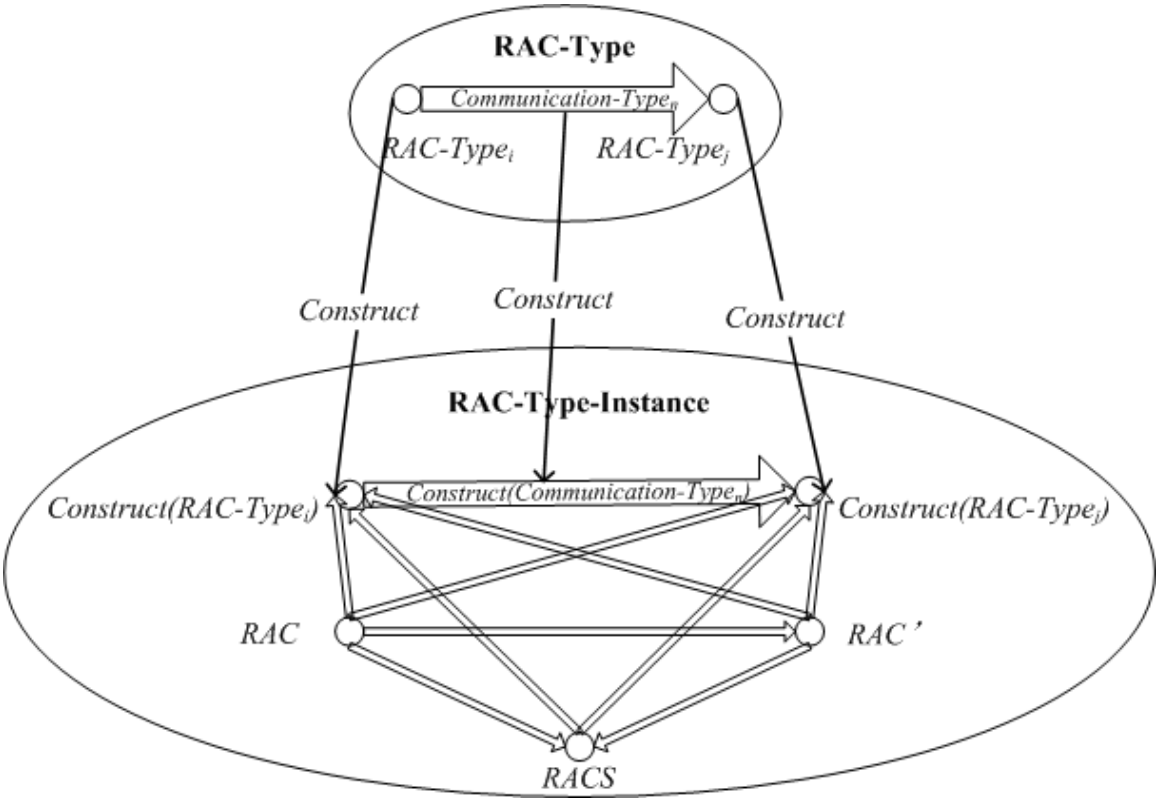


A morphism of cones *Communication* : (*RAC*, *Communication(RAC-Type$_j$)*) → (*RAC*$'$, *Communication (RAC-Type$_j$)*$'$) is a morphism *Communication* in **RAC-Type-Instance** making each triangle commute: *Communication(RAC-Type$_j$)* = *Communication(RAC-Type$_j$)*$'$ ∘ *Communication* for all *RAC-Type$_j$* ∈ **RAC-Type**. Therefore, we have a category **RACG-Behavior**(*Construct*) of cones to *Construct* (see the figure below).



A *limit* for the diagram *Construct* : **RAC-Type** → **RAC-Type-Instance** is a terminal object in the category **RACG-Behavior**(*Construct*) represented as *RACS* (limit object)

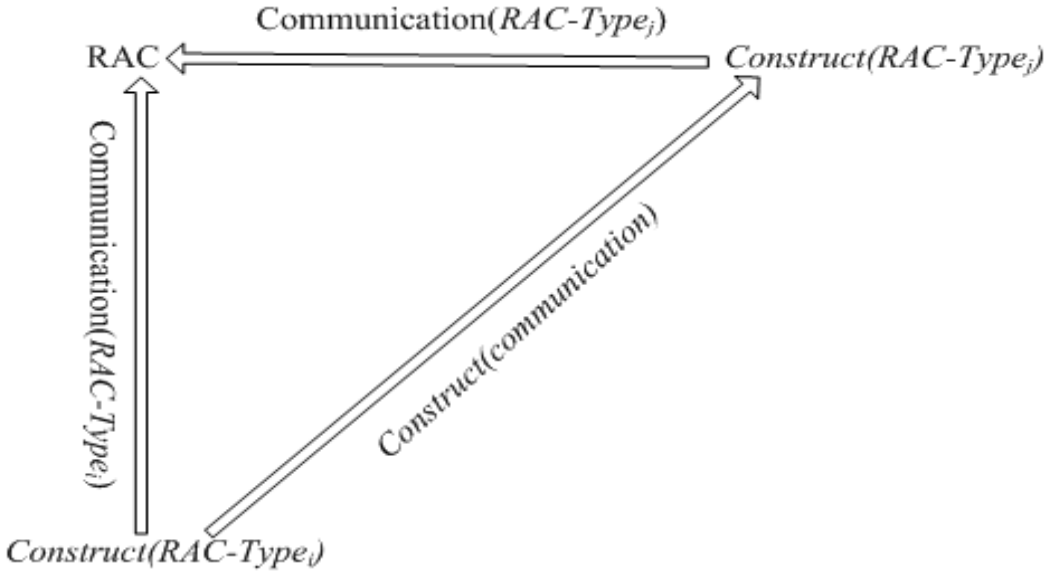
along with the communication from *RACS* to *RAC*. A *finite limit* is a limit for a diagram on a finite index category **RAC-Type**. As a result, the grid-like communication among the RAC can be regarded as a cone-like incoming communication between those RAC and their RACS, through converting their relationship of many-to-many to one-to-many by a categorical computation. Such model facilitates the specification of the designated behavior of those RAC by hiding the many-to-many relationship details (see Definition 3.5.6 and the figure below).



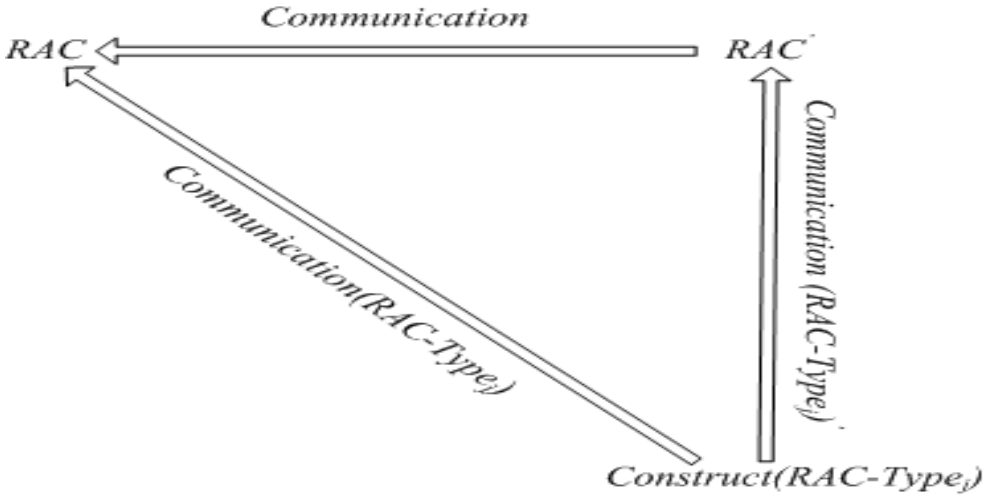
The validity of the category **RAC-Type-Instance** in the figure above is guaranteed by the functor from **RAC-Type-Instance** to **RAC-Type** (see Property 6.1.1).

Property 6.2.15: Similarly to **Property 6.2.14**, a *colimit* for the diagram *Construct*: **RAC-Type → RAC-Type-Instance** is an initial object in the category of *cocones* from

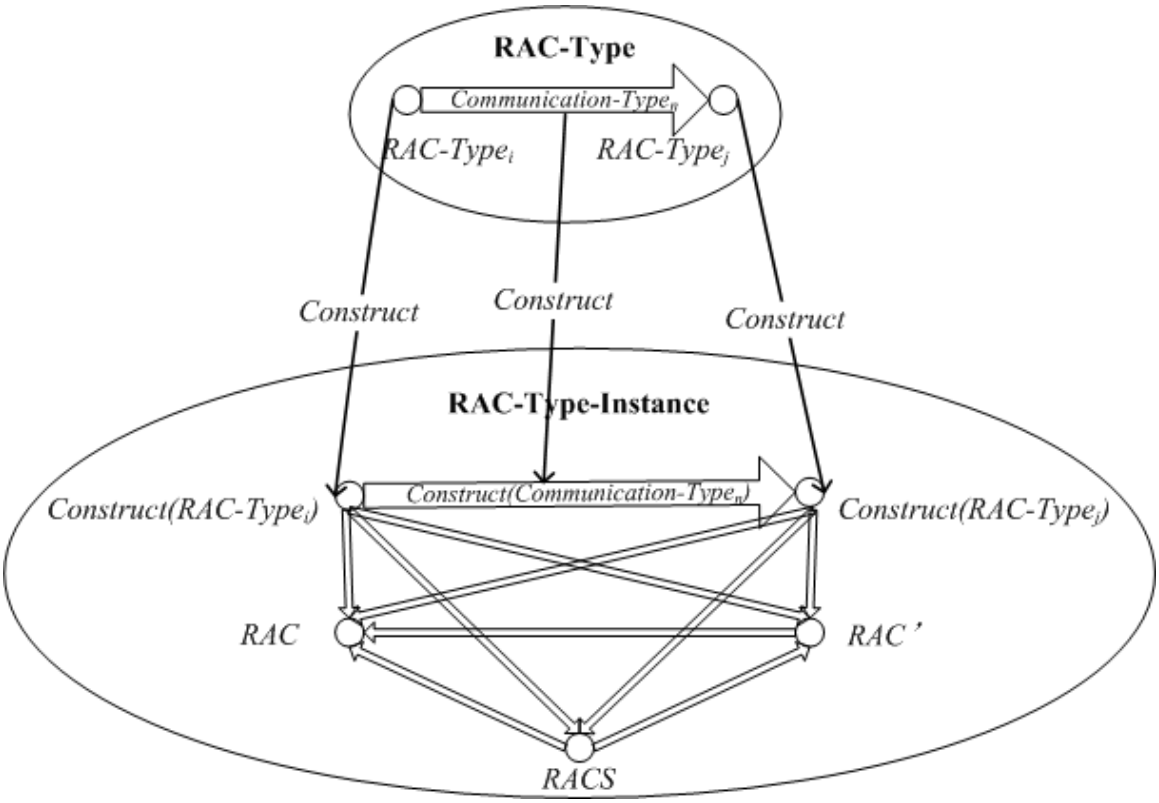
the base *Construct* represented as *RACS* (colimit object) along with the communication from *RAC* to *RACS*. Each cocone consists of an object *RAC* (the vertex) and morphisms *Communication*(*RAC-Type*$_j$) : *Construct(RAC-Type*$_j$*)* → *RAC* for every *RAC-Type*$_j \in$ **RAC-Type**, such that for all the *Communication* : *RAC-Type*$_i$→*RAC-Type*$_j$ in **RAC-Type**, the triangle below commutes: *Communication(RAC-Type*$_j$*)* ∘ *Construct(Communication)* = *Communication (RAC-Type*$_i$*)* as the definition 3.5.9 (see the figure below).



A morphism of the cocones *Communication* : (*RAC*′, *Communication(RAC-Type*$_j$*)*′) → (*RAC*, *Communication(RAC-Type*$_j$*)*) is a morphism *Communication*: *RAC′* → *RAC* in **RAC-Type-Instance**, the triangle below commutes: *Communication* ∘ *Communication (RAC-Type*$_j$*)*′ = *Communication(RAC-Type*$_j$*)* for all *RAC-Type*$_j \in$ **RAC-Type** (see the figure below).

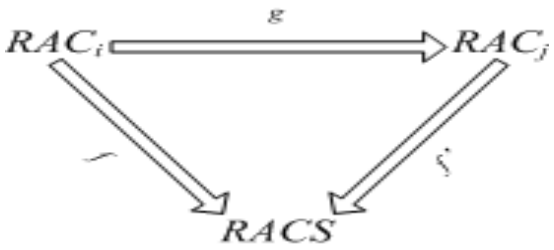


As a result, the grid-like communication among the RAC can be regarded as a cone-like outgoing communication between those RAC and their RACS, by converting their relationship of many-to-many to many-to-one through a categorical computation. Such model facilitates the specification of the achieved behavior of those RAC by hiding the many-to-many relationship details (see the figure below).

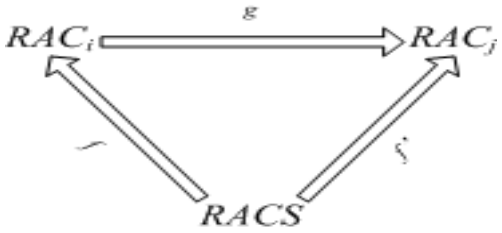


The validity of the category **RAC-Type-Instance** in the figure above is guaranteed by the functor from **RAC-Type-Instance** to **RAC-Type** (see Property 6.1.1).

Property 6.2.16: The outgoing communication from the RAC to its RACS in a RACG can be specified by a slice category as **RACG**/*RACS*, where each object is the outgoing communication (f, f') and the morphism is the arrow g from f: $RAC_i \rightarrow RACS$ to f': $RAC_j \rightarrow RACS$ such that $f' \circ g = f$ (see Definition 3.2.4).

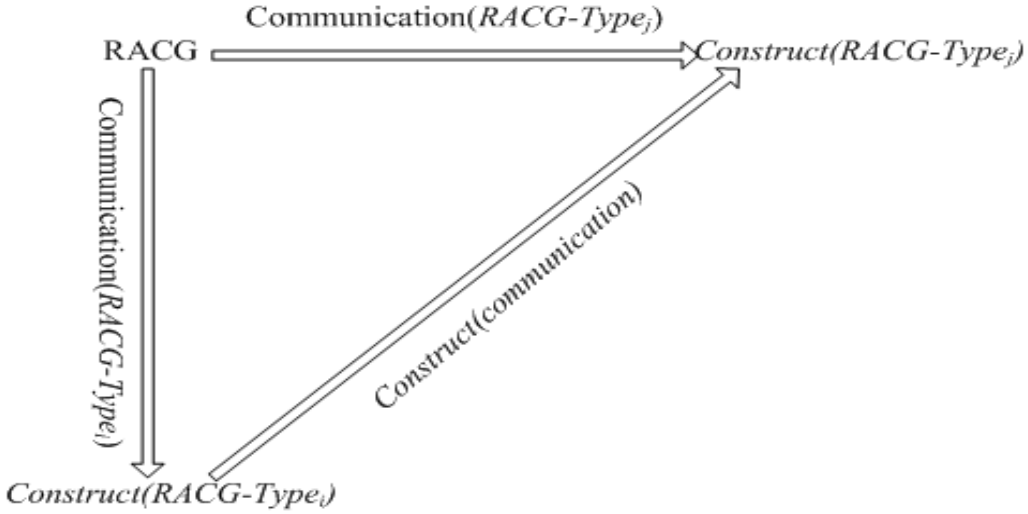


Property 6.2.17: The incoming communication from the RACS to its RAC in a RACG can be specified by a coslice category as *RACS*/**RACG**, where each object is the incoming communication (f, f') and the morphism is an arrow g from f: $RACS \rightarrow RAC_i$ to f': $RACS \rightarrow RAC_j$ such that $g \circ f = f'$ (see Definition 3.2.5).

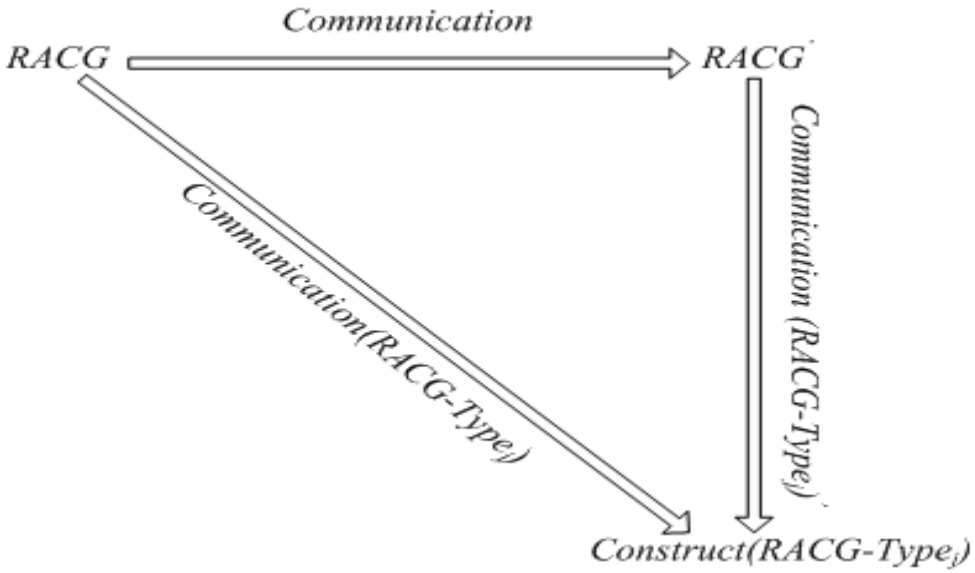


Property 6.2.18: As the behavior of a RAS is aggregated from the behavior of its RACG (RAS is aggregated from its RACG), a limit for the category (diagram) **RAS** can represent its designated group behavior. Let **RACG-Type** and **RACG-Type-Instance** be categories (see Property 6.1.1). A *diagram* of the *type* **RACG-Type** in **RACG-Type-**

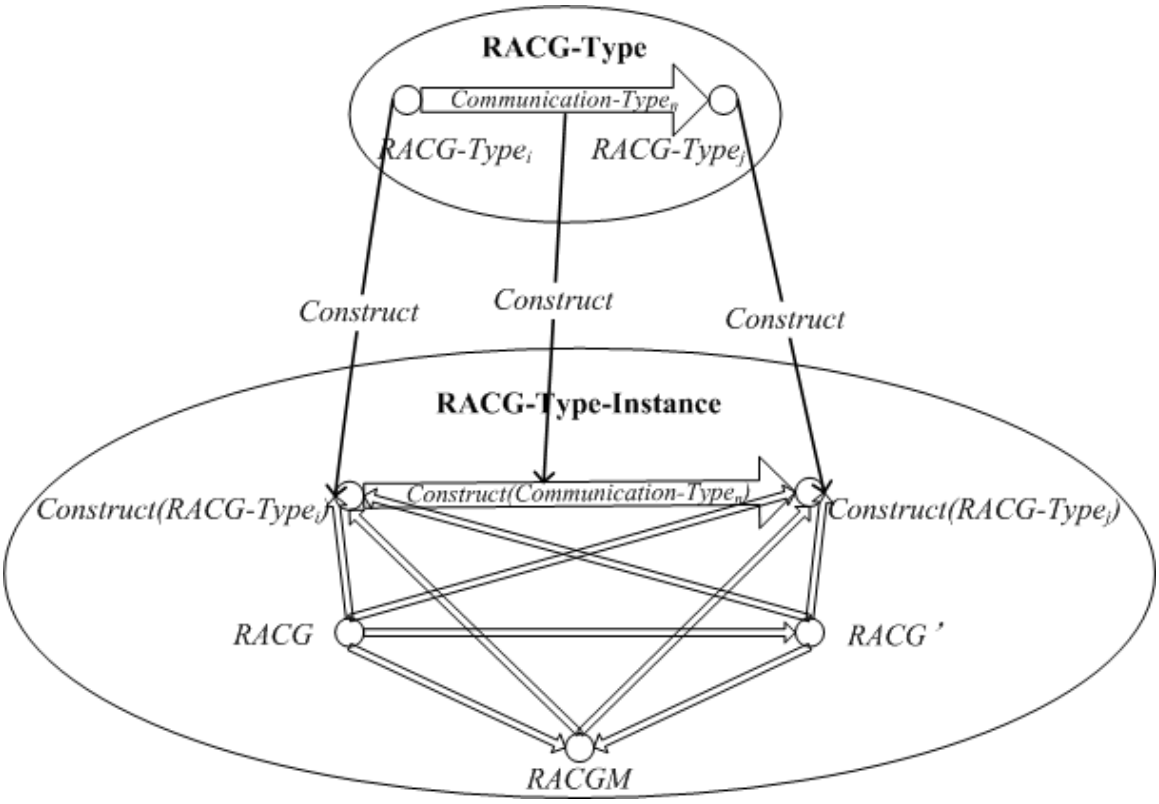
Instance is a functor *Construct* : **RACG-Type** → **RACG-Type-Instance**. The objects in the *index category* **RACG-Type** can be represented as $RACG\text{-}Type_i$, $RACG\text{-}Type_j$, … and the values of functor *Construct* are in the form of $Construct(RACG\text{-}Type_i)$, $Construct\ (RACG\text{-}Type_j)$, …. A *cone* to the diagram *Construct* consists of an object *RACG* and a family of morphisms in **RACG-Type-Instance**, Communication($RACG\text{-}Type_j$): $RACG \rightarrow Construct\ (RACG\text{-}Type_j)$ for each object $RACG\text{-}Type_j \in$ **RACG-Type** such that for each morphism *Communication* : $RACG\text{-}Type_i \rightarrow RACG\text{-}Type_j$ in **RACG-Type**, the following triangle commutes.



A morphism of cones *Communication* : (*RACG*, $Communication(RACG\text{-}Type_j)$) → ($RACG'$, $Communication(RACG\text{-}Type_j)'$) is a morphism *Communication* in **RACG-Type-Instance** making each triangle commute: $Communication(RACG\text{-}Type_j) = Communication\ (RACG\text{-}Type_j)' \circ Communication$ for all $RACG\text{-}Type_j \in$ **RACG-Type**. Therefore, we have a category **RAS-Behavior**(*Construct*) of cones to *Construct*.

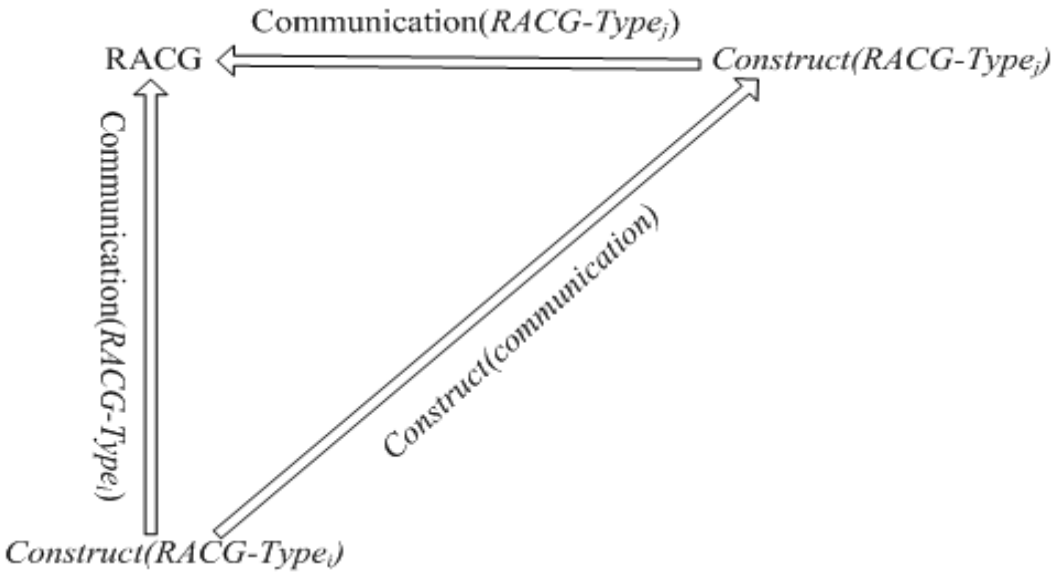


A *limit* for the diagram *Construct* : **RACG-Type** → **RACG-Type-Instance** is a terminal object in the category **RAS-Behavior**(*Construct*) represented as *RACGM* (limit object) along with the communication from *RACGM* to *RACG*. A *finite limit* is a limit for a diagram on a finite index category **RACG-Type**. Thus, the grid-like communication among the RACG can be regarded as a cone-like outgoing communication between those RACG and their RACGM, through converting their relationship of many-to-many to many-to-one by a categorical computation. Such model facilitates the specification of the designated behavior of those RACG by hiding the many-to-many relationship details (see Definition 3.5.6 and the figure below).

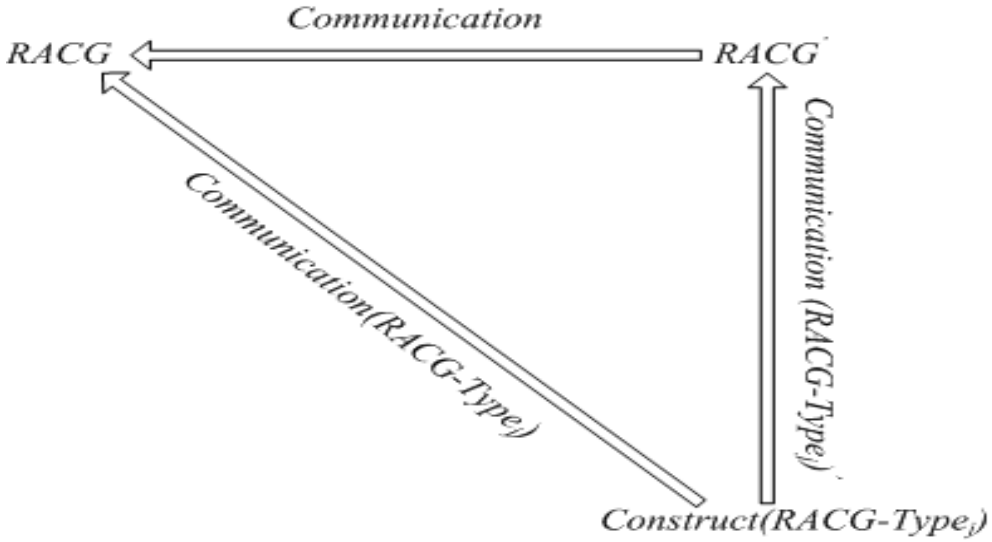


The validity of the category **RACG-Type-Instance** in the figure above is guaranteed by the functor from **RACG-Type-Instance** to **RACG-Type** (see Property 6.1.1).

Property 6.2.19: Similarly to **Property 6.2.18**, a *colimit* for the diagram *Construct*: **RACG-Type** → **RACG-Type-Instance** is an initial object in the category of *cocones* from the base *Construct* specified as *RACGM* (colimit object) with the communication from *RACG* to *RACGM*. Each cocone consists of an object *RACG* (the vertex) and morphisms *Communication*(*RACG-Type$_j$*) : *Construct(RACG-Type$_j$)* → *RACG* for every *RACG-Type$_j$* ∈ **RACG-Type**, such that for all the *Communication* : *RACG-Type$_i$*→*RACG-Type$_j$* in **RACG-Type**, the following triangle commutes: *Communication(RACG-Type$_j$)* ∘ *Construct(Communication) = Communication(RACG-Type$_i$)*.

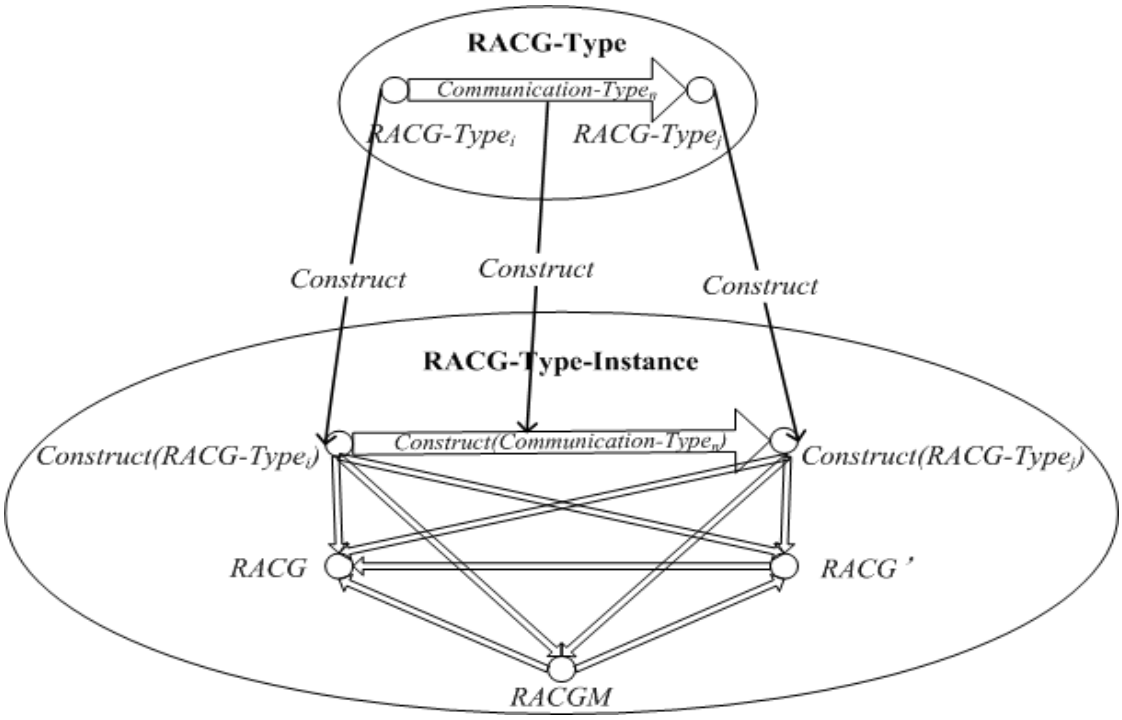


A morphism of the cocones *Communication* : (*RACG*′, *Communication(RACG-Type$_j$)*′) →(*RACG*, *Communication(RACG-Type$_j$)*) is a morphism *Communication*: *RACG*′ → *RACG* in **RACG-Type-Instance** such that the triangle below commutes: *Communication* ◦ *Communication(RACG-Type$_j$)′* = *Communication(RACG-Type$_j$)* for all *RACG-Type$_j$*∈ **RACG-Type** as the definition 3.5.9.



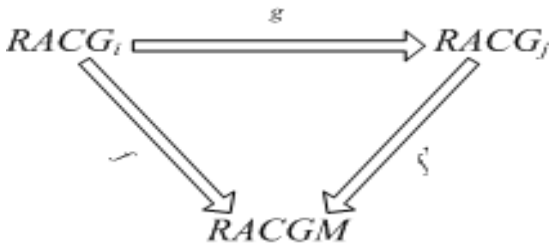
Thus, the grid-like communication among the RACG can be regarded as a cone-like outgoing communication between those RACG and their RACGM, through converting

their relationship of many-to-many to many-to-one by a categorical computation. Such model facilitates the specification of the achieved behavior of those RACG by hiding the many-to-many relationship details.

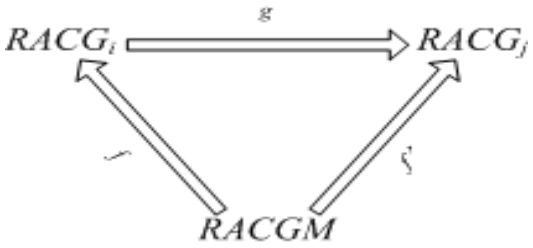


The validity of the category **RACG-Type-Instance** in the figure above is guaranteed by the functor from **RACG-Type-Instance** to **RACG-Type** (see Property 6.1.1).

Property 6.2.20: The outgoing communication from the RACG to its RACGM in a RAS can be specified by a slice category as **RAS**/*RACGM*, where each object is the outgoing communication (f, $f^{'}$) and the morphism is the arrow g from f: $RACG_i \rightarrow RACGM$ to $f^{'}$: $RACG_j \rightarrow RACGM$ such that $f^{'} \circ g = f$ (see Definition 3.2.4).

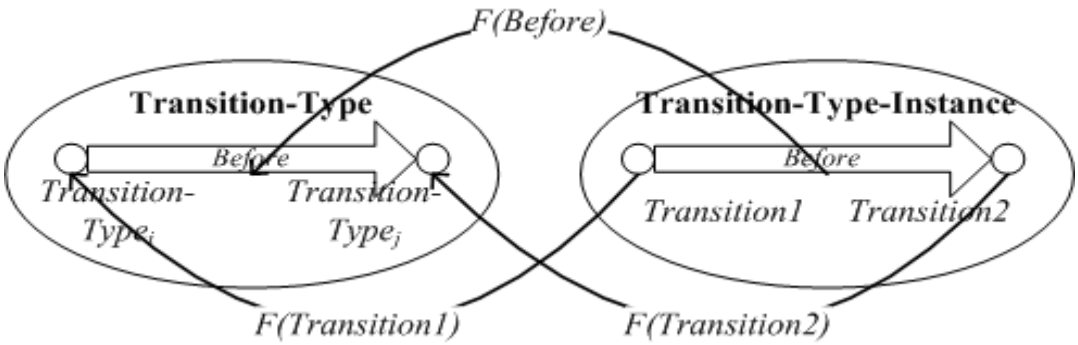


Property 6.2.21: The incoming communication from the RACGM to its RACG in a RAS can be specified by a coslice category as *RACGM*/**RAS**, where objects are incoming communications (f, f') and the morphism is an arrow g from f: $RACGM \rightarrow RACG_i$ to f': $RACGM \rightarrow RACG_j$ such that $g \circ f = f'$ (see Definition 3.2.5).



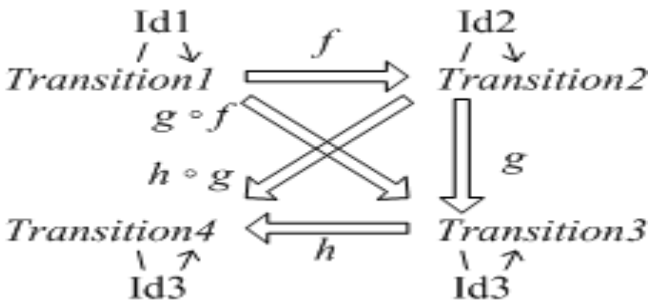
Property 6.2.22: **Transition-Type** is an instance category of the **Type** category (see Property 6.0.1) in which objects represent the transition types denoted by *ObjType (***Transition-Type***)* and morphisms represent the morphism types denoted by *MorType (***Transition-Type***)*. For example, **Transition-Type-Instance** is a category where objects are denoted by *Obj(***Transition-Type-Instance***)* and morphisms are denoted by *Mor (***Transition-Type-Instance***)*. There is a functor F from **Transition-Type-Instance** to **Transition-Type**, a structure-preserving mapping of the objects of the **Transition-Type-Instance** to the objects of **Transition-Type**: *F(ObjType(***Transition-Type-Instance***))* = *Obj(***Transition-Type***)*, and of the morphisms of the **Transition-Type-Instance** to the morphisms of **Transition-Type** denoted as *F(MorType(***Transition-Type-Instance***))* =

*Mor (***Transition-Type***)* Definition 3.1.2 (see the figure below).

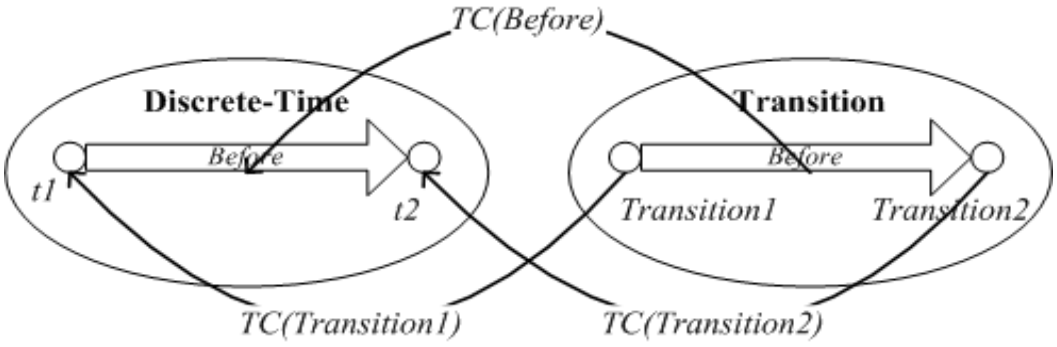


Property 6.2.23: The transitions between the states in Property 6.2.4 can be specified as a category **Transition** where objects are transitions: $Transition_1$, $Transition_2$…., and morphisms are their preorder relationship "*before*" (see Definition 3.1.1). Every transition is modeled by the triple (*state*, *event*, *state*) indicating the source state, trigger event, and destination state of the corresponding transition.

Proof. *All what we need is to prove: i) the existence of composition and identity morphism, and ii) prove associativity. Let* $Transition_1$, $Transition_2$ *and* $Transition_3$ *be three transitions such that* $Transition_1$ *occurs before* $Transition_2$, *which occurs before* $Transition_3$. *Then* $Transition_1$ *occurs before* $Transition_3$ *(indirectly through* $Transition_2$*), which means the existence of a composition of morphisms from* $Transition_1$ *to* $Transition_3$. *The identity morphism does exist as a natural representation of internal transitions. Let f, g and h be morphisms so that f:* $Transition_1 \rightarrow Transition_2$, *g:* $Transition_2 \rightarrow Transition_3$ *and h:* $Transition_3 \rightarrow Transition_4$. *It is clear that* $h \circ (g \circ f) = (h \circ g) \circ f$.■



Property 6.2.24: The time constraints on the transitions above can be specified as a functor from **Transition** to **Discrete-Time** (see Property 6.2.3), a structure-preserving mapping of the objects (transitions) of **Transition** to the objects (time unit expressed as integers) of **Discrete-Time**, and of the morphisms of the **Transition** (before) to the morphisms (before) of **Discrete-Time** (see Definition 3.1.2 and the figure below).

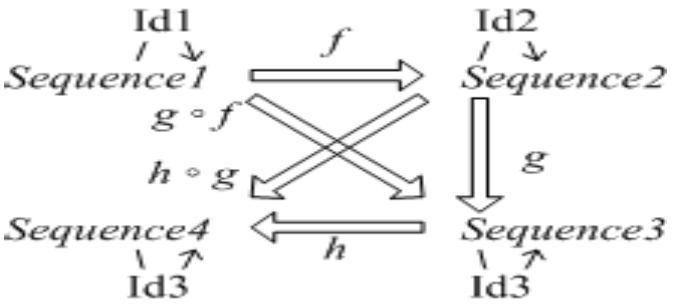


Property 6.2.25: Two transition sequences are considered equivalent (or isomorphic) denoted as $TransSeq_1 \sim TransSeq_2$ iff their first transitions $Trans_1$ and $Trans_2$ have the same source state; their last transitions have the same destination state; and the event of $Trans_1$ is exactly the event of $Trans_2$.

For instance, $TransSeq_1$ = <(*Monitor*, HasChange, *Analyze*)> is equivalent to the composite transition $TransSeq_2$ = < (*Monitor*, HasChange, *Analyze*), (*Analyze*, Analyze-Exception, *HandleException*), *(HandleException*, Handled-Analyze, *Analyze*)>.

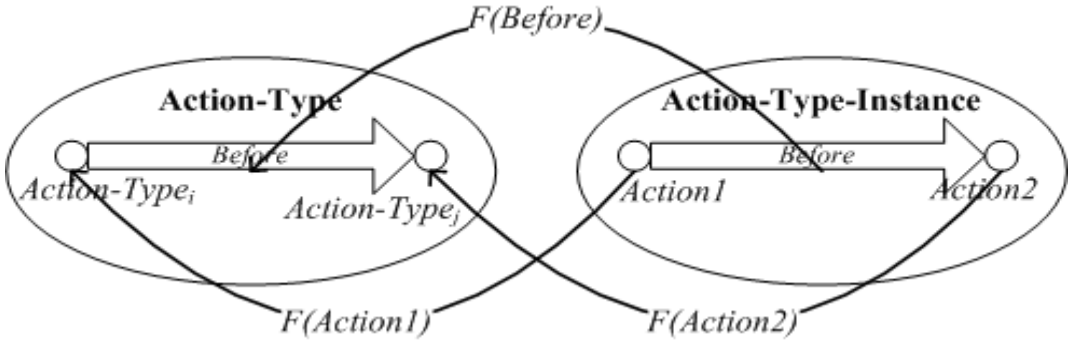
Property 6.2.26: The sequences of the transitions in Property 6.2.2 can be specified as a category **TRANSITION** where objects are sequences of transitions denoted by $Sequence_1$, $Sequence_2$..., and morphsims are the *equivalence* relationship between those sequences (see Definition 3.1.1).

Proof. *All what we need is to prove: i) the existence of composition and identity morphism, and ii) prove associativity. Let* $Sequence_1$, $Sequence_2$ *and* $Sequence_3$ *be three sequences so that* $Sequence_1$ *is equivalent to* $Sequence_2$, *which is equivalent to* $Sequence_3$. *Then* $Sequence_1$ *is equivalent to* $Sequence_3$ *(indirectly through* $Sequence_2$*), which means the existence of a composition of morphisms from* $Sequence_1$ *to* $Sequence_3$. *The identity morphism does exist as a natural representation of internal equivalence. Let f, g and h be morphisms such that f:* $Sequence_1 \rightarrow Sequence_2$, *g:* $Sequence_2 \rightarrow Sequence_3$ *and h:* $Sequence_3 \rightarrow Sequence_4$. *It is clear that* $h \circ (g \circ f) = (h \circ g) \circ f$.■



Property 6.2.27: **Action-Type** is an instance category of the **Type** category (see Property 6.0.1) where objects represent the action types denoted by *ObjType(***Action-Type***)* and morphisms represent the morphism types denoted by *MorType (***Action-Type***)*. For example, **Action-Type-Instance** is a category in which objects are denoted by *Obj (***Action-Type-Instance***)* and morphisms are denoted by *Mor(***Action-Type-Instance***)*.

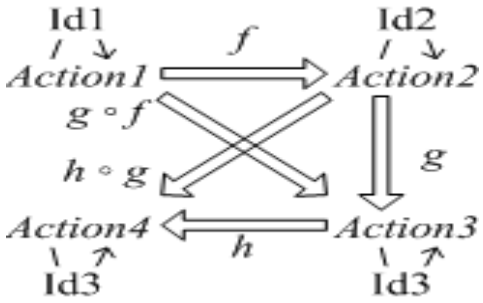
There is a functor *F* from **Action-Type-Instance** to **Action-Type**, a structure-preserving mapping of the objects of **Action-Type-Instance** to objects of **Action-Type**: *F(ObjType (***Action-Type-Instance***)) = Obj(***Action-Type***)* and of the morphisms of **Action-Type-Instance** to the morphisms of **Action-Type**: *F(MorType (***Action-Type-Instance***)) = Mor (***Action-Type***)* as Definition 3.1.2 (see the figure below).



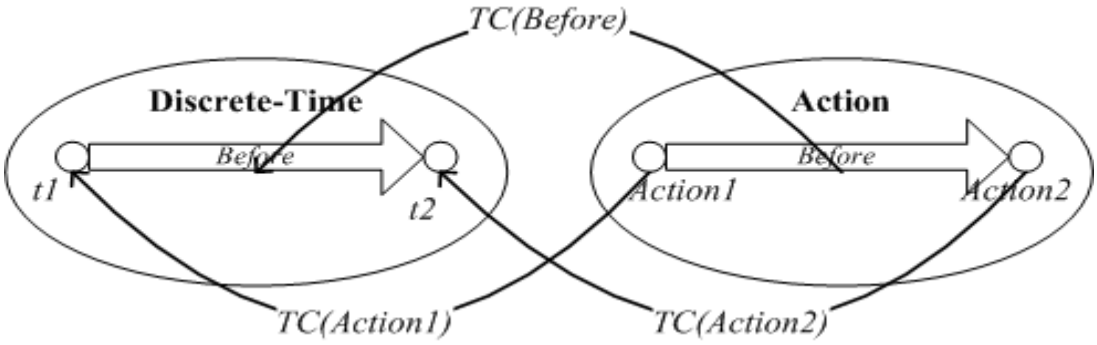
Property 6.2.28: The actions in Property 6.2.2 is a category **Action** where objects are actions: *Action$_1$*, *Action$_2$*…., and morphisms are their preorder relationship "*before*". (see Definition 3.1.1) Every action is specified as the quadruple (*sender*, *trigger-event*, *last-event*, *receiver*) stating the sender of *trigger-event*, the *trigger-event* triggering an action, the *last-event* outputted from the action, and the receiver of *trigger-event*.

Proof. *All what we need is to prove: i) the existence of composition and identity morphism, and ii) prove associativity. Let Action$_1$, Action$_2$ and Action$_3$ be three actions such that Action$_1$ occurs before Action$_2$, which occurs before Action$_3$. Then Action$_1$ occurs before Action$_3$ (indirectly through Action$_2$), which means the existence of a composition of the morphisms from Action$_1$ to Action$_3$. The identity morphism does exist as a natural representation of internal actions. Let f, g and h be morphisms so that f: Action$_1$ →*

Action₂, g: Action₂ → Action₃ and h: Action₃ → Action₄. It is clear that h ∘ (g ∘ f) = (h ∘ g) ∘ f.■



Property 6.2.29: The time constraints on the actions above is specified as a functor from **Action** to **Discrete-Time** (see Property 6.2.3), a structure-preserving mapping of the objects (actions) of **Action** to objects (time unit expressed as integers) of **Discrete-Time**, and of the morphisms of **Action** (before) to the morphisms (before) of **Discrete-Time** (see Definition 3.1.2 and the figure below).



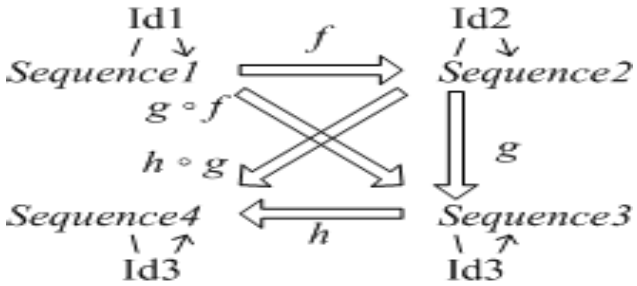
Property 6.2.30: Two sequences of actions are equivalent (or isomorphic) denoted as $ActSeq_1 \sim ActSeq_2$ iff the first actions in both sequences have the same sender and *trigger- event*, the last actions in both sequences have the same receiver and *last-event*.

For example, $ActSeq_1$ = <(*RACS*, *StartRAC*, *HeartbeatRAC*, *RACS*)> is equivalent to $ActSeq_2$ = < (*RACS*, *StartRAC*, *NoHeartbeatRAC*, *RACS*), (*RACS*, *RestartRAC*, *Heartbeat*

-*RAC*, *RACS*)>.

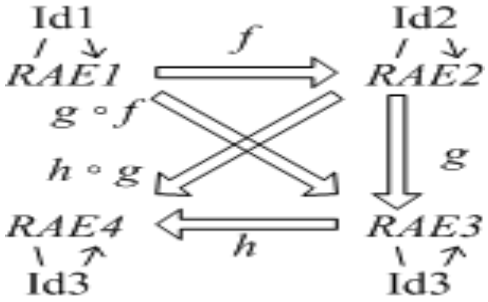
Property 6.2.31: The sequence of actions is a category **INTERACTION** in which objects are sequences consisting of actions that capture the interchanged external events as well as communication parties, and morphisms are *equivalence* relationship between those sequences (see Definition 3.1.1).

Proof. *All what we need is to prove: i) the existence of composition and identity morphism, and ii) prove associativity. Let* $Sequence_1$, $Sequence_2$ *and* $Sequence_3$ *be three sequences so that* $Sequence_1$ *is equivalent to* $Sequence_2$, *which is equivalent to* $Sequence_3$. *Then* $Sequence_1$ *is equivalent to* $Sequence_3$ *(indirectly through* $Sequence_2$*), which means the existence of a composition of morphisms from* $Sequence_1$ *to* $Sequence_3$. *The identity morphism does exist as a natural representation of internal equivalence. Let f, g and h be morphisms such that f:* $Sequence_1 \rightarrow Sequence_2$, *g:* $Sequence_2 \rightarrow Sequence_3$ *and h:* $Sequence_3 \rightarrow Sequence_4$. *It is clear that* $h \circ (g \circ f) = (h \circ g) \circ f$.■



Property 6.2.32: The social life of any *RAE* in the category **RAS** is a subcategory of **RAS** denoted as **SOCIAL**(*RAE*), where the objects are *RAE* and all other $RAE' \in$ |**RAS**| that have the morphisms with *RAE*, and the morphisms are social connections between *RAE* and RAE' as **RAS**(*RAE*, RAE') or **RAS**(RAE', *RAE*) as the Definition 3.1.1.

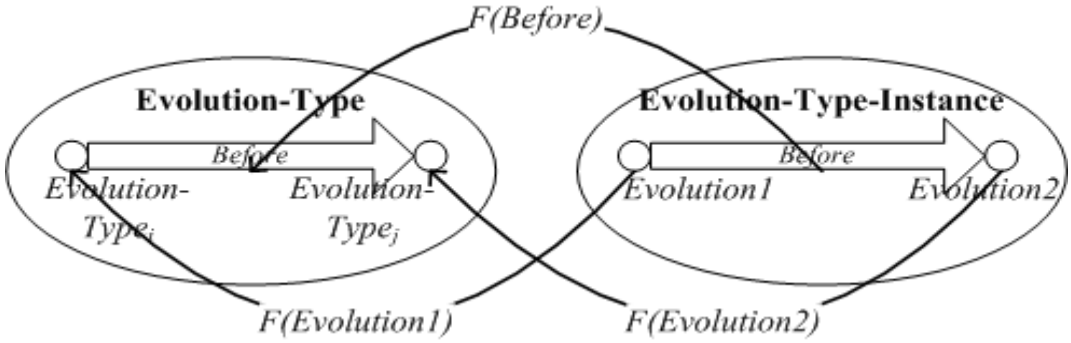
Proof. *All what we need is to prove: i) the existence of composition and identity morphism, and ii) prove associativity. Let RAE_1, RAE_2 and RAE_3 be three RAE such that RAE_1 connects to RAE_2, which connects to RAE_3. Then RAE_1 connects to RAE_3 (indirectly through RAE_2), which means the existence of a composition of morphisms between RAE_1 and RAE_3. The identity morphism does exist as a natural representation of internal connections. Let f, g and h be the morphisms such that f: $RAE_1 \rightarrow RAE_2$, g: $RAE_2 \rightarrow RAE_3$ and h: $RAE_3 \rightarrow RAE_4$. It is clear that $h \circ (g \circ f) = (h \circ g) \circ f$.*■



Property 6.2.33 The social life of *RAE* is equivalent to the social life of *RAE'* iff **SOCIAL**(*RAE*) ~ **SOCIAL**(*RAE'*) as the Property 3.6.8.

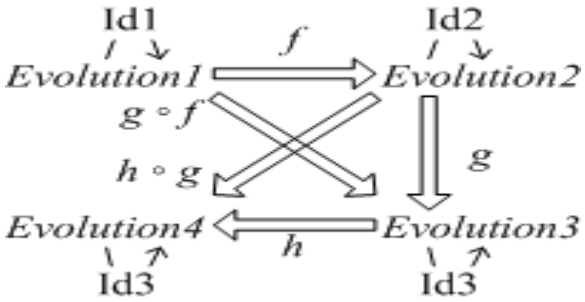
Property 6.2.34: **Evolution-Type** is an instance category of the **Type** category (see Property 6.0.1) in which objects represent the evolution types denoted by *ObjType (***Evolution-Type***)* and morphisms represent the morphism types denoted by *MorType (***Evolution-Type***)*. For example, **Evolution-Type-Instance** is the category where objects are denoted by *Obj(***Evolution-Type-Instance***)* as well as morphisms denoted by *Mor (***Evolution-Type-Instance***)*. There is a functor *F* from the **Evolution-Type-Instance** to the **Evolution-Type**, a structure-preserving mapping of the objects of **Evolution-Type-Instance** to the objects of **Evolution-Type**: *F(ObjType(***Evolution-Type-Instance***))* =

*Obj(***Evolution-Type***)*, and of the morphisms of the **Evolution-Type-Instance** to the morphisms of the **Evolution-Type** as the *F(MorType(***Evolution-Type-Instance***)) = Mor (***Evolution-Type***)* as the Definition 3.1.2 and the figure below.

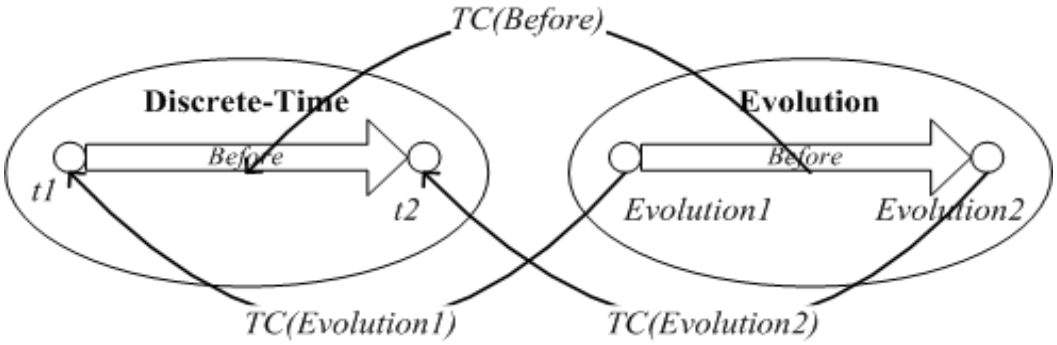


Property 6.2.35: The evolutions of the RAE in RASF is a category **Evolution** where the objects are evolutions: $Evolution_1$, $Evolution_2$…., and morphisms are their preorder relationship "*before*" (see Definition 3.1.1).

Proof. *All what we need is to prove: i) the existence of composition and identity morphism, and ii) prove associativity. Let* $Evolution_1$, $Evolution_2$ *and* $Evolution_3$ *be three evolutions such that* $Evolution_1$ *occurs before* $Evolution_2$, *which occurs before* $Evolution_3$. *Then* $Evolution_1$ *occurs before* $Evolution_3$ *(indirectly through* $Evolution_2$*), which means the existence of a composition of the morphisms from the* $Evolution_1$ *to* $Evolution_3$. *The identity morphism does exist as a natural representation of internal evolutions. Let f, g and h be morphisms so that f:* $Evolution_1 \rightarrow Evolution_2$, *g:* $Evolution_2 \rightarrow Evolution_3$ *and h:* $Evolution_3 \rightarrow Evolution_4$. *It is clear that* $h \circ (g \circ f) = (h \circ g) \circ f$.■



Property 6.2.36: The time constraints on the evolutions above can be specified as a functor from **Evolution** to **Discrete-Time** (see Property 6.2.3), a structure-preserving mapping of the objects (evolutions) of **Evolution** to the objects (time unit expressed as integers) of **Discrete-Time**, and of the morphisms of the **Evolution** (before) to the morphisms (before) of the **Discrete-Time** (see Definition 3.1.2 and the figure below).



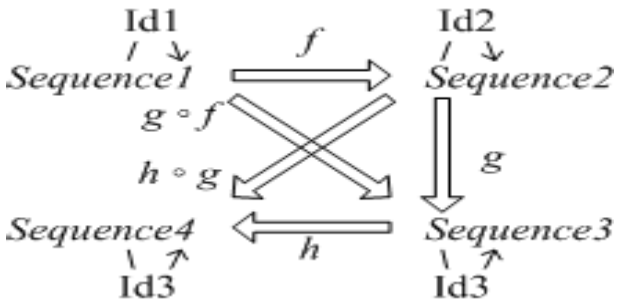
Property 6.2.37: Two evolutions (*Evolution1*: *RAE* -> *RAE*′; *Evolution1*: *RAE* -> *RAE*″) are equivalent or isomorphic denoted as $Evolution_1 \sim Evolution_2$ iff the starting RAE of those two evolutions are the same (*RAE* = *RAE*) and the social lives of the ending RAE are equivalent/isomorphic (**SOCIAL**(*RAE*) ~ **SOCIAL**(*RAE*′)) as Property 6.2.33.

Property 6.2.38: Two sequences of evolutions are equivalent (or isomorphic) denoted as $EvoSeq_1 \sim EvoSeq_2$ iff both the first and the last evolutions in those two sequences are equivalent (or isomorphic) respectively. For example, $EvoSeq_1$ = <*Restart*, *Substitute*,

Take-over> is equivalent to $EvoSeq_2 = <$ *Restart*, *Take-over*>.

Property 6.2.39: The sequence of the RAE evolutions is a category **EVOLUTION** where objects are sequences of the RAE evolutions, and morphisms are the *equivalence* relationship between those sequences (see Definition 3.1.1).

Proof. *All what we need is to prove: i) the existence of composition and identity morphism, and ii) prove associativity. Let* $Sequence_1$, $Sequence_2$ *and* $Sequence_3$ *be three sequences so that* $Sequence_1$ *is equivalent to* $Sequence_2$, *which is equivalent to* $Sequence_3$. *Then* $Sequence_1$ *is equivalent to* $Sequence_3$ *(indirectly through* $Sequence_2$*), which means the existence of a composition of morphisms from* $Sequence_1$ *to* $Sequence_3$. *The identity morphism does exist as a natural representation of internal equivalence. Let f, g and h be morphisms such that f:* $Sequence_1 \rightarrow Sequence_2$, *g:* $Sequence_2 \rightarrow Sequence_3$ *and h:* $Sequence_3 \rightarrow Sequence_4$. *It is clear that* $h \circ (g \circ f) = (h \circ g) \circ f$.■



Based on the categorical constructors and behavior we presented above, Figures 38, 39 and 40 below illustrate the templates for the categorical specification of RAC, RACG as well as RAS respectively, which can be expressed in a XML format later.

CAT-RAC <name>
Objects: <collection of objects specifying a set of RAO in RAC>
Morphisms: <collection of morphisms specifying the interactions between RAO>

Limit Object: < a limit specifying designated behavior model of those RAO>
Colimit Object: < a colimit specifying actual behavior model of those RAO>
Product Objects: <collection of product objects specifying synchronous communication between RAO>
Coproduct Objects: <collection of coproduct objects for asynchronous communication between RAO>
Pushout Objects: <collection of pushout objects for next relays of outgoing communication from RAO>
Pullback Objects: <collection of pullback objects for previous relays of incoming communication to RAO>
Slice Category: <a category specifying outgoing communication and their relations from RAO to RAOL>
Co*slice Category*: <a category specifying incoming communication and their relations from RAOL to RAO>
Functors: <collection of functors specifying the evolutions of a RAC>
Natural Transformations: <collection of natural transformations for the relations of those evolutions in RAC>
Functor Category: <a category specifying all possible evolutions and their relations in RAC>
End CAT-RAC

Figure 38: Template for Categorical Specification of RAC

CAT-RACG <name>
Objects: <collection of objects specifying a set of RAC in RACG>
Morphisms: <collection of morphisms specifying the interactions between RAC>
Limit: <a limit specifying designated behavior model of those RAC>
Co*limit*: <a colimit specifying actual behavior model of those RAC>
Product Objects: <collection of product objects specifying synchronous communication between RAC>
Coproduct Objects: <collection of coproduct objects for asynchronous communication between RAC>
Pushout Objects: <collection of pushout objects specifying next communication relays between RAC>
Pullback Objects: <collection of pullback objects specifying previous communication relays between RAC>
Slice Category: <a category specifying outgoing communication and their relations between RAC>
Co*slice Category*: <a category specifying incoming communication and their relations between RAC>
Subcategories: <collection of subcategories specifying a set of RAO in RAC>
Product Categories: <collection of product categories specifying interactions between RAC>
Functors: <collection of functors specifying the evolutions of a RACG>
Natural Transformations: <collection of natural transformations specifying the relations of those evolutions>
Functor Category: <a category specifying all possible evolutions and their relations in the RACG>
End CAT-RACG

Figure 39: Template for Categorical Specification of RACG

CAT-RAS <name>
Objects: <collection of limit or colimit objects specifying a set of RACS in RAS>
Morphisms: <collection of morphisms specifying the interactions between RACS>
Limit Object: <limit object of RACS specifying RACGM based on interactions from RACGM to RACS>
Co*limit Object*: <colimit object of RACS specifying RACGM based on interactions from RACS to RACGM>
Product Objects: <collection of product objects specifying synchronous communication between RACS>
Coproduct Objects: <collection of coproduct objects for asynchronous communication between RACS>
Pushout Objects: <collection of pushout objects for next relays of outgoing communication from RACS>

Pullback Objects: <collection of pullback objects for previous relays of incoming communication to RACS>
Slice Category: <a category specifying outgoing communication and their relations from RACS to RACGM>
Co*slice Category*: <a category for incoming communication and their relations from RACGM to RACS>
Subcategories: <collection of subcategories specifying a set of RACG in RAS>
Product Categories: <collection of product categories specifying interactions between RACG in RAS>
Functors: <collection of functors specifying the evolutions of a RAS>
Natural Transformations: <collection of natural transformations for the relations of evolutions in RAS>
Functor Category: <a category specifying all possible evolutions and their relations in RAS>
End CAT-RAS

Figure 40: Template for Categorical Specification of RAS

6.3 Representation of Categorical Models in RASF

This section states the research activity 18) in Figure 3. After having the categorical models in RASF, we need to express them using XML in terms of feeding them to our graphical illustration tool.

6.3.1 Representation for Categorical Model of Constructors

The figure below depicts an example of the representation for the categorical model of constructors (defined in Section 6.1) in a XML format, and more XML representation can be found in Appendix A.

```
<CATEGORY name = "RAE-Type">
    <OBJECT>
        <OBJECT name = "RAE-Type_i"/>
        <OBJECT name = "RAE-Type_j"/>
    </OBJECT>
    <MORPHISM>
        <MORPHISM name = "Communication-Type_n"/>
            <FROM-OBJECT name = "RAE-Type_i"/>
            <TO-OBJECT name = "RAE-Type_j"/>
        </MORPHISM>
    </MORPHISM>
</CATEGORY>
```

Figure 41: XML Specification of Index Category **RAE-Type**

6.3.2 Representation for Categorical Model of Behavior

The following figure depicts an example of the representation for the categorical model of behavior (defined in Section 6.2) in a XML format, and more XML representation can be found in Appendix B.

```
<CATEGORY name = "Function-Pair-Type">
    <OBJECT>
        <OBJECT name = "Function-Pair-Type_i"/>
        <OBJECT name = "Function-pair-Type_j"/>
    </OBJECT>
    <MORPHISM>
        <MORPHISM name = "Interaction-Type_n"/>
            <FROM-OBJECT name = "Function-Pair-Type_i"/>
            <TO-OBJECT name = "Function-Pair-Type_j"/>
        <MORPHISM>
    </MORPHISM>
</CATEGORY>
```

Figure 42: XML Specification of Index Category **Function-Pair-Type**

6.4 Graphical Illustration of Categorical Models in RASF

This section states the collaboration and supervision to the master students involved in the RASF project for the research work 11) in Figure 2.

6.4.1 Categorical Modeling Language (CML)

CML is a powerful modeling language with a formal basis from the category theory and graphical modeling notations. The convention of the graphical model is an adaptation of the category theory convention where a circle represents an object in a category and a directed arrow represents a morphism. The CML specification is constructed using the CML formal grammar. The grammar also serves as a basis for generating the XML file

for the CML models constructed using a categorical modeling tool [81].

CML uses Extended Backus-Naur Form (EBNF) for the grammar notation. The grammar can be used to determine the exact syntax for any category construct. An EBNF based grammar consists of “non-terminals” and “terminals”. Non-terminals are symbols within a BNF definition, also defined in the grammar. Terminals are endpoints in BNF definition, consisting of category theory keywords; all non-terminals appear in brackets < > and all terminals appear without brackets. The start symbol in the CML grammar corresponds to a list of non-terminals, each of which translates to a model in the CML as the following. More details about the grammar for the typed-category, functor, natural transformation, diagram, cone, cocone, limit, colimit and product in [81].

- <Typed_Category> consists of a keyword **TYPED-CATEGORY** followed by the non-terminals for the name and Id of the category. The keyword ***Types of Objects*** serves as a heading for a list of object types. The keyword ***Objects*** with a notation for a set of objects in the category is followed by a list of objects in that category.
- <Object_Type> consists of a list of the type names and Ids for each type and a list of object type instances with name and Id for each instance.
- <Object> consists of the object type name and Id followed by the instance objects for that type.
- <Morphism_Type> consists of a name of the morphism type followed by a list of morphisms for that type.
- <Morphism> is <Mor_Instance> that is a list of morphism instances for each

morphism type followed by <Mor_Identity>.

- <Mor_Identity> is the list of Identity morphisms for each object instance in <Object>.
- <Axiom> consists of all properties that must hold true to prove the correctness of the models according to the category theory. It primarily consists of <Property> that is <Identity> and <Associativity>.
- <Id> is a symbol for construction of the names and Ids in CML. It consists of one or more characters.
- The non-terminal <Character> consists of all alphabets and digits from 0 to 9.
- <Empty> facilitates the termination of a name or Id with an empty space.

6.4.2 Graphical Illustration Tool

The name of our graphical illustration tool is CATCanvas, which is inspired from the visual models of category theory constructed on a drawing canvas. So far there are two categorical constructs have been implemented in CATCanvas: Category and Functor. For each construct there is a separate view and drawing canvases. A categorical model can be either drawn manually or imported from a XML file to the canvas. Similarly the model may also be exported to a XML file or saved as an image file. CATCanvas is a Web-based application running in a Flash player, and its UI is a Flex-based Web UI built by the MXML controls [81]. There is a “Rules Engine” in CATCanvas that is responsible for the construction of categorically correct models. The engine plays an active role when performing functor mapping. For the constructed diagrams (models), the XML generator

can build the XML specifications and send them to the Web UI in terms of exporting them to files. The XML parser can extract the XML files and send the parsed data to the UI in terms of rendering the graphical models. Figure 44 depicts an example of using CATCanvas and exporting to a XML file, which is illustrated in Figure 43. More details about our graphical tool can be found in [81].

```
<?xml version="1.0" encoding="utf-8" ?>
- <category name="PAM Team">
    <object name="TM" type="Messenger" />
    <object name="WIM" type="Worker" />
    <object name="L" type="Leader" />
    <object name="WIR" type="Worker" />
    <object name="WAL" type="Worker" />
    <morphism name="Id(L)" type="Identity" fromObject="L" toObject="L" />
    <morphism name="Id(WIR)" type="Identity" fromObject="WIR" toObject="WIR" />
    <morphism name="Id(WAL)" type="Identity" fromObject="WAL" toObject="WAL" />
    <morphism name="Id(WIM)" type="Identity" fromObject="WIM" toObject="WIM" />
    <morphism name="Id(TM)" type="Identity" fromObject="TM" toObject="TM" />
    <morphism name="c1" type="Cooperate" fromObject="TM" toObject="WIM" />
    <morphism name="c2" type="Cooperate" fromObject="TM" toObject="WIR" />
    <morphism name="c3" type="Cooperate" fromObject="TM" toObject="WAL" />
    <morphism name="m3" type="Manage" fromObject="L" toObject="WIM" />
    <morphism name="m2" type="Manage" fromObject="L" toObject="WIR" />
    <morphism name="m1" type="Manage" fromObject="L" toObject="WAL" />
    <morphism name="c4" type="Cooperate" fromObject="WIM" toObject="WIR" />
    <morphism name="c5" type="Cooperate" fromObject="WIR" toObject="WAL" />
    <morphism name="u" type="Unique" fromObject="L" toObject="TM" />
  </category>
```

Figure 43: A Sample of the XML File Exported from the Graphical Model [81]

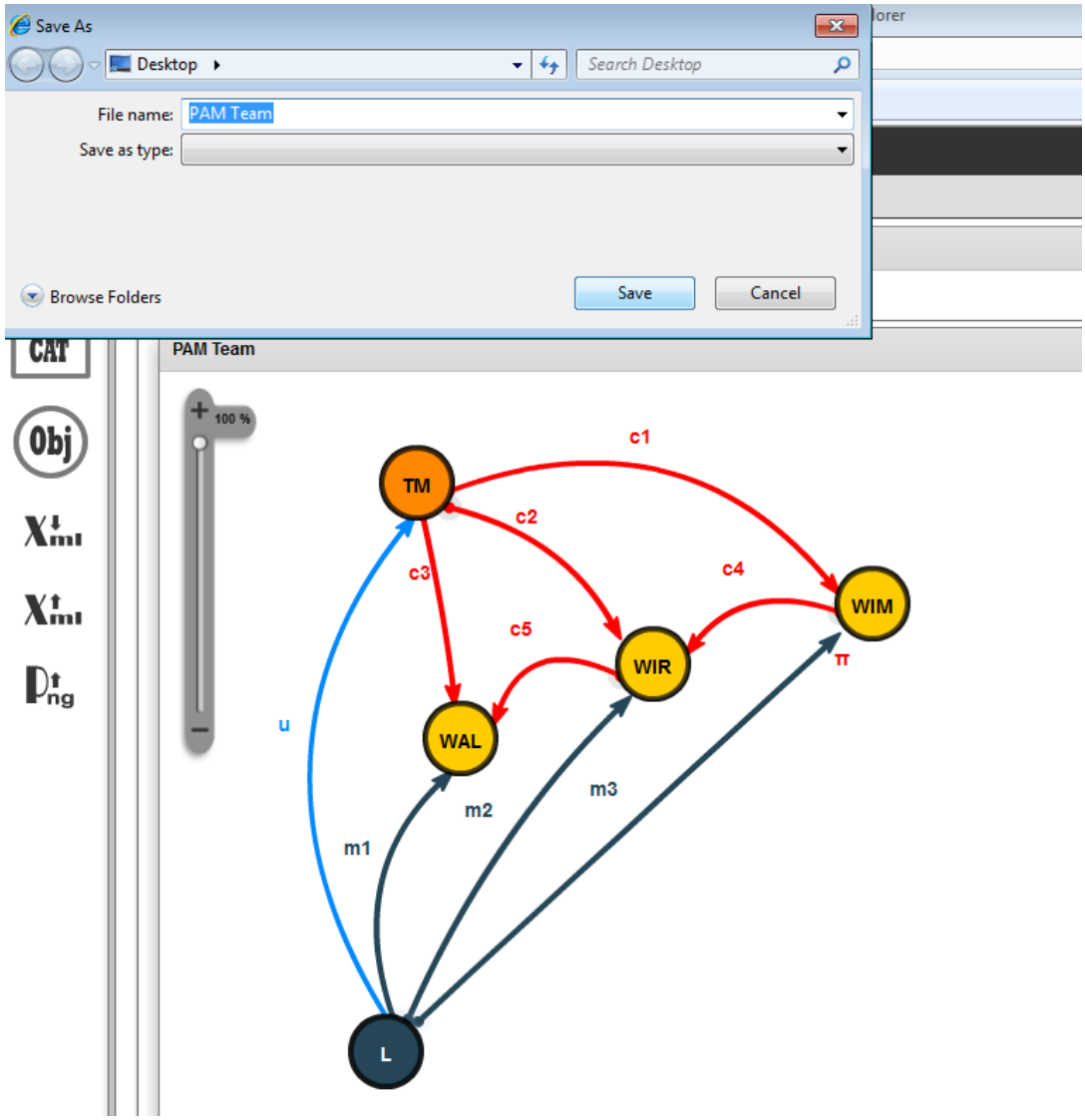


Figure 44: An Example of Using CATCanvas and Exporting to a XML File [81]

6.5 Categorical Specification of MAS Model in RASF

This section states the collaboration and supervision to the master students involved in the RASF project for the research work 13) in Figure 2. In Section 5.3, we stated a mapping from the RAS model to MAS model in the RASF, and we will introduce the categorical specification of the MAS model in this section, such as plans, goals, beliefs and their relationships.

6.5.1 Plans

Plans represent the agent's means to act on the requests initiated by other agents or from its environment, and one single plan is abstracted as a sequence of actions. Thus, plans of an agent are collections of sequences of actions, which are performed in a discrete time.

Definition 6.5.1: **Action** is a discrete category (see Definition 3.1.5) whose objects are actions in the intelligent control loop, denoted by Act_1, Act_2…, and the only morphisms are identity morphisms of those objects [62].

Definition 6.5.2: **Plan** is a category that represents one plan whose objects are actions denoted by Act_1, Act_2… and morphisms are *before* that model the partial order between the actions. A sequence of actions can be understood as a path in category theory (see Property 6.0.3), and only paths of length equal or less than one are considered as morphisms. Inside **Plan**, we define a special object denoted as Act_{Null} (null action), and it doesn't have any morphism from or to other actions; it is used to catch exceptions [62].

Definition 6.5.3: **PLAN** is a category whose objects are plans denoted by $Plan_1$, $Plan_2$… and morphisms are *before* that model the partial order between the plans. This partial order can be understood as a path in category theory [see Property 6.0.3], and only paths of length equal or less than one are considered as validated morphisms. Inside **PLAN**, we define a special object, called $Plan_{Null}$ (null object), and it doesn't have any morphism from or to other plans; it is used to catch exceptions [62].

Definition 6.5.4: *sequence _action* is a functor from **Action** to **Plan**. It provides a rule mapping all the *actions* in **Action** to *actions* in **Plan**, and all the identity morphisms

in **Action** to identity morphisms in **Plan** [62].

Definition 6.5.5: *refined _by _plan* is a functor from **Plan** to **PLAN**. It means that the *actions* in **Plan** are used to complete or build plans in **PLAN**; it also provides a rule that maps all the *actions* in **Plan** to *plans* in **PLAN**, and all the morphisms in **Plan** to identity morphisms in **PLAN** [62].

Definition 6.5.6: *timing _action* is a functor from the **Plan** to **Discrete-Time** (see Property 6.2.3), which maps *actions* in **Plan** to *time units* in **Discrete-Time**, and maps *before* in **Plan** to *before* in **Discrete-Time** [62].

Definition 6.5.7: *timing _plan* is a functor from **PLAN** to **Discrete-Time** (see Property 6.2.3), which maps *plans* in **PLAN** to *time units* in **Discrete-Time**, and maps *before* in **Plan** to *before* in **Discrete-Time** [62].

6.5.2 Goals

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

Definition 6.5.8: **GOAL** is a category whose objects are *goals* and morphisms are *depends*. The definition of *depends* can be the domain of this morphism having a higher or the same priority level than its co-domain. Inside **GOAL**, there is a special goal denoted by $Goal_{Null}$ that stands for an empty object with no morphism from or to other goals; it is used to catch exceptions [62].

Definition 6.5.9: **Dependency** is a category whose objects are integers such as *1*, *0*,

-1 or *unsigned*, and morphisms are *more-than* denoted as "≥". The object *unsigned* doesn't have any relations (morphisms) with other objects. It is used to set up the order of importance or urgency of different goals [62].

Definition 6.5.10: *assigned_dependency* is a functor from **GOAL** to **Dependency**. It models the fact that *goals* in **GOAL** can be assigned to related order in **Dependency**; *depends* in **GOAL** can be mapped to *more-than* in **Dependency** [62].

6.5.3 Beliefs

Beliefs represent agent's knowledge or information about environment and itself. Beliefs are built from different information called *facts*, which are organized into different sets denoted as *fact sets*.

Definition 6.5.11: **FactSet** is a discrete category where objects are facts and the only morphisms are identity morphisms. The facts are information or knowledge about the agent's environments and system. Based on the different usage, facts are classified into different categories **FactSet**. Two special categories of **FactSet** need to be introduced: $\mathbf{FactSet_{Base}}$ and $\mathbf{FactSet_{Null}}$. $\mathbf{FactSet_{Base}}$ includes all the facts every other **FactSet** has, and $\mathbf{FactSet_{Null}}$ contains no facts at all or it's an empty set. Inside **FactSet** (includes $\mathbf{FactSet_{Base}}$, except $\mathbf{FactSet_{Null}}$), we define a special object denoted as $\mathrm{Fact_{Null}}$ (a null fact) which doesn't have morphisms. It is used to catch exceptions [62].

Definition 6.5.12: **BELIEF** is a category of Sets (see Definition 3.1.6), whose objects are categories **FactSets** (one $\mathbf{FactSet_{Base}}$ as well as one $\mathbf{FactSet_{Null}}$ are included as default), and the morphisms are *subset _of*. Any **FactSet** is a subset of $\mathbf{FactSet_{Base}}$, and

more formally, every fact within **FactSet** can be found in $\mathbf{FactSet_{Base}}$. Similarly, $\mathbf{FactSet_{Null}}$ has *subset _of* relations to every **FactSet** [62].

6.5.4 Agents

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

Definition 6.5.13: *plan _goal* is a functor from **PLAN** to **GOAL**. It captures the fact that every *Plan* from **PLAN** dispatches a goal from **GOAL**. Every *Plan* in **PLAN** can be mapped to one *Goal* in **GOAL**, and morphisms *before* in **PLAN** can be mapped to morphisms *depends* in **GOAL**. The functor grantees that: one plan can only dispatch one corresponding goal, but different plans can dispatch a same goal [62].

Definition 6.5.14: *plan _belief* is a functor from **PLAN** to **BELIEF**. It means that agent plans have access to read or write facts from agent's beliefs. Every *plan* in **PLAN** can be mapped to one *FactSet* (either $FactSet_{Base}$ or $FactSet_{Null}$) in **BELIEF**, and all the morphisms in **PLAN** are mapped to the identity morphism of $FactSet_{Null}$ in **BELIEF**. The functor formalizes the communication from plans to beliefs. [62].

Definition 6.5.15: *goal _belief* is a functor from **GOAL** to **BELIEF**. It means every goal has an access to read facts or knowledge from agent beliefs. Every *Goal* in **GOAL** is mapped to one *FactSet* in **BELIEF**, and morphisms *depends* in **GOAL** are mapped to the

identity morphism of $FactSet_{Null}$ in **BELIEF**. The functor formalizes the communication from goals to beliefs. By this functor, goals are able to read data from beliefs and justify if they can be accomplished [62].

Definition 6.5.16: An agent can be represented by categories: **Action**, **Plan**, **PLAN**, **GOAL**, **BELIEF** and **FactSet** with functors*: plan _goal, plan _belief, goal _belief, refined _by _plan* and *sequence _action* [62].

Definition 6.5.17: **MAS** is a category whose objects are *agents* and morphisms are *communication* which represents that one agent has activities of conveying information to another agent; *communication* can be differentiated by types [62].

6.5.5 Repository Agent

Definition 6.5.18: **Repository-Type** is a type category (see Property 6.0.1) whose objects represent the types of agents, and morphisms represent the types of connections between those repository types. **Repository-Type-Instance** is a category whose objects represent repositories, and morphisms represent connections between those repositories.

Definition 6.5.19: *Repository-Access* that maps *agents* in **MAS** to *repositories* in **Repository-Type-Instance**, and maps every *communication* in **MAS** to each *connection* in **Repository-Type-Instance**.

6.6 Representation of Categorical MAS Models in RASF

This section states the research activity 19) in Figure 3. After having the categorical MAS models in RASF, we need to express them using XML in terms of feeding them to our graphical illustration tool as the following example, and more XML representation can be

found in Appendix C.

```
<CATEGORY name = "Plan-Type">
    <OBJECT>
        <OBJECT name = "Plan-Type_i"/>
        <OBJECT name = "Plan-Type_j"/>
    </OBJECT>
    <MORPHISM>
        <MORPHISM name = "Partial-Order"/>
            <FROM-OBJECT name = "Plan-Type_i"/>
            <TO-OBJECT name = "Plan-Type_j"/>
        </MORPHISM>
    </MORPHISM>
</CATEGORY>
```

Figure 45: XML Specification of Type Category **Plan-Type**

6.7 Summary

In this chapter, we described the categorical RASF model, transformation from the categorical RAS model to its XML specification as well as transformation from categorical MAS model to its XML specification.

Based on the architecture model in RASF we presented in Chapter 5, the internal structure of a RAS is both hierarchical (composition of its RAE) and recursive (tiered interconnectivity of its RAE). Therefore, the behavior of a RAS can be characterized by its RAE and their interactions. The categorical models for the structure and behavior in RASF were generated in Section 6.1 & 6.2 respectively.

After having the categorical models in RASF, we expressed them using XML in Section 6.3 for feeding them to our graphical illustration tool CATCanvas, which was introduced in Section 6.4.

Finally, the categorical specifications of the MAS model (plans, goals, beliefs and their relations) together with their XML representations were introduced in Section 6.5 & 6.6. We will introduce the categorical specifications of the self-healing property for the RASF in next chapter.

Chapter 7: Categorical Specifications of Self-Healing in RASF

This chapter states the research activities 10), 11) and 20) in Figure 3 that are prototype design of self-healing, prototype design of categorical specification for self-healing and transformation from the categorical self-healing property to its XML specification. I had two publications [177 & 87] for this chapter. After having described the categorical specification of the RAS model, MAS model with their representation and graphical illustration, a categorical specification of self-healing is explained in this chapter.

In order to simplify our description, we use the Reactive Autonomic Elements (RAE) to represent RAO, RAOL, RAC, RACS, RACG and RACGM. The interactive behavior between RAE in the RAS model can be specified as external event sequences.

If we consider each RAE as a black box with corresponding internal reactive or autonomic behavior, only the external events crossing the RAE boundary are observable, so the interactions between RAE to achieve a usage goal are performed by interchanging their external events that are of two types: 1) input (from destination RAE to source RAE); 2) output (from source RAE to destination RAE). Moreover, the timing requirements can be modeled as global clock events: a *Tick* abstracting one time unit and a *NTR* abstracting *No Timing Requirement*. Incorporating the *Ticks* into interaction patterns allows for the performance analysis of the minimum and maximum delay time for each RAE usage during design validation and testing phase.

We extract the event sequencing and the timing constraints in the form of rules that relate to: 1) *Input Events* (*IE*) to *IE*; 2) *IE* to *Output Events* (*OE*); 3) *OE* to *OE*; 4) *OE* to

IE; 5) *Trigger Events* (*TE*) to *Last Events* (*LE*), where *LE* is an event finalizing the usage of a RAE started by *TE*. The above knowledge is sufficient to produce generic interaction patterns between RAE. The first event in such an interaction pattern is always a *TE* from source RAE, and the order between events in one pattern must satisfy sequencing rules for the partial order between RAE. We call such a pattern a *legal sequence*. More details about the algorithm for generating exhaustive set of legal sequences from a given set of requirements can be found in [125].

Figure 47, 48 and 49 show the specification of a sample RAE depicted in Figure 46 by which we can illustrate the self-healing property in RASF.

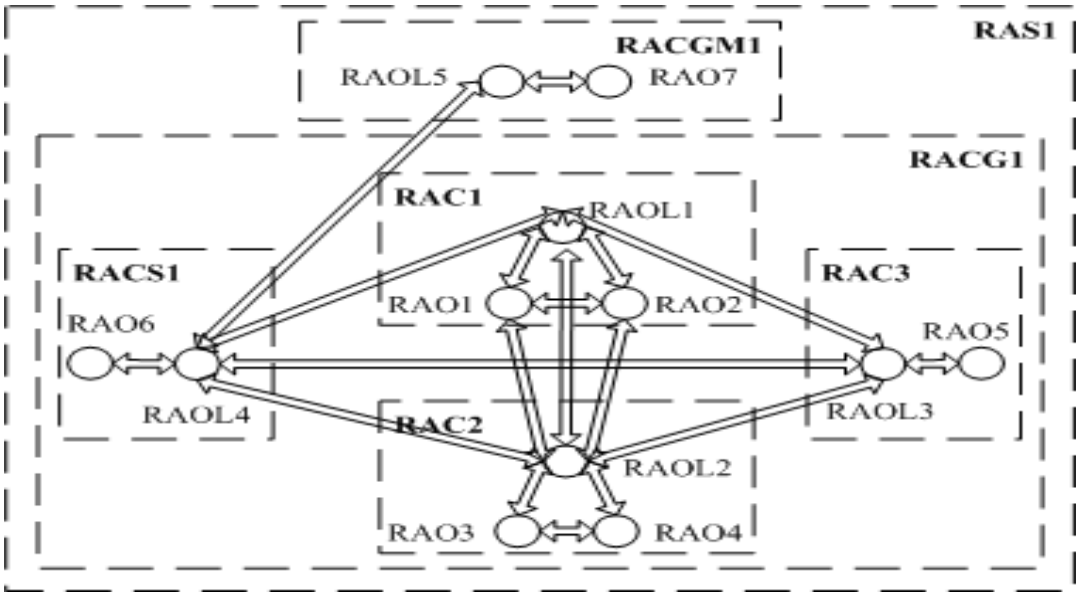


Figure 46: Example of RAS Application Model

```
RAC <RAC1>
 Members: <RAO1, RAO2, RAOL1>
 Configure: <(RAO1, RAO2), (RAO1, RAOL1), (RAO2, RAOL1),
                (RAO1, RAOL2), (RAO2, RAOL2)>
 Leader: <RAOL1>
 Supervisor: <RACS1>
 Neighbors: <RAC2, RAC3>
 Repository: <local repository>
End RAC
```

Figure 47: Specification of RAC1 Structure

```
RACG <RACG1>
 Members: <RAOL1, RAOL2, RAOL3, RACS1>
 Configure: <(RAOL1, RACS1), (RAOL2, RACS1), (RAOL3, RACS1),
             (RAOL1, RAOL2), (RAOL1, RAOL3), (RAOL2, RAOL3)>
 Supervisor: <RACS1>
 Manager: <RACGM1>
 Neighbors: <null>
 Repository: <local repository>
End RACG
```

Figure 48: Specification of RACG1 Structure

```
RAS <RAS1>
 Members: <RACS1, RACGM1>
 Configure: <(RACS1, RACGM1)>
 Manager: <RACGM1>
 User Console: <Console1>
 Neighbors: <null>
 Repository: <local repository>
End RAS
```

Figure 49: Specification of RAS1 Structure

7.1 Scenario1: Crashed RAO

After *RAO1* is started by *RAOL1*, it begins to send its heartbeat messages to *RAOL1* every *t* ticks [177], while *RAOL1* is in the first state of its intelligent control loop,

monitoring the status of *RAO1*. If *RAOL1* receives the heartbeat messages from *RAO1*, *NoChange* event keeps it in *Monitor* state; otherwise, *RAO1-Crashed* event is triggered and then *RAOL1* transits to *Analyze* state, while a time constraint variable (*TCvar1*) is initialized to work as a local clock in terms of time constraints on each transition of the intelligent control loop. The value of *TCvar1* is *t0*, *t1*, *t2*, *t3*... where $t0 < t1 < t2 < t3$. After *RAOL1* enters *Analyze* state, it sends a *Restart* message to *RAO1* in *t0* ticks. If *RAO1* is recoverable, *NoAction* event occurs and *RACS1* goes back to *Monitor* state, while the *TCvar1* is reset; otherwise, *RAOL1* transits to *Plan* state triggered by *HasAction* event in *t1* ticks (see Figure 52).

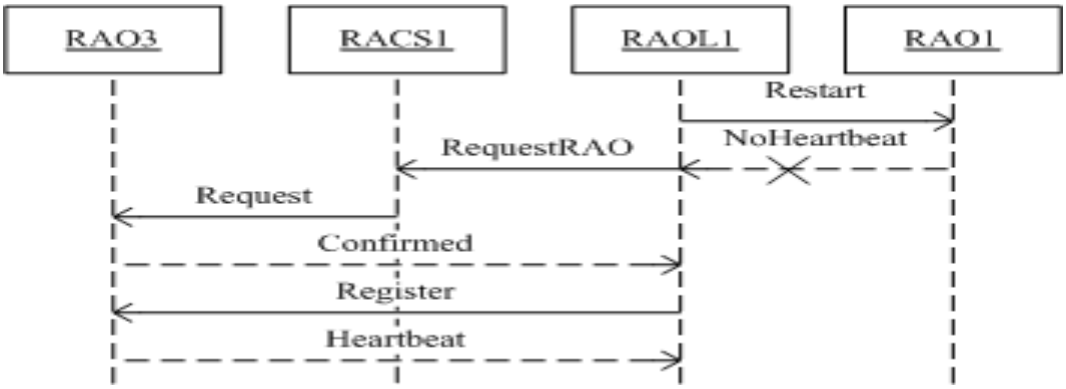


Figure 50: Substitution Work Flow in RAC1

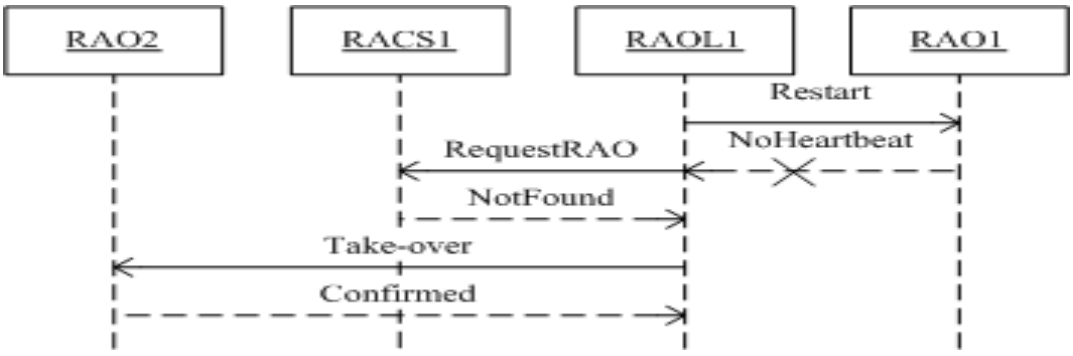


Figure 51: Take-Over Work Flow in RAC1

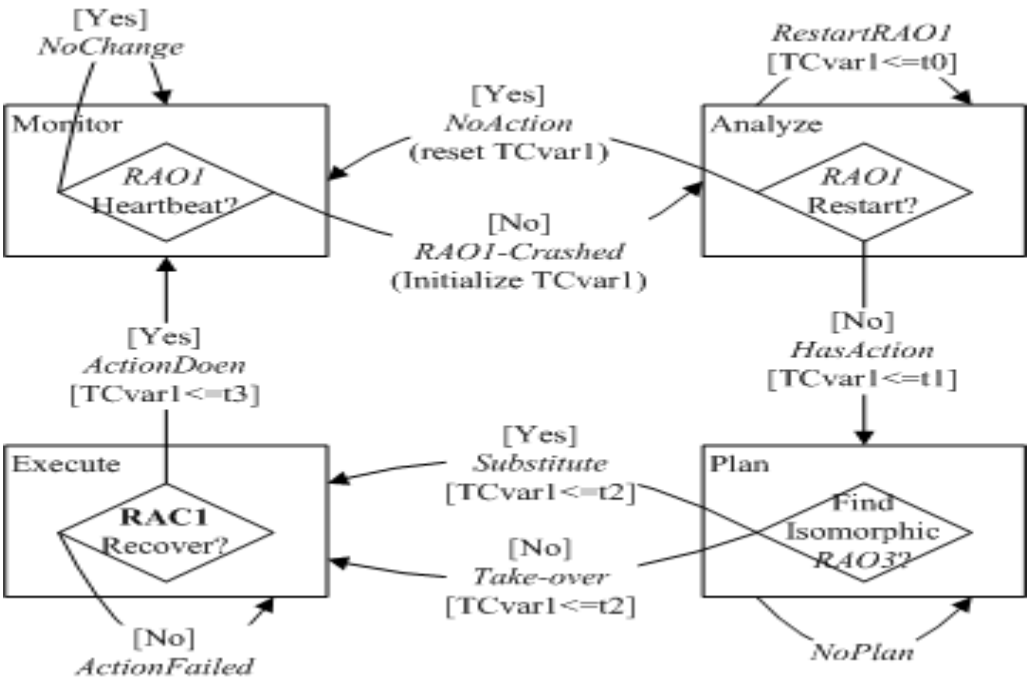


Figure 52: Intelligent Control Loop in RAOL1 for Self-Healing

When *RAOL1* is in *Plan* state, it sends the *RequestRAO1* message with the type information of *RAO1* to *RACS1*, and the latter tries to replace *RAO1* by an available *RAO* which is isomorphic to *RAO1* in **RAS1**, such as *RAO3*, since isomorphic objects behave in the same way. If *RACS1* finds *RAO3* successfully, *RAOL1* chooses the plan *Substitute* and transits to *Execute* state triggered by *Substitute* event in *t2* ticks; otherwise, it selects the plan *Take-over* and enters *Execute* state triggered by *Take-over* event in *t2* ticks. In this plan, *RAO2* takes the responsibilities of *RAO1* and works as the product object of original *RAO1* and *RAO2*, since they behave as a synchronous product machine.

When *RAOL1* is in *Execute* state and the plan *Substitute* is applicable, *RAOL1* sends a *register* message to *RAO3* and then initializes it to the status of *RAO1* according to the checkpoint made before. When the plan *take-over* is applicable, *RAOL1* sends a *take-over*

message to *RAO2* and update it to the status of the synchronous product machine of *RAO1* and *RAO2* based on the checkpoint. After the plan execution, *RAOL1* validates the behavior of **RAC1** and its evolution (see Property 6.1.3) **RAC1'** based on categorical specifications. If **RAC1'** has equivalent behavior with **RAC1**, *ActionDone* event occurs and *RAOL1* transits to *Monitor* state in *t3* ticks; otherwise, *ActionFailed* event keeps it in *Execute* state for the user intervention from *User Console* by *RACS1* and *RACGM1*.

7.2 Categorical Illustration of Scenario1

The actions in the substitution work flow and take-over work flow of RAC1 can be specified as the categories where objects are those actions (*Restart*, *RequestRAO1*, *Register*, etc.), and morphisms are their preorder relationship *before*. Each object (action) in those categories is a quadruple (see Property 6.2.28). For example, *RequestRAO* = (*RAOL1*, *NoHeartbeat-AfterRestart*, *SearchRAO*, *RACS1*). Moreover, the sequences of those actions can be specified as the categories in which objects are those sequences (<*Restart*, *NoHeartbeat*, *RequestRAO*, *Request*>, <*RequestRAO*, *Request*, *Confirmed*, *Register*, *Heartbeat*>), and morphisms are *equivalence* relationship between those sequences (see Property 6.2.31).

The transitions in the intelligent control loop of RAOL1 for self-healing can be specified as a category in which objects are those transitions (*NoChange, RAO1-Crashed*, *RestartRAO1*, etc.), and morphisms are their preorder relations *before*. Each object (transition) in that category is a triple (see Property 6.2.23). For example, *RAO1-Crashed* = (*Monitor*, *NoHeartbeat*, *Analyze*). Moreover, the sequences of those transitions can be

specified as a category in which objects are those sequences (<*No- Change*, *RAO1-Crashed*, *RestartRAO1*, *NoAction*>, <*RestartRAO1*, *HasAction*, *Take- over*, *ActionDone*>), and morphisms are *equivalence* relation between those sequences (see Property 6.2.26).

CAT-RAC <RAC1>
Objects: <RAO1, RAO2, RAOL1>
Morphisms: <Work1(RAO1, RAO2), Work2(RAO2, RAO1), Report1(RAO1, RAOL1),
Order1(RAOL1, RAO1), Report2(RAO2, RAOL1), Order2(RAOL1, RAO2)>
Limit Object: <RAOL1>
Colimit Object: <RAOL1>
Product Objects: <SPM1(RAO1, RAO2), SPM2(RAO1, RAOL1), SPM3(RAO2, RAOL1)>
Coproduct Objects: <null>
Pushout Objects: <NR1(Order1, Order2), NR2((RAOL2, RAO1), (RAOL2, RAO2))>
Pullback Objects: <PR1((RAO1, RAOL2), (RAO2, RAOL2)), PR2(Report1, Report2)>
Slice Category: <(Report1, Report2), Work1>
Co*slice Category*: <(Order1, Order2), Work2>
Functors: <Restart(RAC1-1, RAC1-0), Substitute(RAC1-2, RAC1-0), Take-over(RAC1-3, RAC1-0)>
Natural Transformations: <NT1(Restart, Substitute), NT2(Substitute, Take-over), NT3(Restart, Take-over)>
Functor Category: <(Restart, Substitute, Take-over), NT1, NT2, NT3>
End CAT-RAC

Figure 53: Categorical Constructs in RAC1 Representation

Figure 53 depicts the categorical representation of RAC1 before *RAO1* is crashed. The category **RAC1** consists of three objects *RAO1*, *RAO2* and *RAOL1*. The bidirectional communications among those objects are six morphisms to specify working collaboration between *RAO1* and *RAO2*, as well as the leadership from *RAOL1* to *RAO1* and *RAO2* (see Property 6.1.2). The SPM (Synchronous Product Machines) for the synchronous communication between two objects in **RAC1** can be represented by SPM1, SPM2 and SPM3 (see Property 6.2.6). Slice category models *Report* actions (Report1 and Report2)

with their relations (Work1) from *RAO1* and *RAO2* to *RAOL1* (see Property 6.2.12); coslice category may interpret *Order* actions (Order1 and Order2) with their relations (Work2) from *RAOL1* to *RAO1* and *RAO2* (see Property 6.2.13).

Let **RAC1** be a subcategory (consisting of objects RAOL1, RAO1, RAO2 and the morphisms among them) of **RAC1-0** (a category consisting of all potential RAE for the self-healing in RAC1). If **RAC1** is recovered by restarting crashed *RAO1*, it evolves to **RAC1-1** (consisting of objects RAOL1, RAO1-1, RAO2 and the morphisms among them in **RAC1-0**), which has the same categorical structure as **RAC1** except for the different initial status of *RAO1*. This evolution is represented by the *Restart* functor (a structure-preserving mapping) from **RAC1-1** to **RAC1-0**. If **RAC1** is recovered by substituting *RAO1* by its isomorphic object *RAO3* (see Definition 3.1.3), it will evolve to **RAC1-2** (consisting of objects RAOL1, RAO3, RAO2 along with the morphisms among them in **RAC1-0**), which has the same categorical structure as **RAC1** but replacing *RAO1* with *RAO3*. The above is specified by the *Substitute* functor, a structure-preserving mapping (see Figure 54).

However, if **RAC1** is recovered by asking *RAO2* to take over the responsibilities of *RAO1*, it will evolve to **RAC1-3** (consisting of objects RAOL1-1, SPM1 along with the morphisms among them in **RAC1-0**), which has different categorical structure, but both of them have the equivalent social lives (see Property 6.2.33). The mapping among those evolutions *Restart*, *Substitute* and *Take-over* of the **RAC1** can be interpreted as natural transformations (see Property 6.1.4). The functor category having those functors as

objects and their natural transformations as morphisms illustrate all possible evolutions with their relations. In addition, the natural transformation3 is a composition of natural transformation1 and natural transformation2, which may be interpreted as the following: the result of the evolution *Restart -> Substitute -> Take-over* is equivalent to the evolution *Restart -> Take-over*. Figure 56, 57 and 58 depict those natural transformations and their composition respectively.

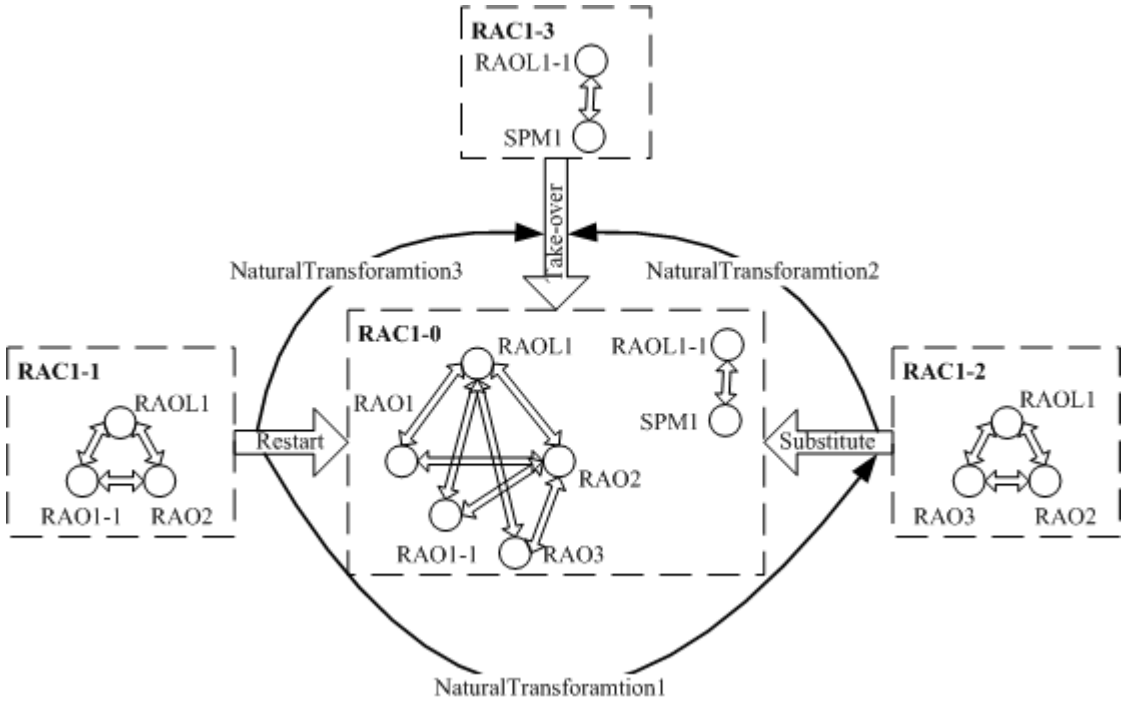


Figure 54: Evolution for Self-Healing in RAC1

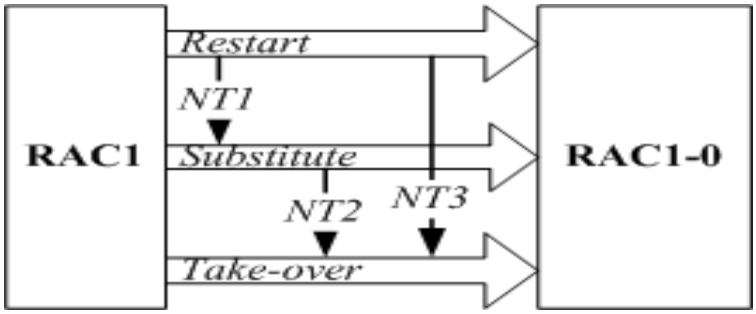


Figure 55: Natural Transformation for Self-Healing in RAC1

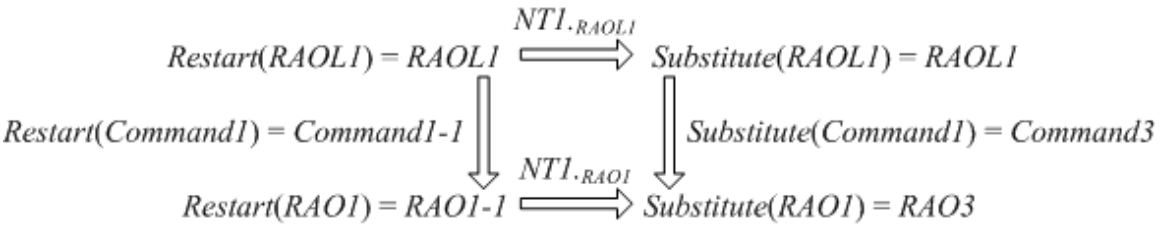


Figure 56: Natural Transformation from *Restart* to *Substitute* in RAC1

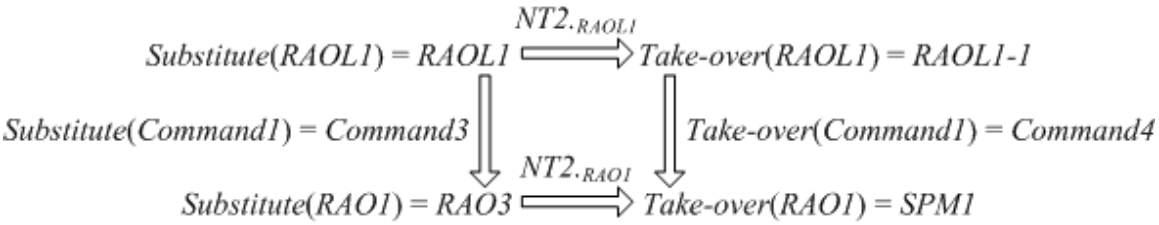


Figure 57: Natural Transformation from *Substitute* to *Take-over* in RAC1

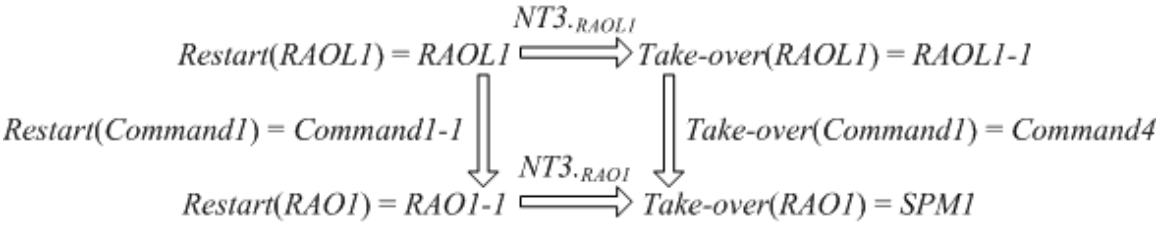


Figure 58: Natural Transformation from *Restart* to *Take-over* in RAC1

When both *RAO1* and *RAO2* are crashed at the same time, *RAOL1* tries to restart them first. If neither of them can be restarted, *RAOL1* will send a message to *RACS1* for requesting the isomorphic objects (*RAO3* and *RAO4*) of *RAO1* and *RAO2*; otherwise, the remaining process is the same as the illustration above. When *RACS1* cannot find *RAO3* and *RAO4*, *RAOL1* will take over the responsibilities of *RAO1* and *RAO2*, working as a product object of them, such as *SPM1*; otherwise, the description above may indicate the remaining process. If a **RAC** consists of more than two *RAOL*, the similar categorical representation can be generated as we explained previously.

In this scenario, we proposed three solutions (*restart*, *substitute* and *take-over*) for

the fault-tolerance in case of the crashed RAO, which cover all possible situations in terms of self-healing from the practical usage.

7.3 Scenario2: Crashed RAOL

After *RAOL1* is started by *RACS1*, it begins to send its heartbeat messages to *RACS1* every *t* ticks, while *RACS1* is in the first state of its intelligent control loop, monitoring the status of *RAOL1*. If *RACS1* receives the heartbeat messages from *RAOL1*, *NoChange* event keeps it in *Monitor* state; otherwise, *RAOL1-Crashed* event is triggered and then *RACS1* transits to *Analyze* state, while a time constraint variable (*TCvar2*) is initialized to work as a local clock in terms of time constraints on each transition of the intelligent control loop. The value of *TCvar1* is *t0*, *t1*, *t2*, *t3*... where $t0 < t1 < t2 < t3$. After *RACS1* enters *Analyze* state, it sends a *Restart* message to *RAOL1* in *t0* ticks. If *RAOL1* is recoverable, *NoAction* event occurs and *RACS1* goes back to *Monitor* state, while the *TCvar2* is reset; otherwise, *RACS1* transits to *Plan* state triggered by *HasAction* event in *t4* ticks (see Figure 61).

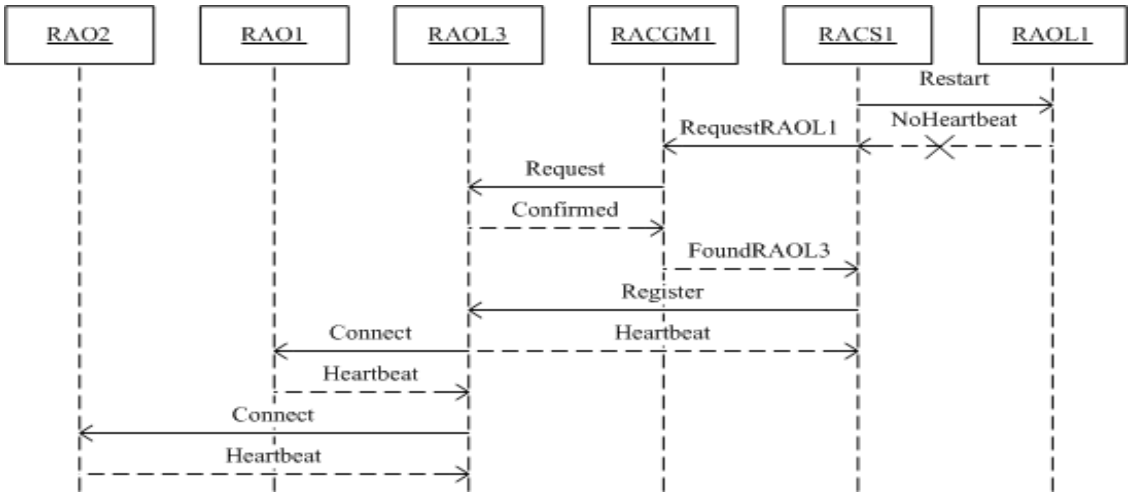


Figure 59: Substitution Work Flow in RACG1

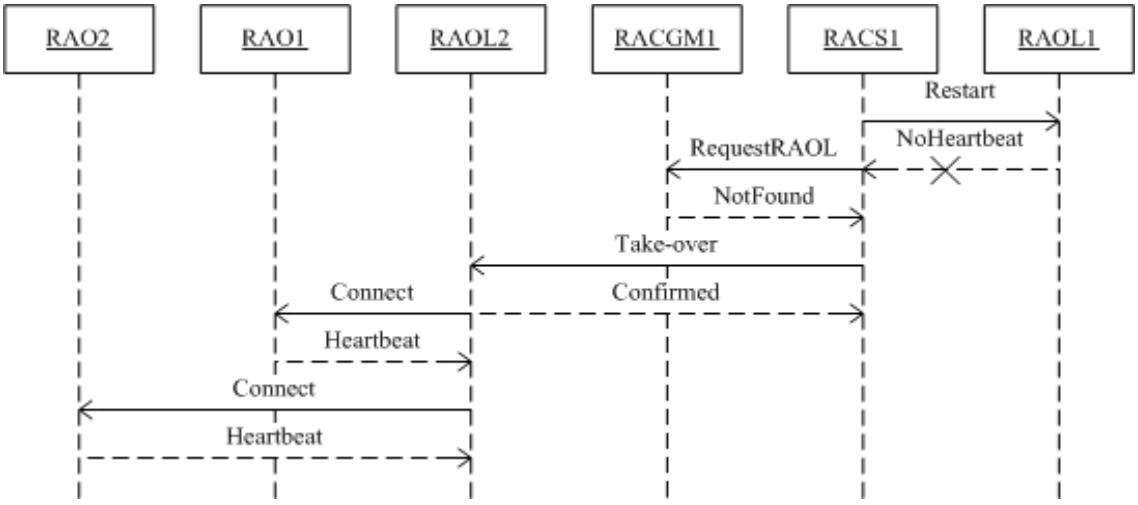


Figure 60: Take-Over Work Flow in RACG1

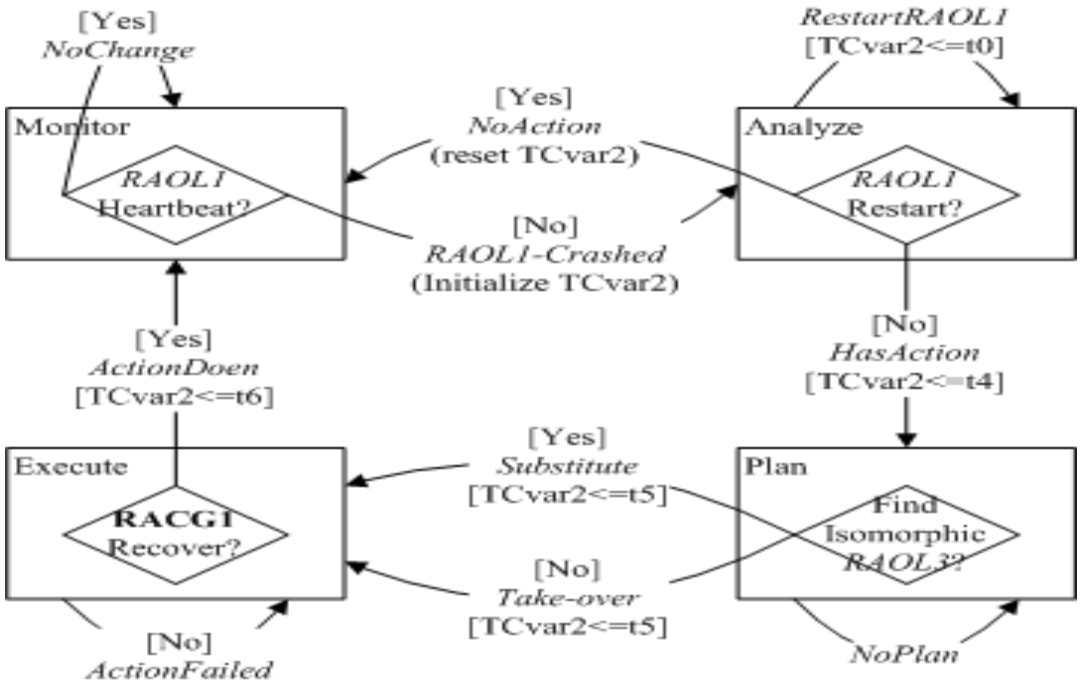


Figure 61: Intelligent Control Loop in RACS1 for Self-Healing

When *RACS1* is in *Plan* state, it sends the *RequestRAOL1* message with the type information of *RAOL1* to *RACGM1*, and the latter tries to replace *RAOL1* by an available *RAOL* which is isomorphic to *RAOL1* in **RAS1**, such as *RAOL3*, since isomorphic objects behave in the same way. If *RACGM1* finds *RAOL3* successfully, *RACS1* chooses the plan

Substitute and transits to *Execute* state triggered by *Substitute* event in *t5* ticks; otherwise, it selects the plan *Take-over* and enters *Execute* state triggered by *Take-over* event in *t5* ticks. In this plan, *RAOL2* takes the responsibilities of *RAOL1* and works as the product object of the original *RAOL1* and *RAOL2*, since they behave as a synchronous product machine.

When *RACS1* is in *Execute* state and the plan *Substitute* is applicable, *RACS1* sends a *register* message to *RAOL3* and then initializes it to the status of *RAOL1* according to the checkpoint made before. When the plan *take-over* is applicable, *RACS1* sends a *take-over* message to *RAOL2* and update it to the status of the synchronous product machine of *RAOL1* and *RAOL2* based on the checkpoint. After the plan execution, *RACS1* validates the behavior of **RACG1** and its evolution (see Property 6.1.3) **RACG1'** according to their categorical specifications. If **RACG1'** has equivalent behavior with **RACG1**, *ActionDone* event occurs and *RACS1* transits to *Monitor* state in *t6* ticks; otherwise, *ActionFailed* event keeps it in *Execute* state for the user intervention from *User Console* by *RACGM1*.

7.4 Categorical Illustration of Scenario2

The actions in the substitution work flow and take-over work flow of RACG1 can be specified as the categories where objects are those actions (*Restart*, *RequestRAOL1*, *Register*, etc.), and morphisms are their preorder relation *before*. Each object (action) in those categories is a quadruple (see Property 6.2.28). For example, *Register* = (*RACS1*, *RAOL3IsAvailable*, *RAOL3IsRegistered*, *RAOL3*). Furthermore, the sequences of those

actions can be specified as the categories where objects are those sequences (<*Register*, *Heartbeat*, *Connect*>, <*Connect*, *Heartbeat*, *Connect*, *Heartbeat*>), and morphisms are the *equivalence* relationship between those sequences (see Property 6.2.31).

The transitions in the intelligent control loop of RACS1 for self-healing can be specified as a category in which objects are those transitions (*HasAction, Substitute*, *ActionDone*, etc.), and morphisms are their preorder relations *before*. Each object (transition) in a category is a triple (see Property 6.2.23). For example, *HasAction* = (*Analyze*, *NoHeartBeatAfterRestart*, *Plan*). Moreover, the sequences of the transitions can be specified as a category where objects are those sequences (<*HasAction*, *Substitute*, *ActionFailed*>, <*HasAction*, *NoPlan*, *Take-over*, *ActionDone*>), and morphisms are the *equivalence* relation between those sequences (see Property 6.2.26).

Figure 62 depicts a categorical representation of RACG1 before *RAOL1* is crashed. The category **RACG1** consists of the objects *RAOL1*, *RAOL2, RAOL3* and *RACS1*. The bidirectional communications among those objects are morphisms to specify the working collaboration among *RAOL1*, *RAOL2, RAOL3*, as well as the leadership from *RACS1* to *RAOL1*, *RAOL2* and *RAOL3* (see Property 6.1.2). The SPM for the synchronous communication between two objects in **RACG1** can be represented by SPM5, SPM6 and SPM7 (see Property 6.2.6). Slice category models *Report* actions (Report1, Report2 and Report3) with their relations (Work1, Work3 and Work5) from *RAOL1*, *RAOL2* and *RAOL3* to *RACS1* (see Property 6.2.12); coslice category may interpret *Order* actions (Order1, Order2 and Order3) with their relations (Work2, Work4 and Work6) from

RACS1 to *RAOL1*, *RAOL2* and *RAOL3* (see Property 6.2.13).

CAT-RACG <RACG1>
Objects: <RAOL1, RAOL2, RAOL3, RACS1>
Morphisms: <Work1(RAOL1, RAOL2), Work2(RAOL2, RAOL1), Work3(RAOL1, RAOL3),
Work4(RAOL3, RAOL1), Work5(RAOL2, RAOL3), Work6(RAOL3, RAOL2),
Report1(RAOL1, RACS1), Report2(RAOL2, RACS1), Report3(RAOL3, RACS1),
Order1(RACS1, RAOL1), Order2(RACS1, RAOL2), Order3(RACS1, RAOL3)>
Limit Object: <RACS1>
Colimit Object: <RACS1>
Product Objects: <SPM5(RAOL1, RAOL2), SPM6(RAOL1, RAOL3), SPM7(RAOL2, RAOL3)>
Coproduct Objects: <MQ1(RAOL1, RACS1), MQ2(RAOL2, RACS1), MQ3(RAOL3, RACS1)>
Pushout Objects: <NR3(Order1, Order2), NR4(Work4, Work6)>
Pullback Objects: <PR3(Report1, Report2), PR4(Work3, Work5)>
Slice Category: <(Report1, Report2, Report3), Work1, Work3, Work5>
Co*slice Category*: <(Order1, Order2, Order3), Work2, Work4, Work6>
Subcategories: <RAC1, RAC2, RAC3>
Product Categories: <Interact1(RAC1, RAC2), Interact2(RAC1, RAC3), Interact3(RAC2, RAC3)>
Functors: <Restart(RACS1-1, RACS1-0), Substitute(RACS1-2, RACS1-0), Take-over(RACS1-3, RACS1-0)>
Natural Transformations: <NT1(Restart, Substitute), NT2(Substitute, Take-over), NT3(Restart, Take-over)>
Functor Category: <(Restart, Substitute, Take-over), NT1, NT2, NT3>
End CAT-RACG

Figure 62: Categorical Constructs in RACG1 Representation

Let **RACG1** be a subcategory (consisting of objects RACS1, RAOL1, RAOL2 and the morphisms among them) of **RACG1-0** (a category consisting of all potential RAE for the self-healing in RACG1). If **RACG1** is recovered by restarting crashed *RAOL1*, it evolves to **RACG1-1** (consisting of objects RACS1, RAOL1-1, RAOL2 along with the morphisms among them in **RACG1-0**), which has the same categorical structure as the **RACG1** except for different initial status of *RAOL1*. This evolution is represented by the *Restart* functor (a structure-preserving mapping) from the **RACG1-1** to **RACG1-0**. If **RACG1** is recovered by substituting *RAOL1* by its isomorphic object *RAOL3* (see

Definition 3.1.3), it will evolve to **RACG1-2** (consisting of objects RACS1, RAOL3, RAOL2 and the morphisms among them in **RAC1-0**) with the same categorical structure as **RACG1** but replacing *RAOL1* with *RAOL3*. The above is specified by the *Substitute* functor, a structure-preserving mapping (see Figure 63).

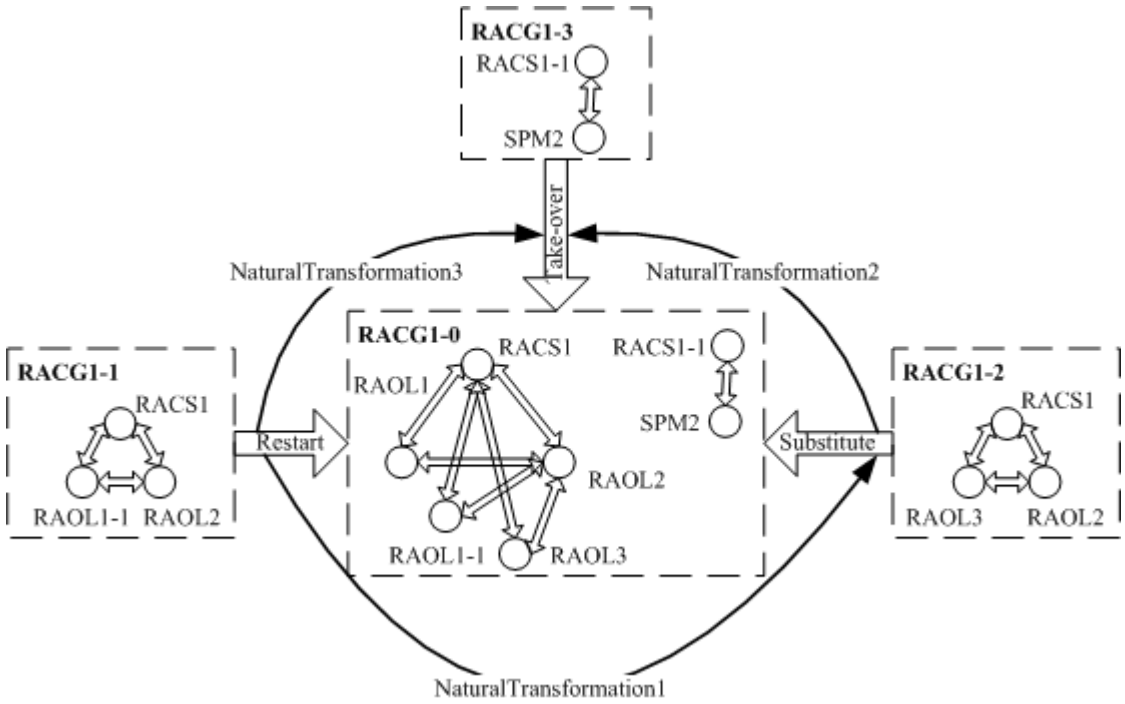


Figure 63: Evolution for Self-Healing in RACG1

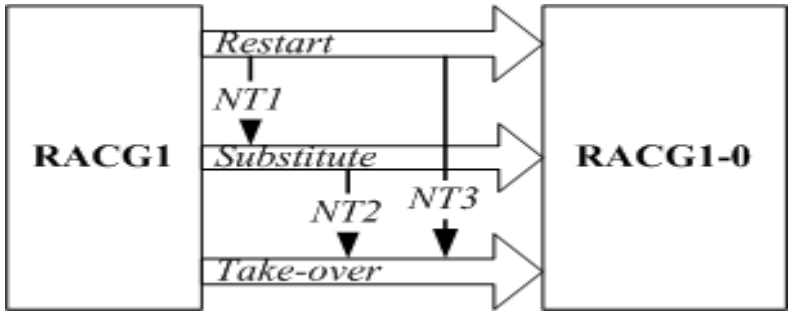


Figure 64: Natural Transformation for Self-Healing in RACG1

However, if **RACG1** is recovered by asking *RAOL2* to take over the responsibilities of *RAOL1*, it will evolve to **RACG1-3** (consisting of objects RACS1-1, SPM2 and the

morphisms between them in **RACG1-0**), which has different categorical structure, but both of them have the equivalent social lives (see Property 6.2.33). The mapping among those evolutions *Restart*, *Substitute* and *Take-over* of the **RACG1** can be interpreted as natural transformations (see Property 6.1.4). The functor category having those functors as objects and their natural transformations as morphisms illustrate all possible evolutions with their relations. In addition, the natural transformation3 is a composition of natural transformation1 and natural transformation2, which may be interpreted as the following: the result of the evolution *Restart -> Substitute -> Take-over* is equivalent to the evolution *Restart -> Take-over*. Figure 65, 66 and 67 depict those natural transformations and their composition respectively.

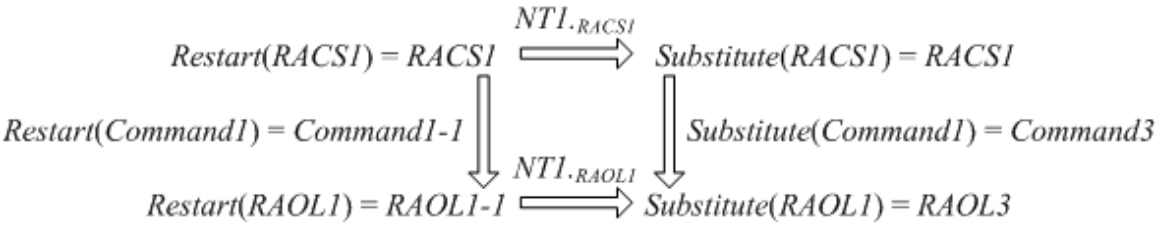


Figure 65: Natural Transformation from *Restart* to *Substitute* in RACG1

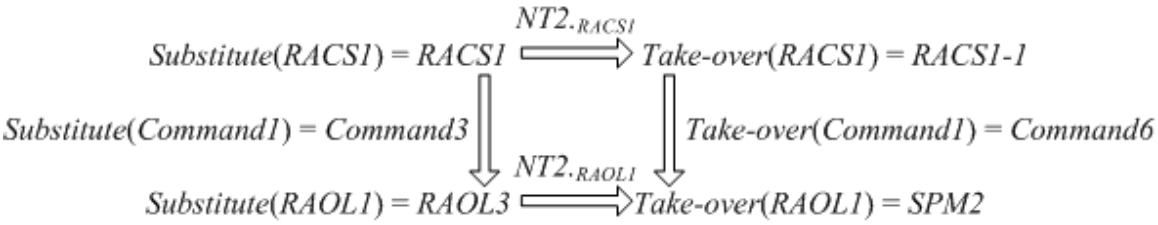


Figure 66: Natural Transformation from *Substitute* to *Take-over* in RACG1

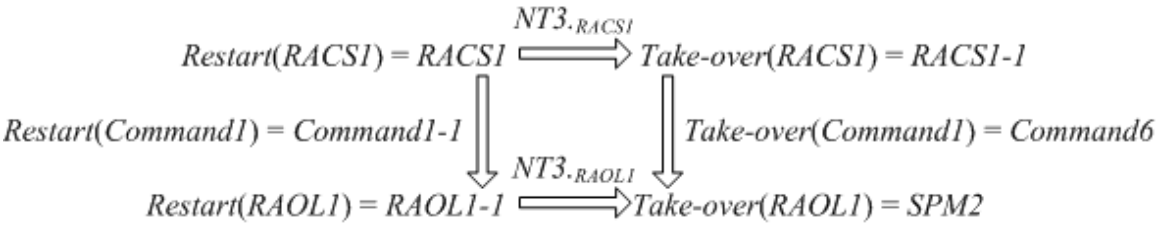


Figure 67: Natural Transformation from *Restart* to *Take-over* in RACG1

When both *RAOL1* and *RAOL2* are crashed at the same time, *RACS1* tries to restart them first. If neither of them can be restarted, *RACS1* will send a message to *RACGM1* for requesting the isomorphic objects (*RAOL3* and *RAOL4*) of *RAOL1* and *RAOL2*; otherwise, the remaining process is the same as the illustration above. When *RACGM1* cannot find *RAOL3* and *RAOL4*, *RACS1* will take over the responsibilities of *RAOL1* and *RAOL2*, working as a product object of them, such as *SPM1*; otherwise, the description above may indicate the remaining process. If a **RACG** consists of more than two *RACS*, the similar categorical representation can be generated as we explained previously.

In this scenario, we proposed three solutions (*restart*, *substitute* and *take-over*) for the fault-tolerance in case of the crashed RAOL, which cover all possible situations in terms of self-healing from the practical usage.

7.5 Scenario3: Crashed RACS

After *RACS1* is started by *RACGM1*, it begins to send its heartbeat messages to *RACGM1* every *t* ticks, while *RACGM1* is in the first state of its intelligent control loop, monitoring the status of *RACS1*. If *RACGM1* receives the heartbeat messages from *RACS1*, *NoChange* event keeps it in *Monitor* state; otherwise, *RACS1-Crashed* event is triggered and then *RACGM1* transits to *Analyze* state, while a time constraint variable (*TCvar3*) is

initialized to work as a local clock in terms of time constraints on each transition of the intelligent control loop. The value of *TCvar3* is *t0*, *t1*, *t2*, *t3*... where $t0 < t1 < t2 < t3$. After *RACGM1* enters *Analyze* state, it sends a *Restart* message to *RACS1* in *t0* ticks. If *RACS1* is recoverable, *NoAction* event occurs and *RACGM1* goes back to *Monitor* state, while the *TCvar3* is reset; otherwise, *RACGM1* transits to the *Plan* state triggered by *HasAction* event in *t7* ticks (see Figure 70).

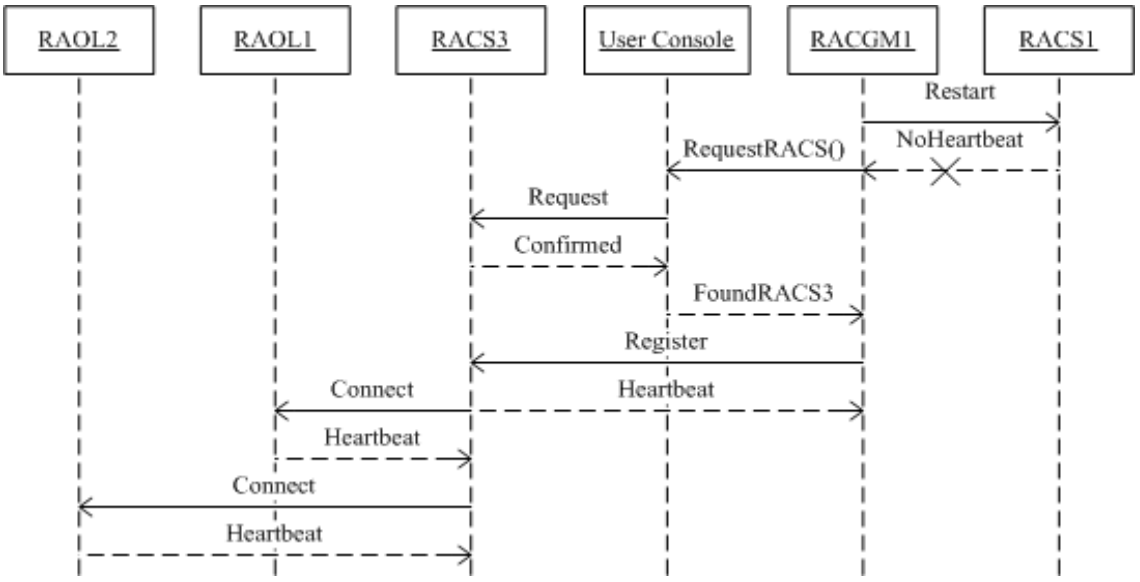


Figure 68: Substitution Work Flow in RAS1

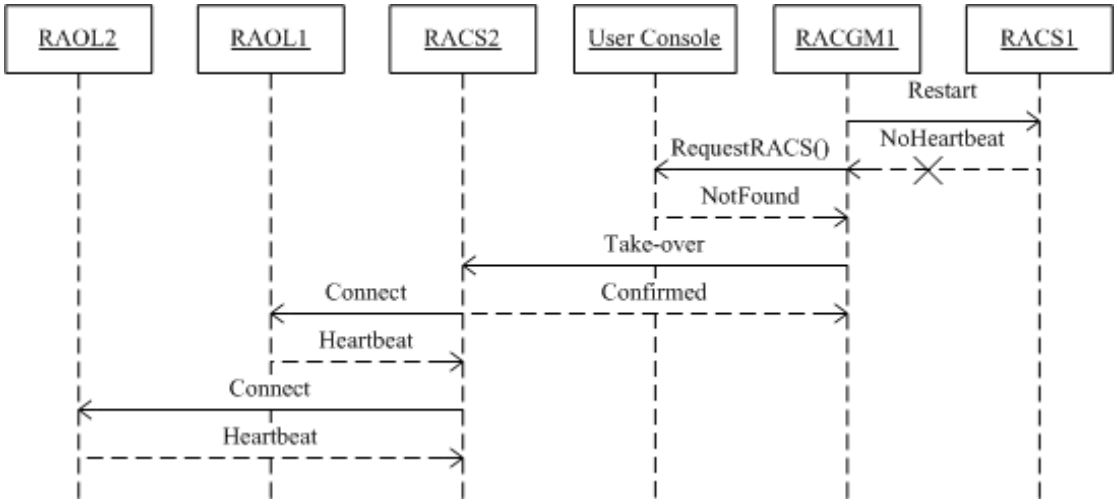


Figure 69: Take-Over Work Flow in RAS1

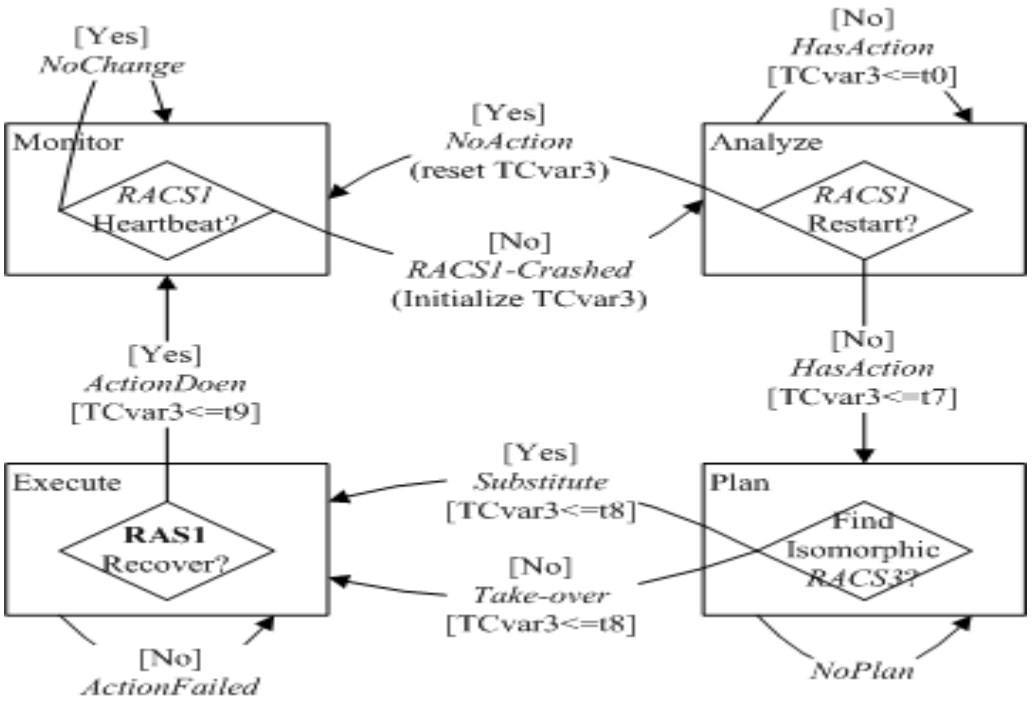


Figure 70: Intelligent Control Loop in RACGM1 for Self-Healing

When *RACGM1* is in *Plan* state, it sends the *RequestRACS1* message with the type information of *RACS1* to *User Console*, and the latter tries to replace *RACS1* by an available *RACS* that is isomorphic to *RACS1* in **RAS**, such as *RACS3*, since isomorphic objects behave in the same way. If *User Console* finds *RACS3* successfully, *RACGM1* chooses the plan *Substitute* and transits to *Execute* state triggered by *Substitute* event in *t8* ticks; otherwise, it selects the plan *Take-over* and enters *Execute* state triggered by *Take-over* event in *t8* ticks. In this plan, *RACS2* takes the responsibilities of *RACS1* and works as the product object of the original *RACS1* and *RACS2*, since they behave as a synchronous product machine.

When *RACGM1* is in *Execute* state and the plan *Substitute* is applicable, *RACGM1* sends a *register* message to *RACS3* and initializes it to the status of *RACS1* according to

the checkpoint made before. When the plan *take-over* is applicable, *RACGM1* sends a *take-over* message to *RACS2* and update it to the status of the synchronous product machine of *RACS1* as well as *RACS2* based on the checkpoint. After the plan execution, *RACGM1* validates the behavior of **RAS1** and its evolution (see Property 6.1.3) **RAS1'** according to their categorical specifications. If **RAS1'** has the equivalent behavior with **RAS1**, *ActionDone* event occurs and *RACGM1* transits to *Monitor* state in *t9* ticks; otherwise, *ActionFailed* event keeps it in *Execute* state for the user intervention from *User Console*.

7.6 Categorical Illustration of Scenario3

The actions in the substitution work flow and take-over work flow of RAS1 can be specified as the categories where objects are those actions (*Restart*, *RequestRACS*, *NotFound*, etc.), and morphisms are their preorder relation *before*. Each object (action) in those categories is a quadruple (see Property 6.2.28). For example, *Take-over* = (*RACGM1*, *NotFoundRACS*, *RequestSubstitution*, *RACS2*). Furthermore, the sequences of those actions can be specified as the categories where objects are those sequences (<*RequestRACS*, *NotFound*, *Take-over*>, <*Take-over*, *Confirmed*, *Connect*, *Heartbeat*>), and morphisms are *equivalence* relationship between those sequences (see Property 6.2.31).

The transitions in the intelligent control loop of RAS1 for self-healing can be specified as a category in which objects are those transitions (*HasAction, Substitute*, *ActionDone*, etc.), and morphisms are their preorder relations *before*. Each object

(transition) in a category is a triple (see Property 6.2.23). For example, *HasAction* = (*Analyze*, *NoHeartBeatAfterRestart*, *Plan*). Moreover, the sequences of the transitions can be specified as a category where objects are those sequences (<*HasAction*, *Substitute*, *ActionFailed*>, <*HasAction*, *NoPlan*, *Take-over*, *ActionDone*>), and morphisms are the *equivalence* relation between those sequences (see Property 6.2.26).

CAT-RAS <RAS1>
Objects: <RACS1, RACS2, RACS3, RACGM1>
Morphisms: <Work1(RACS1, RACS2), Work2(RACS2, RACS1), Work3(RACS1, RACS3),
Work4(RACS3, RACS1), Work5(RACS2, RACS3), Work6(RACS3, RACS2),
Report1(RACS1, RACGM1), Report2(RACS2, RACGM1), Report3(RACS3, RACGM1),
Order1(RACGM1, RACS1), Order2(RACGM1, RACS2), Order3(RACGM1, RACS3)>
Limit Object: <RACGM1>
Colimit Object: <RACGM1>
Product Objects: <SPM1(RACS1, RACS2), SPM2(RACS1, RACS3), SPM3(RACS2, RACS3)>
Coproduct Objects: <MQ1(RACS1, RACGM1), MQ2(RACS2, RACGM1), MQ3(RACS3, RACGM1)>
Pushout Objects: <NR3(Order1, Order2), NR4(Work4, Work6)>
Pullback Objects: <PR3(Report1, Report2), PR4(Work3, Work5)>
Slice Category: <(Report1, Report2, Report3), Work1, Work3, Work5>
Coslice *Category*: <(Order1, Order2, Order3), Work2, Work4, Work6>
Subcategories: <RACG1, RACG2, RACG3>
Product Categories: <Interact1(RACG1, RACG2), Interact2(RACG1, RACG3), Interact3(RACG2, RACG3)>
Functors: <Restart(RACGM1, RACGM1-1), Substitute(RACGM1, RACGM1-2),
Take-over(RACGM1, RACGM1-3)>
Natural Transformations: <NT1(Restart, Substitute), NT2(Substitute, Take-over), NT3(Restart, Take-over)>
Functor Category: <(Restart, Substitute, Take-over), NT1, NT2, NT3>
End CAT-RAS

Figure 71: Categorical Illustration of RAS1

Figure 71 depicts the categorical representation of RAS1 before *RACS1* is crashed. The category **RAS1** consists of objects *RACS1*, *RACS2, RACS3* as well as *RACGM1*. The bidirectional communications among those objects are morphisms to specify the working collaboration among *RACS1*, *RACS2, RACS3*, and the leadership from *RACGM1* to

RACS1, *RACS2* as well as *RACS3* (see Property 6.1.2). The SPM for the synchronous communication between two objects in **RAS1** can be represented by SPM1, SPM2 and SPM3 (see Property 6.2.6). Slice category models *Report* actions (Report1, Report2 and Report3) with their relations (Work1, Work3 and Work5) from *RACS1*, *RACS2* and *RACS3* to *RACGM1* (see Property 6.2.12); coslice category may interpret *Order* actions (Order1, Order2 and Order3) with their relations (Work2, Work4 and Work6) from *RACGM1* to *RACS1*, *RACS2* and *RACS3* (see Property 6.2.13).

Let **RAS1** be a subcategory (consisting of objects RACGM1, RACS1, RACS2 and the morphisms among them) of **RAS1-0** (a category consisting of all potential RAE for the self-healing in RAS1). If **RAS1** is recovered by restarting crashed *RACS1*, it evolves to **RAS1-1** having the same categorical structure as **RAS1** except for different initial status of *RACS1*. This evolution can be represented by the *Restart* functor (a structure-preserving mapping) from **RAS1-1** to **RAS1-0**. If **RAS1** is recovered by substituting *RACS1* with its isomorphic object *RACS3* (see Definition 3.1.3), it evolves to **RAS1-2** (consisting of objects RACGM1, RACS3, RACS2 and the morphisms among them in **RAS1-0**), which has the same categorical structure as **RAS1** but replacing *RACS1* with *RACS3*. The above is specified by the *Substitute* functor, a structure-preserving mapping (see Figure 72).

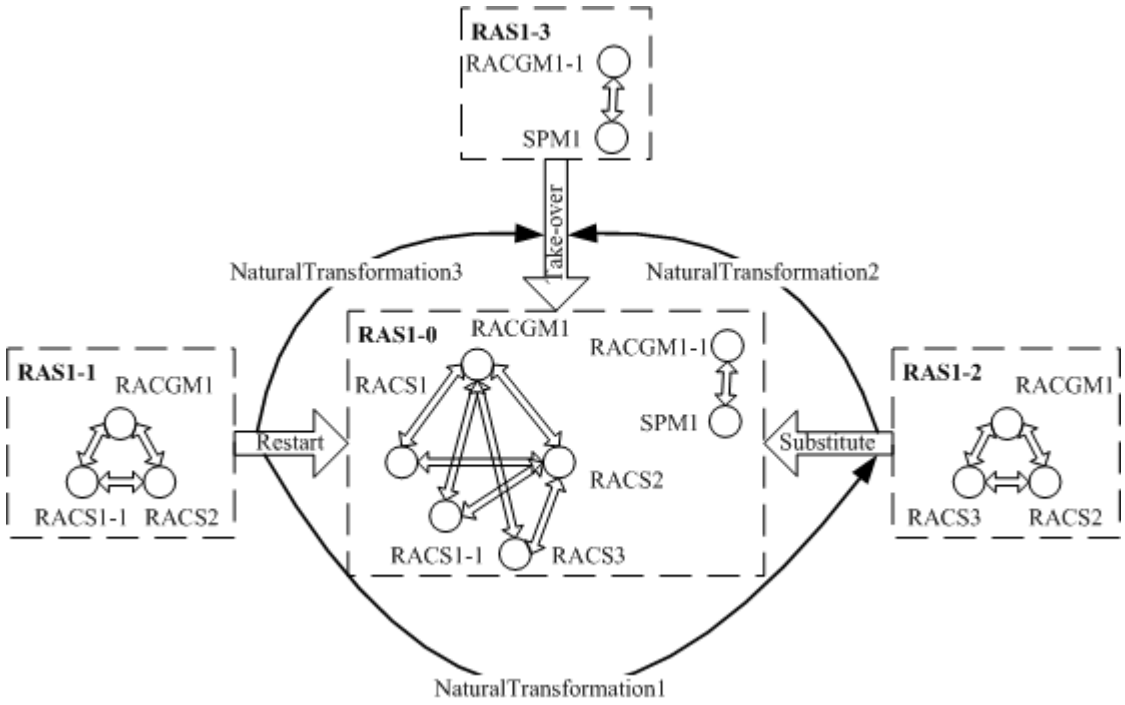


Figure 72: Evolution for Self-Healing in RAS1

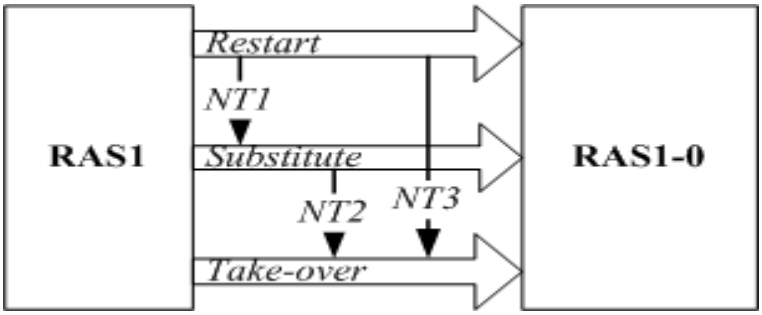


Figure 73: Natural Transformation for Self-Healing in RAS1

However, if **RAS1** is recovered by asking *RACS2* to take over the responsibilities of *RACS1*, it evolves to **RAS1-3** (consisting of objects RACGM1-1, SPM1 along with the morphisms between them in **RAS1-0**), which has the different categorical structure, but both of them have the equivalent social lives (see Property 6.2.33). The mapping among those evolutions *Restart*, *Substitute* and *Take-over* can be interpreted as their natural transformations (see Property 6.1.4). The functor category having those functors as its

objects and their natural transformations as morphisms illustrate all possible evolutions with their relations. In addition, the natural transformation3 is a composition of natural transformation1 and natural transformation2, which may be interpreted as the following: the result of the evolution *Restart -> Substitute -> Take-over* is equivalent to the evolution *Restart -> Take-over*. Figure 74, 75 and 76 illustrate those natural transformations and their composition respectively.

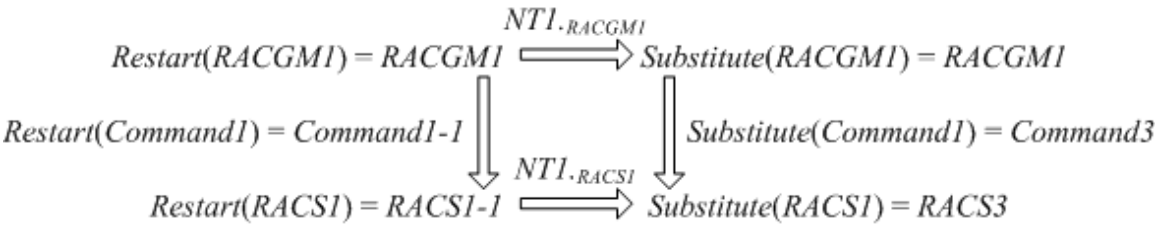


Figure 74: Natural Transformation from *Restart* to *Substitute* in RAS1

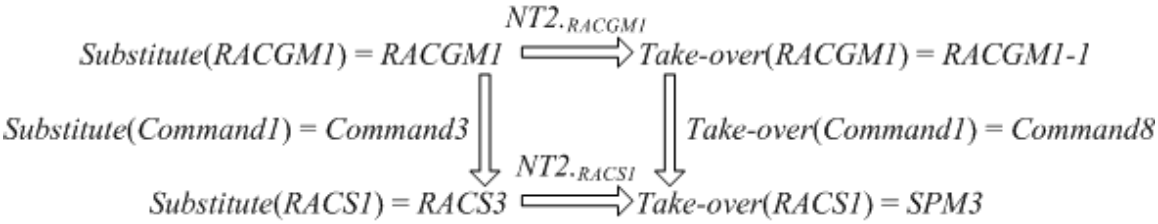


Figure 75: Natural Transformation from *Substitute* to *Take-over* in RAS1

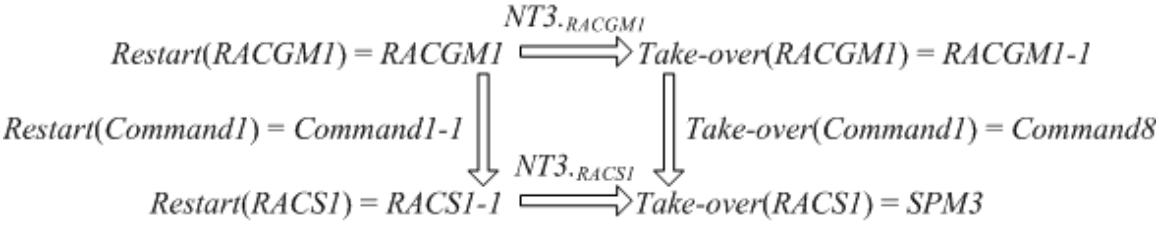


Figure 76: Natural Transformation from *Restart* to *Take-over* in RAS1

When both *RACS1* and *RACS2* are crashed at the same time, *RACGM1* tries to restart them first. If neither of them can be restarted, *RACGM1* will send a message to *User Console* for requesting isomorphic objects (*RACS3* and *RACS4*) of *RACS1* and *RACS2*;

otherwise, the remaining process is the same as the illustration above. When the *User Console* cannot find *RACS3* and *RACS4*, *RACGM1* will take over the responsibilities of *RACS1* and *RACS2*, working as a product object of them, such as *SPM1*; otherwise, the description above may indicate the remaining process. If a **RAS** consists of more than two *RACGM*, the similar categorical representation can be generated as we explained previously.

In this scenario, we proposed three solutions (*restart*, *substitute* and *take-over*) for the fault-tolerance in case of the crashed RACS, which cover all possible situations in terms of self-healing from the practical usage.

7.7 Categorical Specifications of Self-Healing

As we stated in Chapter 5 and Chapter 6, the self-* (autonomic) behavior of a RAE is modeled as sequences of transitions corresponding to execution paths derived from the matching labeled transition system, and the interactive behavior between the RAE are modeled by the sequences of external (observable) events interchanged between the interfaces of those RAE. The sequences of the transitions and external event sequences are modeled as the categories **TRANSITION** and **INTERACTION** respectively. For example, the behavior of the ICLM (Intelligent Control Loop Model) depicted in Chapter 5 is interpreted by the category **TRANSITION**(ICLM) = $<Seq_1, Seq_2, \ldots, Seq_n>$, where the objects are sequences of transitions, such as $Seq_1 = <Trans_{1\text{-}1}, Trans_{1\text{-}2}, \ldots, Trans_{1\text{-}m}>$ ($n, m \geq 1$), and the morphisms are *equivalence* relations between those sequences. Thus, **TRANSITION**(ICLM) is a category because identity morphisms exist and associativity

of morphisms holds. Moreover, every transition in a sequence is modeled by the triple (*state*, *event*, *state*) indicating the source state, trigger event, and destination state of the corresponding transition in the ICLM, such as $Trans_{1\text{-}1}$ = (*Monitor*, HasChange, *Analyze*).

Similarly, the interactive behavior between RAE can be interpreted by the category **INTERACTION**(*RAE*), where the objects are sequences consisting of actions which capture the interchanged external events and the communication parties (the sender of the external event which triggers the communication, the receiver of the expected outcome event), such as $Seq_1 = <Act_{1\text{-}1}, Act_{1\text{-}2}, \ldots, Act_{1\text{-}n}>$; the morphisms are *equivalence* relations between those sequences. In addition, every action in a sequence is specified as the quadruple (*sender*, *trigger-event*, *last-event*, *receiver*) stating the sender of *trigger-event*, the *trigger-event* triggering an action, the *last-event* outputted from the action, and the receiver of *trigger-event*.

After a RAS is formed, RACGM, RACS and RAOL start their intelligent control loops and monitor the heartbeat messages sent in a certain time interval by RACS, RAOL, and RAO respectively (see Figure 77), which status are carried by those heartbeat messages. After RACGM receives a task from the user console, it chooses an action sequence (*ActSeqRACGM*) from the category **INTERACTION**(*RACGM*) in terms of fulfilling the task and interact with RACS1 because of the action $ActRACGM_1$ in the action sequence *ActSeqRACGM*; then RACS1 selects an action sequence (*ActSeqRACS1*) from **INTERACTION**(*RACS1*) in order to perform $ActRACGM_1$ and communicates with RAC1 due to the action $ActRACS1_1$ in the action sequence *ActSeqRACS1*. Similarly,

RAOL1 picks an action sequence (*ActSeq-RAOL1*) from **INTERACTION**(*RAOL1*) to implement $ActRACS1_1$ and collaborates with the RAO1 for the action $ActRAOL1_1$ in *ActSeqRAOL1*; eventually, RAO1 chooses the action sequence (*ActSeqRAO1*) from the **INTERACTION**(*RAO1*) in terms of realizing $ActRAOL1_1$ and cooperates with RAO2 due to the action $ActRAO1_1$ in *ActSeqRAO1*.

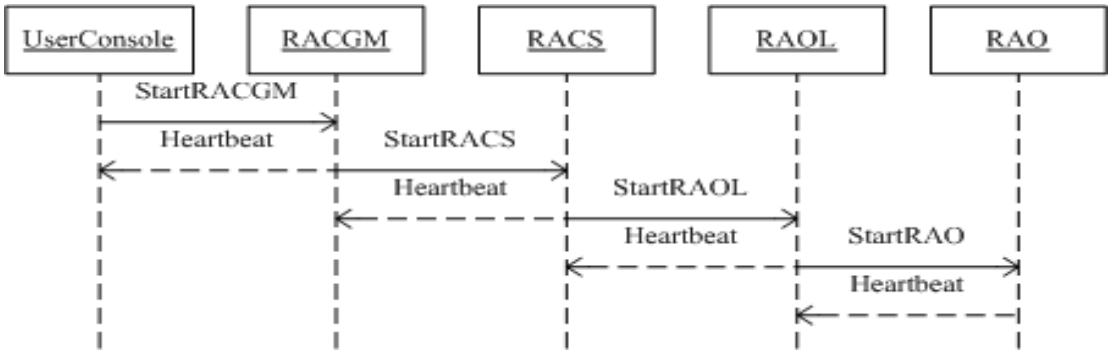


Figure 77: Work Flow of Formatting a RAS

Property 7.7.1 Fault-tolerance property in RAE: We state how fault-tolerance is applied to the internal behavior of RAE through the ICLM described in Chapter 5. If there is an exception during the transition $Trans_2$ = (*Analyze*, HasAction, *Plan*), RAE transits to *HandleException* state instead of *Plan* state triggered by *AnalyzeException* event as $Trans_{2e}$ = (*Analyze*, AnalyzeException, *HandleException*); after an exception is handled successfully, RAE rolls back to *Analyze* state triggered by *HandledAnalyze* event with the same status as before the exception as $Trans_{2h}$ = (*HandleException*, Handled-Analyze, *Analyze*). It proceeds to the $Trans_2$ as $TransSeq_1'$ = <$Trans_1$, $Trans_{2e}$, $Trans_{2h}$, $Trans_2$, …, $Trans_n$> that is isomorphic to $TransSeq_1$ = <$Trans_1$, $Trans_2$, …, $Trans_n$> (see Property 6.2.32). Thus, the category **TRANSITION**(*RAE*) including objects $TransSeq_1$,

$TransSeq_2$, …, $TransSeq_m$ is a full subcategory of the category **TRANSITION** (RAE') having objects $TransSeq_1$, $TransSeq_1'$, $TransSeq_2$, …, and both categories are equivalent (see Property 3.6.8). This means that two sequences exhibit the same internal behavior in RAE for performing an action and leads to the fault-tolerance property.

Property 7.7.2 Interactive behavior equivalence between RAE. If RAC1 cannot be started by a RACS due to an exception as $ActRAC1_{1e}$ = (*RACS1*, *StartRAC1*, *NoHeartbeat -RAC1*, *RACS1*), RACS1 tries to restart RAC1 later as $ActRAC1_{1h}$ = (*RACS1*, *Restart- RAC1*, *Heartbeat-RAC1*, *RACS1*) if the exception can be processed and then it continues to $ActRAC1_2$ as $ActSeqRAC1_1'$ = <$ActRAC1_{1e}$, $ActRAC1_{1h}$, $ActRAC1_2$, …, $ActRAC1_n$>, which is equivalent to $ActSeqRAC1_1$ = <$ActRAC1_1$, $ActRAC1_2$, …, $ActRAC1_n$> (see Property 7.7.3). Therefore, the category **INTERACTION**(*RAC1*) including the objects $ActSeqRAC1_1$, $ActSeqRAC1_2$, …, $ActSeqRAC1_m$ is a full subcategory of **INTERACTION** ($RAC1'$) with objects $ActSeqRAC1_1$, $ActSeqRAC1_1'$, $ActSeqRAC1_2$, …, $ActSeqRAC1_m$ and two categories are equivalent (see Property 3.6.8). It demonstrates that both sequences have the same interactive behavior between RACS1 and RAC1 in terms of executing an action so that the fault-tolerance property is achieved.

Property 7.7.3 Substitutability of RAE. RAE is equivalent to RAE' denoted as RAE ~ RAE' iff 1) they belong to the same type (RAO, RAOL, RAC, RACS, RACG or RACGM); 2) they have equivalent social lives SOCIAL(RAE) ~ SOCIAL(RAE'); 3) they have the equivalent internal structures when regarding them as two categories so that

CAT(*RAE*) ~ **CAT**(*RAE'*); and 4) they have equivalent internal and interactive behavior as **TRANSITION**(*RAE*) ~ **TRANSITION**(*RAE'*), **INTERACTION**(*RAE*) ~ **INTERACTION**(*RAE'*). If RAE ~ RAE', they can be substituted by each other.

7.8 Representation of Categorical Specification for Self-Healing

The figure below depicts an example of the representation for a categorical specification (in XML format) of the self-healing property we present earlier in this chapter, and more XML representation can be found in Appendix D.

```
<CATEGORY name = "Substitution-Work-Flow-for-Self-Healing">
    <OBJECT>
        <OBJECT name = "Restart" type = "Work-Flow-Action"/>
        <OBJECT name = "NoHeartbeat" type = "Work-Flow-Action"/>
        <OBJECT name = "RequestRAE" type = "Work-Flow-Action"/>
        <OBJECT name = "Request" type = "Work-Flow-Action"/>
        <OBJECT name = "Confirmed" type = "Work-Flow-Action"/>
        <OBJECT name = "Register" type = "Work-Flow-Action"/>
    </OBJECT>
    <MORPHISM>
        <MORPHISM name = "Before" type ="Preorder"/>
            <FROM-OBJECT name = "Restart" type = "Work-Flow-Action"/>
            <TO-OBJECT name = "NoHeartbeat" type = "Work-Flow-Action"/>
        <MORPHISM>
        <MORPHISM name = "Before" type ="Preorder"/>
            <FROM-OBJECT name = "NoHeartbeat"
                                type = "Work-Flow-Action"/>
            <TO-OBJECT name = "RequestRAE" type = "Work-Flow-Action"/>
        <MORPHISM>
        <MORPHISM name = "Before" type ="Preorder"/>
            <FROM-OBJECT name = "RequestRAE" type = "Work-Flow-Action"/>
            <TO-OBJECT name = "Request" type = "Work-Flow-Action"/>
        <MORPHISM>
        <MORPHISM name = "Before" type ="Preorder"/>
            <FROM-OBJECT name = "Request" type = "Work-Flow-Action"/>
            <TO-OBJECT name = "Confirmed" type = "Work-Flow-Action"/>
        <MORPHISM>
```

```
        <MORPHISM name = "Before" type ="Preorder"/>
            <FROM-OBJECT name = "Confirmed" type = "Work-Flow-Action"/>
            <TO-OBJECT name = "Register" type = "Work-Flow-Action"/>
        <MORPHISM>
    </MORPHISM>
</CATEGORY>
```

Figure 78: XML Specification of Category **Substitution-Flow-Self-Healing**

7.9 Summary

In this chapter, we presented a prototype design of self-healing property, prototype design of the categorical specification for self-healing and transformation from the categorical self- healing property to its XML specification.

We described three scenarios regarding the self-healing which are crashed RAO (Section 7.1), crashed RAOL (Section 7.3) as well as crashed RACS (Section 7.5) using intelligent control loops. We also presented the categorical illustration for those three scenarios in section 7.2, 7.4 & 7.6 respectively using functors, natural transformations and functor categories.

Finally, the categorical specifications for the self-healing related properties together with their XML representations were presented in Section 7.7 & 7.8, such as the fault-tolerance property, interactive behavior equivalence among RAE and substitutability of the RAE. We will introduce the categorical specifications of self-configuration property for RASF in next chapter.

Chapter 8: Categorical Specification of Self-Configuration

This chapter states the research activity 15, 16) and 21) in Figure 3, which are prototype design of self-configuration, prototype design of categorical specification for the self-configuration and transformation from the categorical self-configuration property to its XML specification. I have one publication in preparation for this chapter. In Chapter 5 & 6, we introduced the RAS model and MAS model with their architectures, composition rules, communication protocols and related categorical specifications. The configuration of the RAS and MAS must follow those rules and protocols during the life cycle of them, such as their formation and evolution because of the self-protection, self-optimization, or self-healing. Furthermore, those formation as well as evolution should be achieved in an autonomic way by giving RACGM tasks from User Console.

8.1 Forming a RAS

After receiving the task of forming a RAS from User Console, RACGM starts to create RACS and establish corresponding connections among them based on the composition rules and communication protocols specified by the index category **RAS-Formation** (see Property 6.1.1). Figure 79 depicts an example of forming categories **RAS1** and **RAS2** from their index category **RAS-Formation**. Figure 80 and 81 illustrate the detailed object as well as morphism mappings from **RAS1** and **RAS2** to the **RAS-Formation** respectively. The RACGM can be initialized and validated by an initialization manager in User Console.

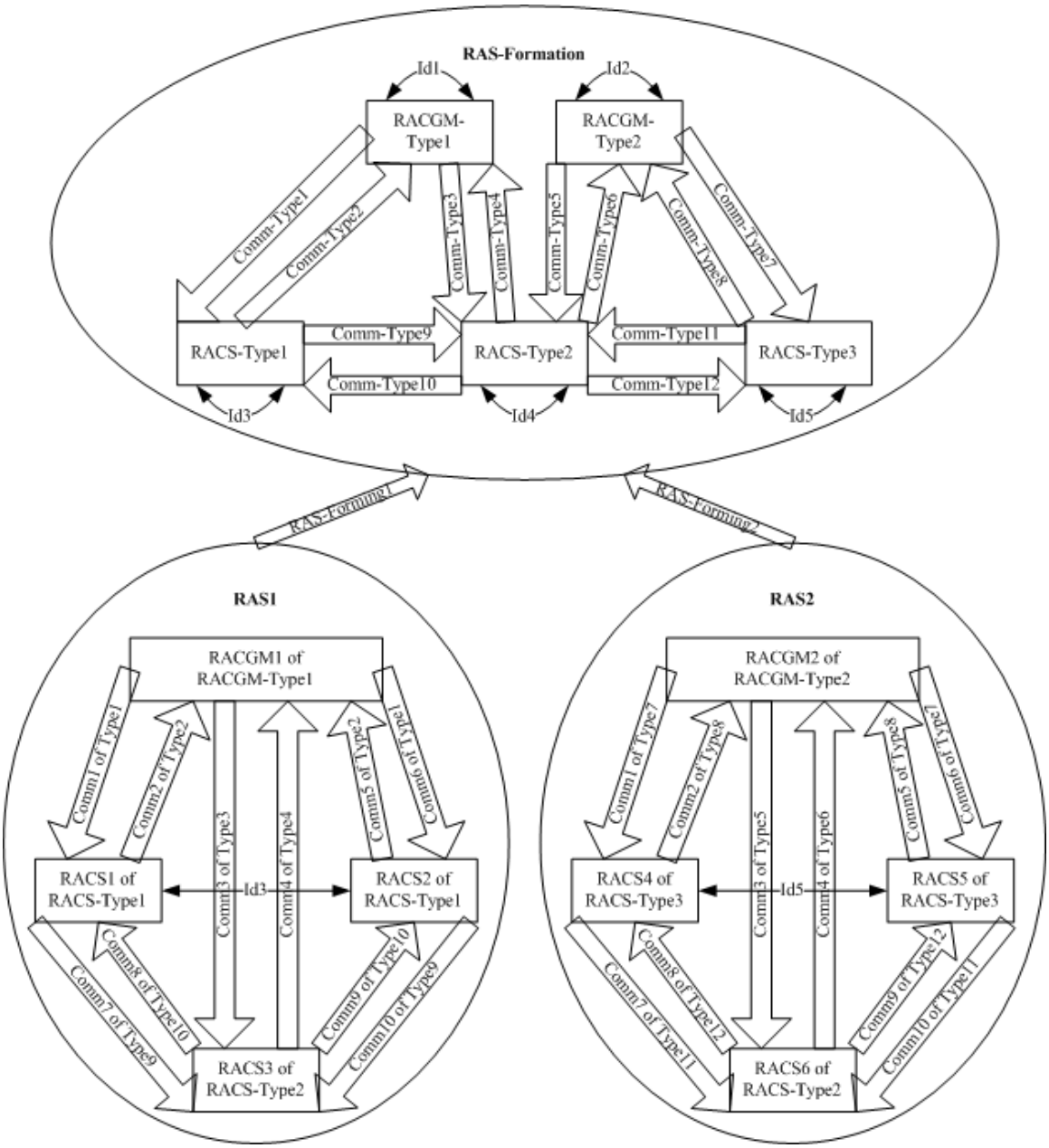


Figure 79: Example of Forming RAS from **RAS-Formation**

Object Mapping	Morphism Mapping (RAS-Formation <- RAS1)
RACGM1 ->RACGM-Type1	Comm1->Comm-Type1, Comm2->Comm-Type2
RACS1->RACS-Type1	Comm3->Comm-Type3, Comm4->Comm-Type4
RACS2->RACS-Type1	Comm5->Comm-Type2, Comm6->Comm-Type1

RACS3->RACS-Type2	Comm7->Comm-Type9, Comm8->Comm-Type10
	Comm9->Comm-Type10, Comm10->Comm-Type9
	RACS1–RACS2-> Identity-Mor(RACS-Type1)

Figure 80: Object and Morphism Mapping of Functor RAS-Forming1

Object Mapping	Morphism Mapping (RAS-Formation <- RAS2)
RACGM2->RACGM-Type2	Comm1->Comm-Type7, Comm2->Comm-Type8
RACS4->RACS-Type3	Comm3->Comm-Type5, Comm4->Comm-Type6
RACS5->RACS-Type3	Comm5->Comm-Type8, Comm6->Comm-Type7
RACS6->RACS-Type2	Comm7->Comm-Type11, Comm8->Comm-Type12
	Comm9->Comm-Type12, Comm10->Comm-Type11
	RACS4–RACS5-> Identity-Mor(RACS-Type3)

Figure 81: Object and Morphism Mapping of Functor RAS-Forming2

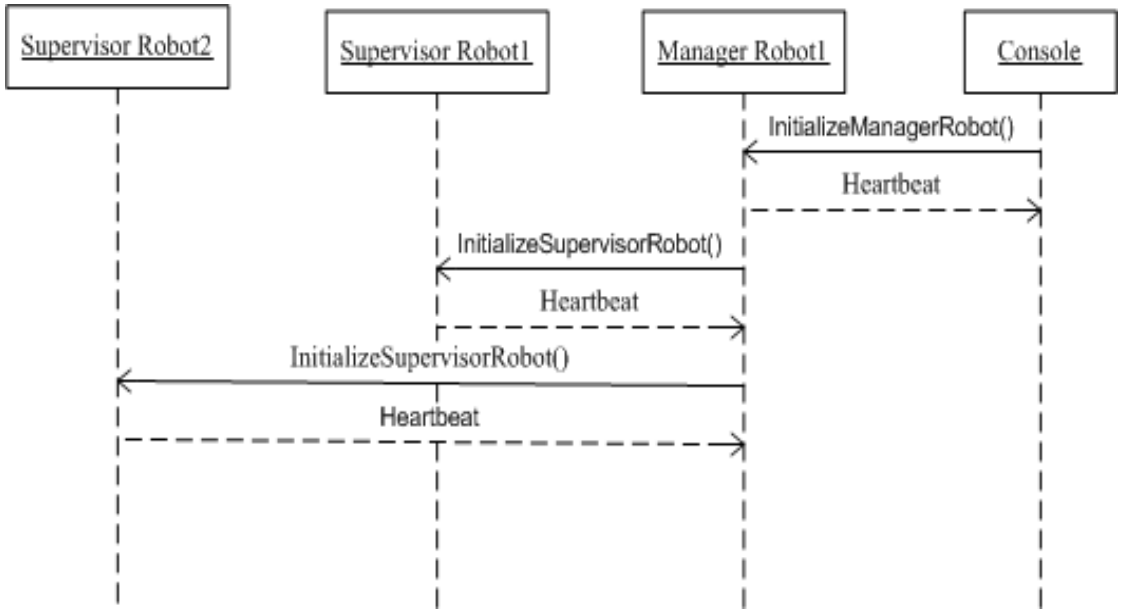


Figure 82: Formation Work Flow in RAS1

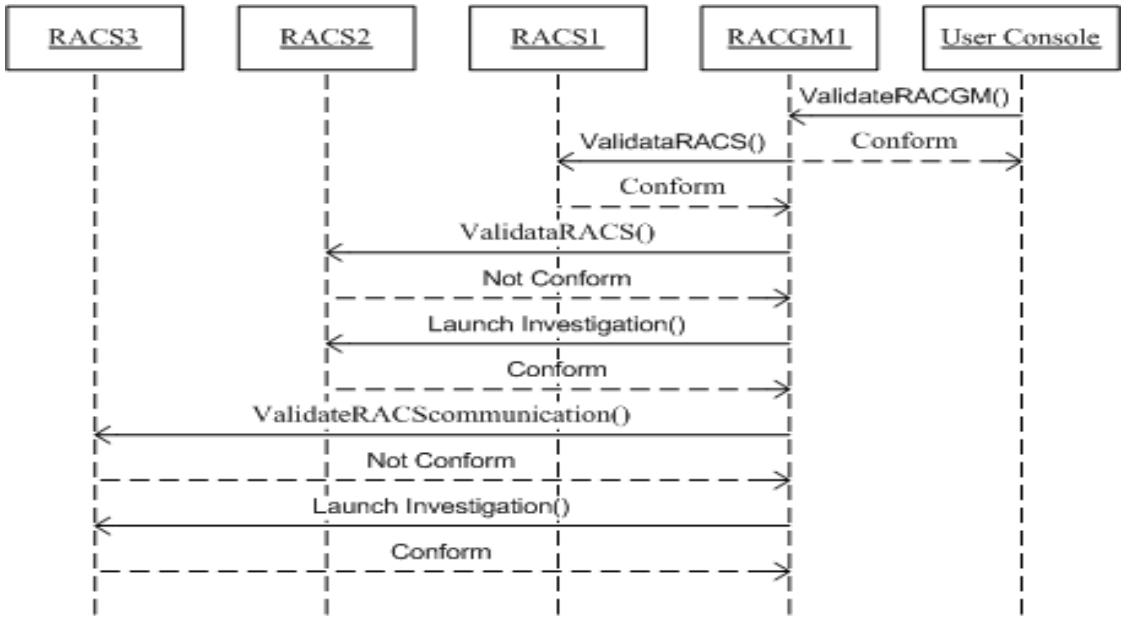


Figure 83: Self-Configuration Work Flow in RAS1

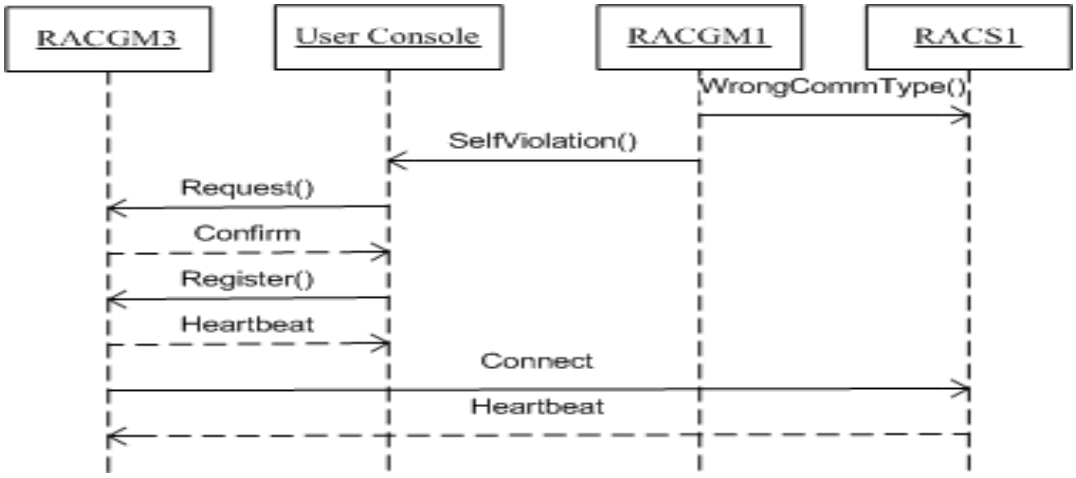


Figure 84: Self-Configuration Work Flow of Substitution in RAS1

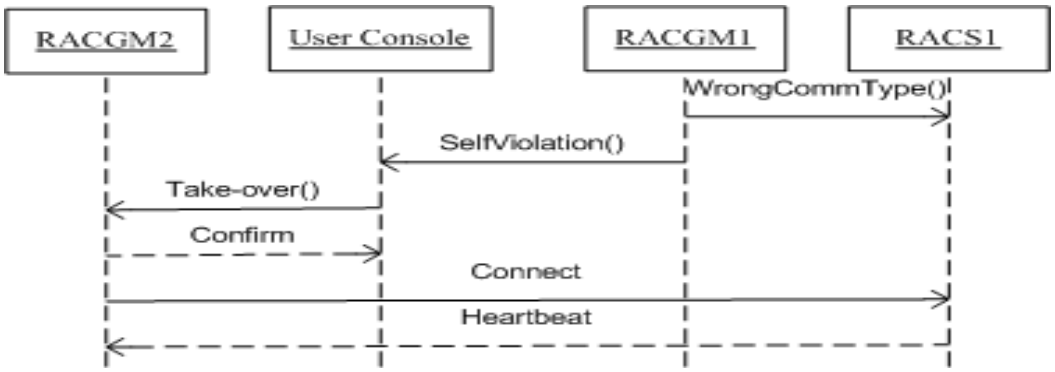


Figure 85: Self-Configuration Work Flow of Take-over in RAS1

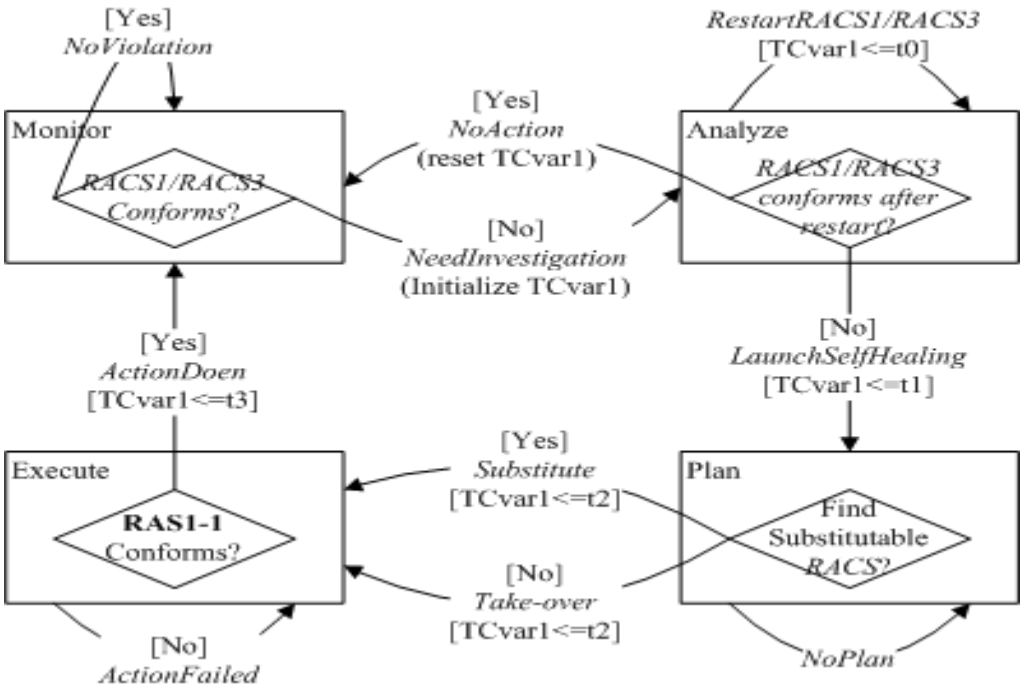


Figure 86: Intelligent Control Loop in RACGM1 for Self-Configuration

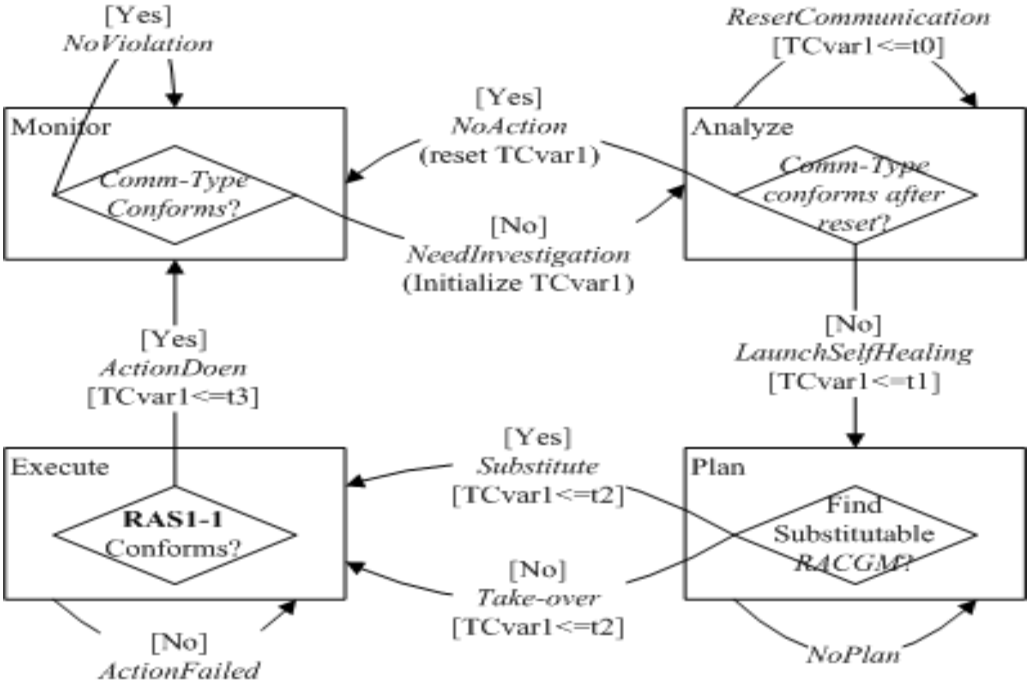


Figure 87: ICL in RACGM1 for Communication Self-Configuration

After *RACGM1* initializes its *RACS* according to the requirements from the *User Console* and the capabilities of those *RACS*, it validates the configuration of those *RACS*

against their types every *t* ticks (a tick is an abstraction of one time unit under a global clock in **RAS1**), while *RACGM1* is in the first state of its intelligent control loop (monitoring). If the configuration of those *RACS* conforms to their types, composition rules and communication protocols (see Property 8.7.4), *NoViolation* event keeps *RACGM1* in *Monitor* state; otherwise, *NeedInvestigation* event is triggered and *RACGM1* transits to *Analyze* state, while a time constraint variable (*TCvar1*) is initialized to work as a local clock in terms of time constraints on each transition of the intelligent control loop. The value of *TCvar1* is *t0*, *t1*, *t2*, *t3*... where $t0 < t1 < t2 < t3$.

After *RACGM1* enters *Analyze* state, 1) it sends a *Restart* message to *RACS1* in *t0* ticks where the violation is caused by incorrect RACS type or incorrect communication type from *RACS1* to *RACGM1*. If *RACS1* conforms to its type and communication type, *NoAction* event occurs and *RACGM1* goes back to *Monitor* state, while the *TCvar1* is reset; otherwise, *RACGM1* transits to *Plan* state triggered by *LaunchSelfHealing* event in *t1* ticks. 2) If the violation is caused by incorrect communication type from other RACS (*RACS3*) to *RACS1*, *RACGM1* sends a *Restart* message to *RACS3*. If the communication conforms to its type, *NoAction* event occurs and *RACGM1* goes back to *Monitor* state, while the *TCvar1* is reset; otherwise, *RACGM1* transits to *Plan* state triggered by the *LaunchSelfHealing* event within *t1* ticks. 3) If the violation is caused by the incorrect communication type from *RACGM1* to *RACS1*, *RACGM1* resets that communication. If it conforms to the correct one specified in the index category **RAS-Formation** (see Property 8.7.4), *NoAction* event occurs and *RACGM1* goes back to the *Monitor* state,

while the *TCvar1* is reset; otherwise, *RACGM1* transits to *Plan* state triggered by the *LaunchSelfHealing* event in *t1* ticks.

When *RACGM1* is in *Plan* state, it chooses either *Substitute* plan or *Take-over* plan, based on the availability of substitutable RACS for *RACS1* (scenario 1 in the paragraph above) or for *RACS3* (scenario 2). *RACGM1* transits to *Execute* state triggered by the *Substitute* event or *Take-over* event respectively in *t2* ticks. For scenario 3, *RACGM1* sends a *selfViolation* message to *User Console*, and the latter chooses either *Substitute* plan or *Take-over* plan based on the availability of substitutable RACGM for *RACGM1*. It transits to *Execute* state triggered by *Substitute* or *Take-over* event in *t2* ticks.

When *RACGM1* is in *Execute* state and *Substitute* plan is applicable, it sends a *register* message to the substitutable RACS of *RACS1* (scenario 1) or *RACS3* (scenario 2) and initialize it to the status of *RACS1* or *RACS3* according to the checkpoint made before. When the *take-over* plan is applicable, *RACGM1* sends a *take-over* message to *RACS2* (scenario 1 or scenario 2) and update it to the status of the synchronous product machine of *RACS1* and *RACS2*, or *RACS3* and *RACS2* based on the checkpoint. After the plan execution, *RACGM1* validates the configuration of **RAS1'**, an evolution of **RAS1** (see Property 6.1.3) against its index category **RAS-Formation** according to their categorical specifications. If that configuration conforms to the index category (see Property 8.7.4), *ActionDone* event occurs and *RACGM1* transits to the *Monitor* state in *t3* ticks; otherwise, *ActionFailed* event keeps it in *Execute* state for the user intervention from *User Console*. For scenario 3, *RACGM1* is substituted by *RACGM3* or taken over

by *RACGM2*.

8.2 Categorical Illustration of Forming a RAS

The actions in the formation work flow, self-configuration work flow, substitution work flow and take-over work flow of RAS1 can be specified as the categories where objects are those actions (*InitializeRACS*, *ValidateRACS*, *ValidateRACcommunication*, etc.), and morphisms are their preorder relationship *before*. Each object (action) in those categories is a quadruple (see Property 6.2.28). For example, *LaunchInvestigation* = (*RACGM1*, *NotConfrom-RACS*, *InvestigateRACS*, *RACS1*); the sequences of those actions can be specified as the categories in which objects are those sequences (<*InitializeRACGM*, Heartbeat, *InitializeRACS*, *Heartbeat*>, <*ValidateRACGM*, *Conform*, *ValidateRACS*, *NotConform*>), and morphisms are the *equivalence* relationship between those sequences (see Property 6.2.31).

The transitions in the intelligent control loop of RACGM1 for self-configuration can be specified as a category in which objects are those transitions (*NoViolation, NeedInvestigation*, *RestartRACS*, *NoAction*, etc.), and morphisms are their preorder relations *before*. Each object (transition) in that category is a triple (see Property 6.2.23). For example, *NeedInvestigation* = (*Monitor*, *NotConform-RACS*, *Analyze*); the sequences of those transitions can be specified as a category in which objects are those sequences (<*NoViolation*, *NeedInvestigation*, *RestartRACS1*, *NoAction*>, <*RestartRACS1*, *LaunchSelfHealing*, *Substitute*, *ActionDone*>), and morphisms are *equivalence* relations between those sequences (see Property 6.2.26).

Let **RAS1** be a subcategory (consisting of the objects RACGM1, RACS1, RACS2, RACS3 and the morphisms among them) of **RAS1-0** (a category consisting of all the potential RAE for the self-configuration in RAS1). If **RAS1** is conformed to the index category **RAS-Formation** by restarting the violated *RACS1* or *RACS3*, it will evolve to **RAS1-1** (consisting of the objects RACGM1, RACS1 or RACS1-1, RACS2, RACS3 or RACS3-1 and the morphisms among them in **RAS1-0**), which has the same configuration and categorical structure as **RAS1** except for the different initial status of *RACS1* or *RACS3*. This evolution is specified by a *Restart* functor (a structure-preserving mapping) from the **RAS1-1** to **RAS1-0**. If **RAS1** is conformed to **RAS-Formation** by substituting the *RACS1* or *RACS3* with their isomorphic objects *RACS7* or *RACS9* (see Definition 3.1.3), it will evolve to **RAS1-2** (consisting of objects RACGM1, RACS1 or RACS7, RACS2, RACS3 or RACS9 and the morphisms among them in **RAS1-0**) that has the same configuration and categorical structure as **RAS1** but replacing *RACS1* or *RACS3* with *RACS7* or *RACS9*. The above is specified by the *Substitute* functor, a structure-preserving mapping. If **RAS1** is conformed to the **RAS-Formation** by asking *RACS2* to take over the responsibilities of *RACS1* or *RACS3*, it will evolve to **RAS1-3** (consisting of objects RACGM1-1, SPM, RACS1 or RACS3 and the morphisms among them in the **RAS1-0**), which has different categorical structure, but both of them have the equivalent configuration (see Property 8.7.5 and the figure below).

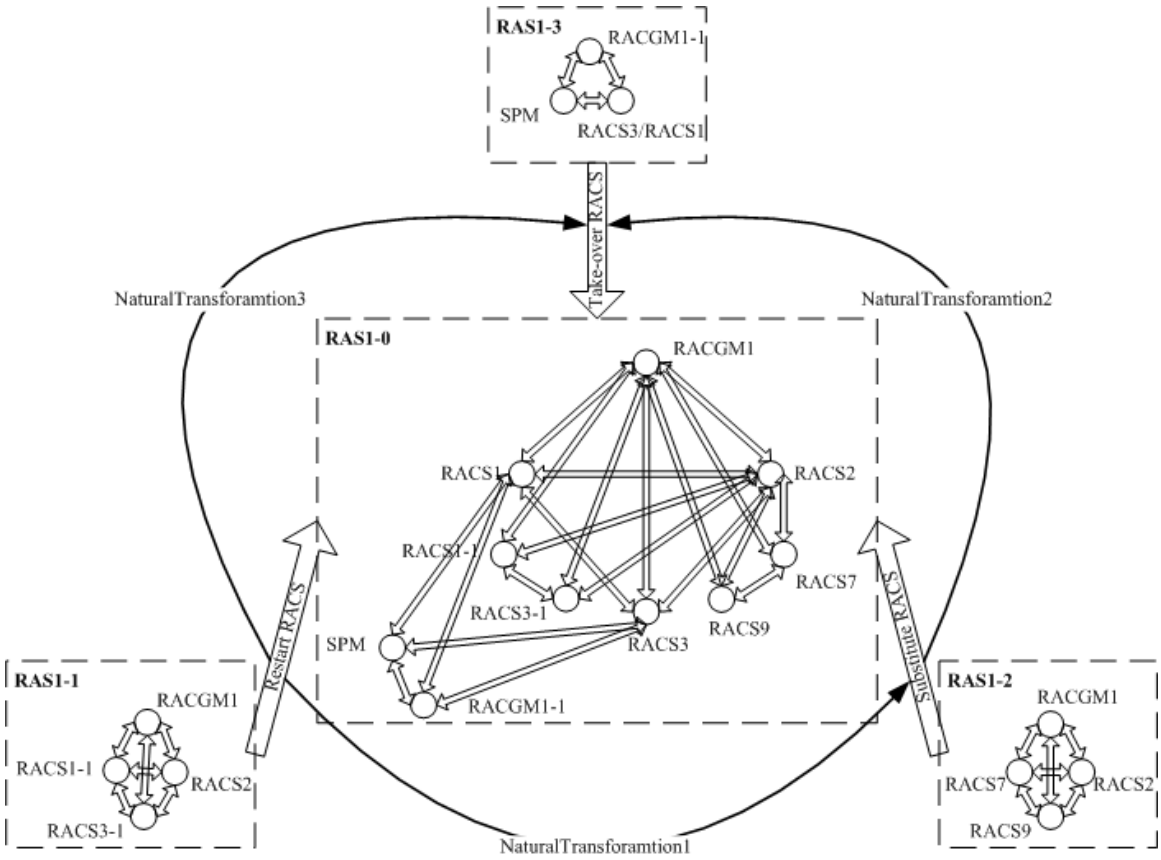


Figure 88: Evolution for Self-Configuration in RAS1

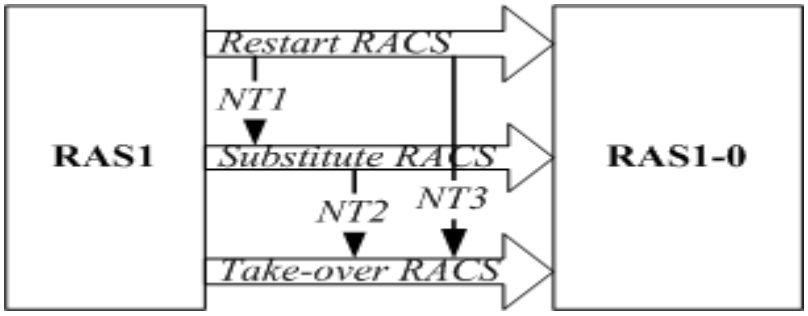


Figure 89: Natural Transformation for Self-Configuration in RAS1

The mapping among those evolutions *RestartRACS*, *SubstituteRACS* and *Take-over-RACS* of the **RAS1** can be interpreted as natural transformations (see Property 6.1.4). The functor category having those functors as objects and their natural transformations as morphisms illustrate all possible evolutions with their relations. In addition, the natural

transformation3 is a composition of natural transformation1 and natural transformation2, which may be interpreted as the following: the result of the evolution *RestartRACS -> SubstituteRACS -> Take-over-RACS* is equivalent to the evolution *RestartRACS -> Take-over-RACS*. Figure 90, 91 and 92 illustrate those natural transformations and their composition respectively.

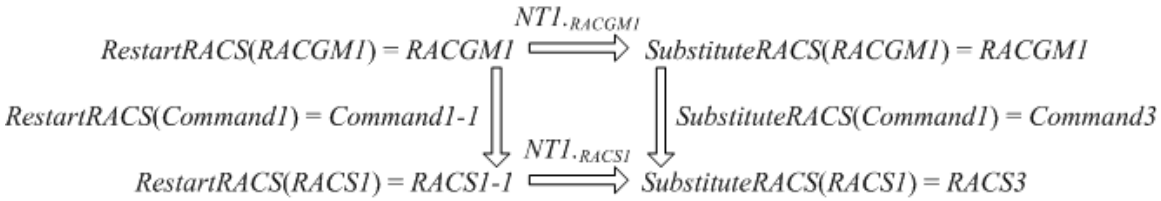


Figure 90: Natural Transformation *RestartRACS -> SubstituteRACS* in RAS1

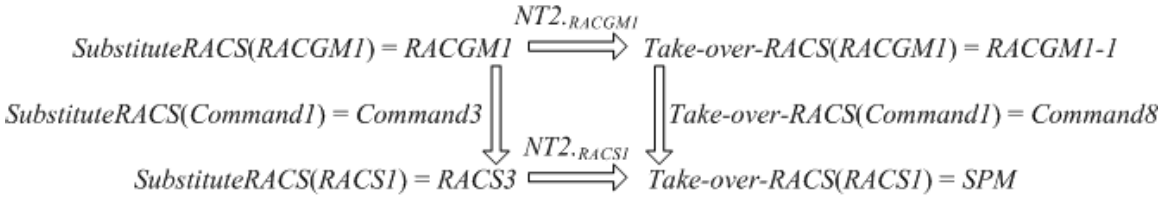


Figure 91: Natural Transformation *SubstituteRACS -> Take-over-RACS* in RAS1

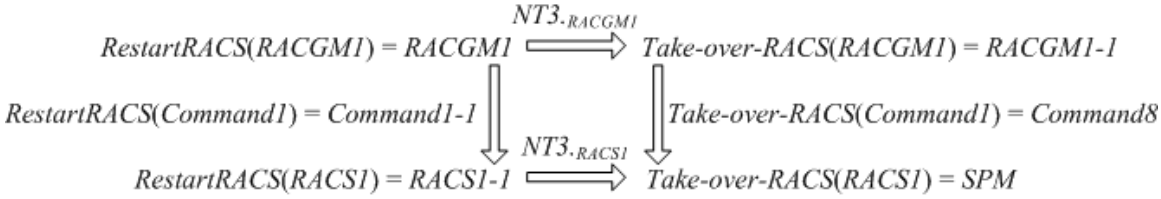


Figure 92: Natural Transformation *RestartRACS -> Take-over-RACS* in RAS1

Let **RAS1** be a subcategory (consisting of the objects RACGM1, RACS1, RACS2, RACS3 and the morphisms among them) of **RAS1-0'** (a category consisting of all the potential RAE for the communication self-configuration in RAS1). If **RAS1** is conformed to the **RAS-Formation** by restarting the communication from *RACGM1* to *RACS1* or restarting *RACGM1*, it evolves to **RAS1-4** (consisting of objects RACGM1-1, RACS1,

RACS2, RACS3 and the morphisms among them in the **RAS1-0'**), which has the same configuration and categorical structure as **RAS1** except for the different initial status of *RACGM1*. This evolution is specified by a *RestartRACGM* functor (a structure-preserving mapping) from **RAS1-4** to **RAS1-0'**. If **RAS1** is conformed to the **RAS-Formation** by substituting *RACGM1* with its isomorphic objects *RACGM3* (see Definition 3.1.3), it evolves to **RAS1-5** (consisting of objects RACGM3, RACS1, RACS2, RACS3 and the morphisms among them in the **RAS1-0'**), which has the same configuration and categorical structure as the **RAS1** but replacing the *RACGM1* with *RACGM3*. The above is specified by a *SubstituteRACGM* functor, a structure-preserving mapping. If **RAS1** is conformed to **RAS-Formation** by asking *RACGM2* to take over the responsibilities of *RACGM1*, it evolves to **RAS1-6** (consisting of objects SPM, RACS1, RACS2, RACS3 and the morphisms among them in **RAS1-0'**), which has different categorical structure, but both of them have the equivalent configuration (see Property 8.7.5 and Figure 93).

The mapping among those evolutions *RestartRACGM*, *SubstituteRACGM* and *Take-over-RACGM* of the **RAS1** is interpreted as natural transformations (see Property 6.1.4). The functor category having those functors as objects and their natural transformations as morphisms illustrate all possible evolutions with their relations. In addition, the natural transformation6 is a composition of natural transformation4 and natural transformation5, which may be interpreted as the following: the result of the evolution *RestartRACGM -> SubstituteRACGM -> Take-over-RACGM* is equivalent to the evolution *RestartRACGM -> Take-over-RACGM*. Figure 95, 96 and 97 illustrate those natural transformations and

their composition respectively.

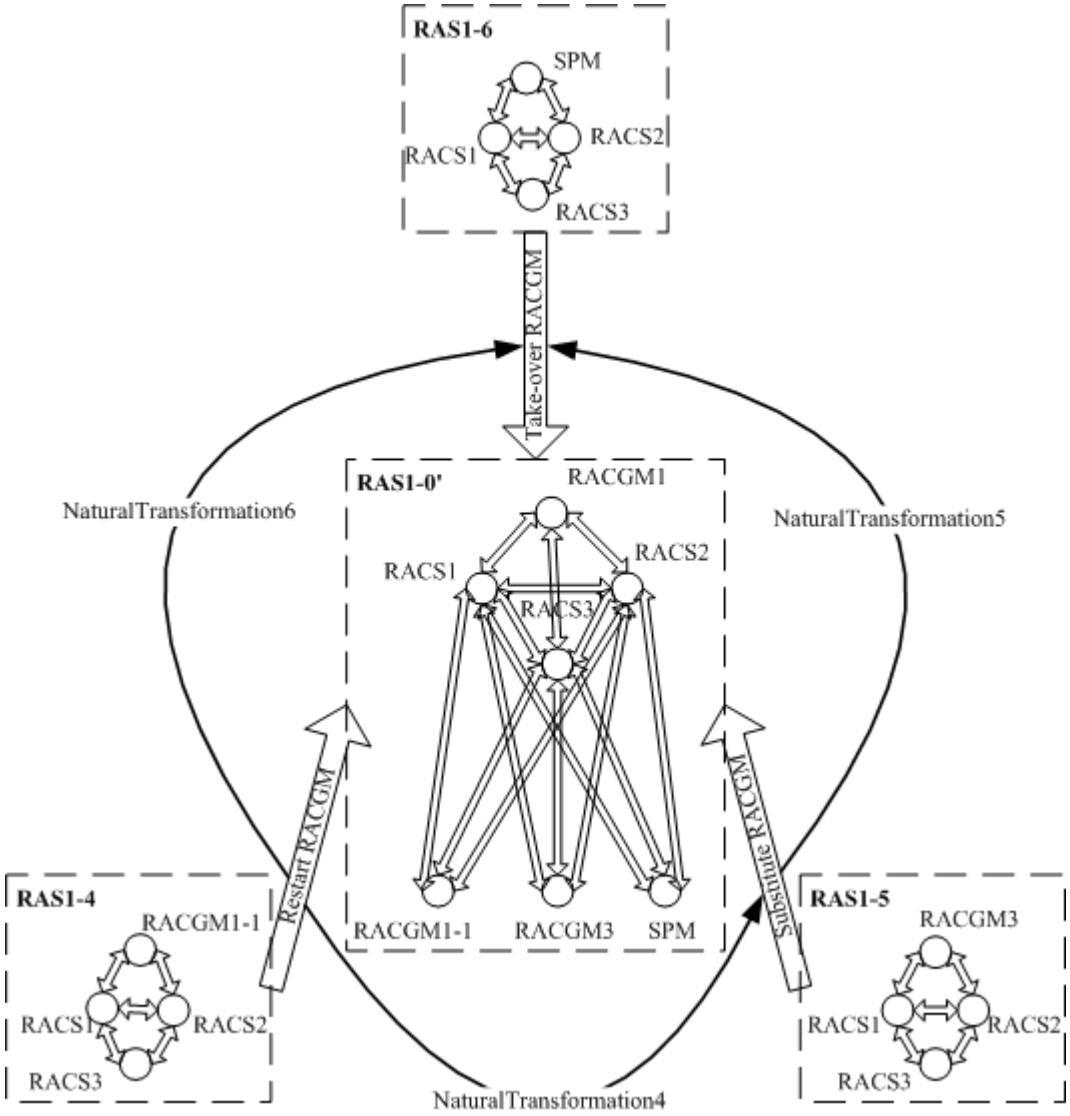


Figure 93: Evolution for Communication Self-Configuration in RAS1

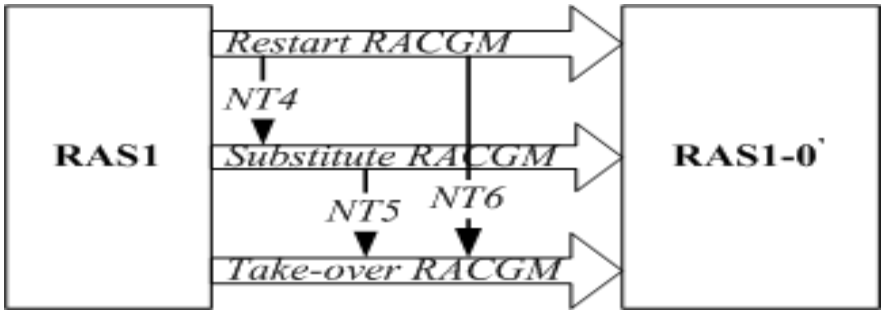


Figure 94: Natural Transformation for Communication Self-Configuration in RAS1

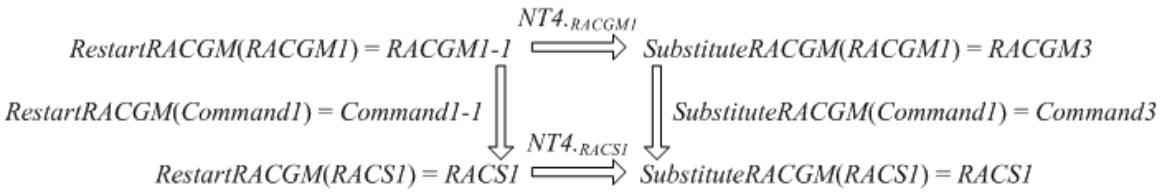


Figure 95: Natural Transformation *RestartRACGM -> SubstituteRACGM* in RAS1

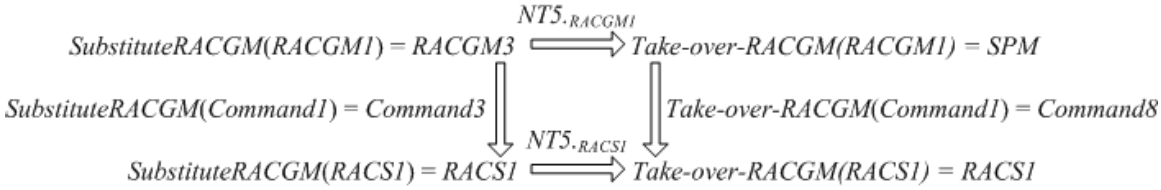


Figure 96: Natural Transformation *SubstituteRACGM->Take-over-RACGM* in RAS1

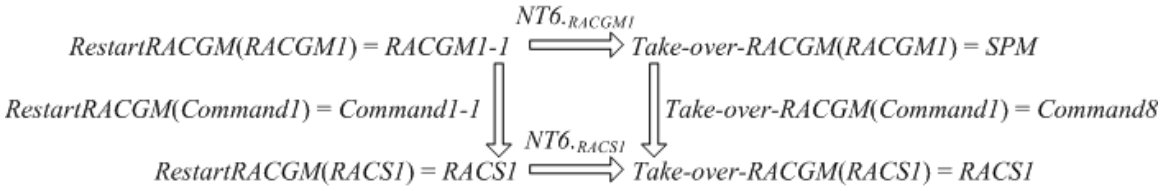


Figure 97: Natural Transformation *RestartRACGM -> Take-over-RACGM* in RAS1

When both *RACGM1* and *RACGM2* cannot conform to **RAS-Formation** at the same time, *User Console* tries to restart them first. If neither of them can conform to **RAS-Formation** after being restarted, *User Console* will substitute *RACGM1* and *RACGM2* with their isomorphic objects *RACGM3* and *RACGM4*; otherwise, the remaining process is the same as the illustration above. When the *User Console* cannot find *RACGM3* or *RACGM4*, it will send an *action-required* message to end users; otherwise, the description above may indicate the remaining process. If a **RAS** consists of more than two *RACGM*, the similar categorical representation can be generated as we explained.

8.3 Forming a RACG

After receiving the task of forming a RACG from RACGM, RACS starts to create RAOL

and establish corresponding connections among them based on the composition rules and communication protocols specified by index category **RACG-Formation** (see Property 6.1.1). Figure 98 illustrates an example of forming the categories **RACG1** and **RACG2** from their index category **RACG-Formation**. Figure 99 and 100 describe the detailed object as well as morphism mappings from **RACG1** and **RACG2** to **RACG-Formation** respectively.

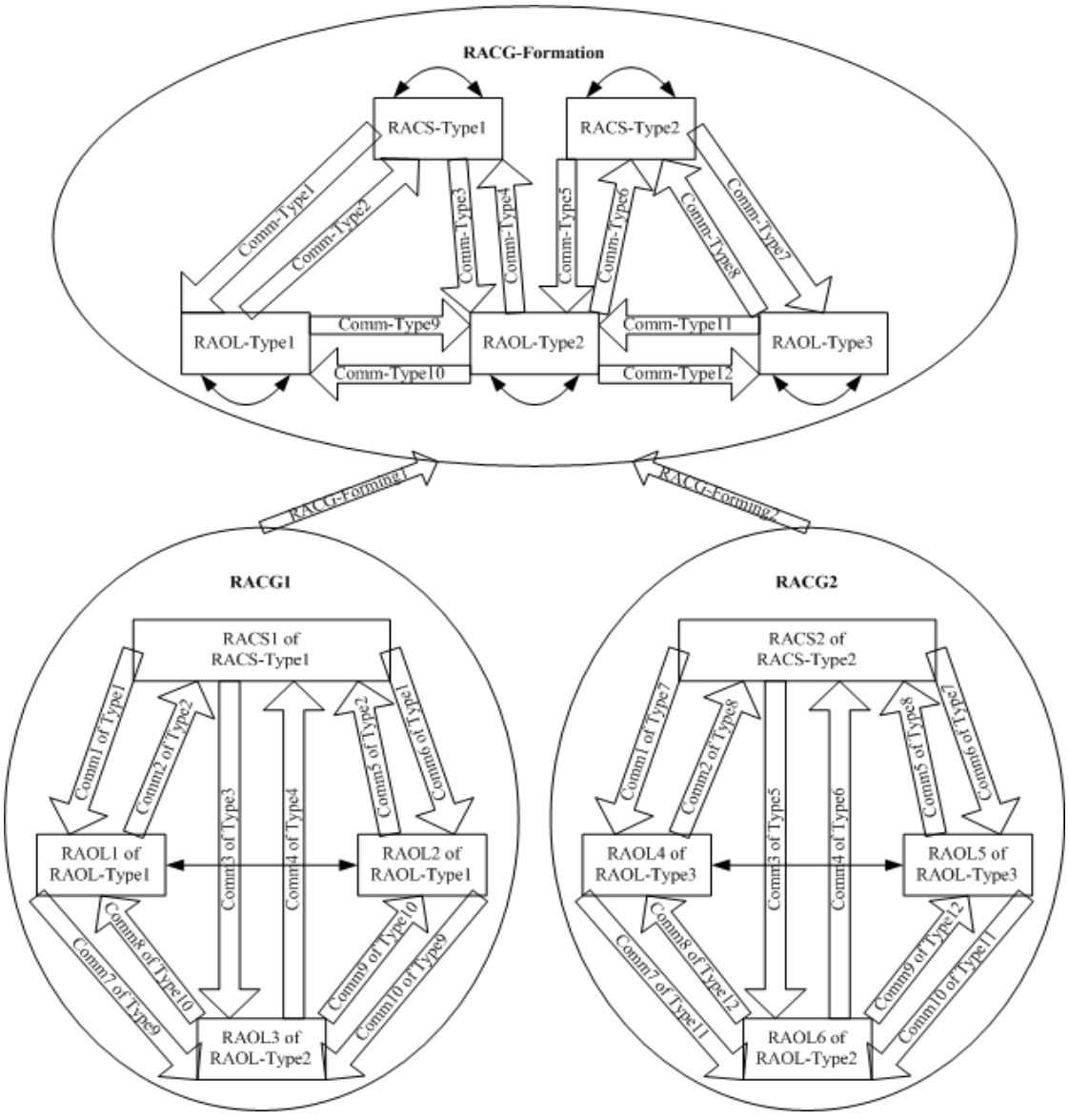


Figure 98: Example of Forming RACG from **RACG-Formation**

Object Mapping	Morphism Mapping (RACG-Formation <- RACG1)
RACS1->RACS-Type1	Comm1->Comm-Type1, Comm2->Comm-Type2
RAOL1->RAOL-Type1	Comm3->Comm-Type3, Comm4->Comm-Type4
RAOL2->RAOL-Type1	Comm5->Comm-Type2, Comm6->Comm--Type1
RAOL3->RAOL-Type2	Comm7->Comm-Type9, Comm8->Comm-Type10
	Comm9->Comm-Type10, Comm10->Comm-Type9
	RAOL1–RAOL2-> Identity-Mor(RAOL-Type1)

Figure 99: Object and Morphism Mapping of Functor RACG-Forming1

Object Mapping	Morphism Mapping (RACG-Formation <- RACG2)
RACS2->RACS-Type2	Comm1->Comm1-Type7, Comm2->Comm-Type8
RAOL4->RAOL-Type3	Comm3->Comm-Type5, Comm4->Comm-Type6
RAOL5->RAOL-Type3	Comm5->Comm-Type8, Comm6->Comm-Type7
RAOL6->RAOL-Type2	Comm7->Comm-Type11, Comm8->Comm-Type12
	Comm9->Comm-Type12, Comm10->Comm-Type11
	RAOL4–RAOL5-> Identity-Mor(RAOL-Type3)

Figure 100: Object and Morphism Mapping of Functor RACG-Forming2

After *RACS1* initializes its *RAOL* according to the requirements from the *RACGM* and the capabilities of those *RAOL*, it validates the configuration of those *RAOL* against their types every *t* ticks (a tick is an abstraction of one time unit under a global clock in the **RACG1**), while *RACS1* is in the first state of its intelligent control loop (monitoring).

If the configuration of those *RAOL* conforms to their types, composition rules as well as communication protocols (see Property 8.7.4), *NoViolation* event keeps *RACS1* in *Monitor* state; otherwise, *NeedInvestigation* event is triggered and *RACS1* transits to *Analyze* state, while a time constraint variable (*TCvar2*) is initialized to work as a local clock in terms of time constraints on each transition of the intelligent control loop. The value of *TCvar2* is *t0*, *t1*, *t2*, *t3*... where $t0 < t1 < t2 < t3$.

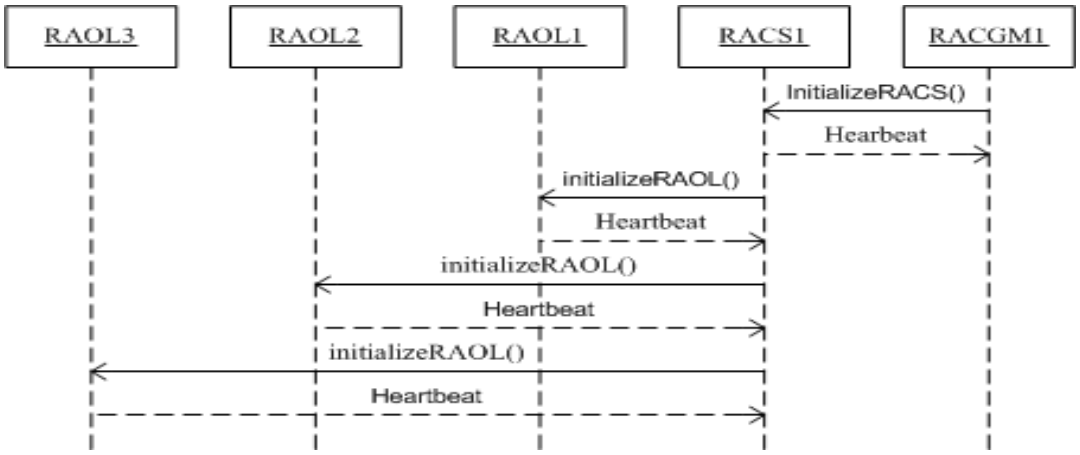


Figure 101: RACG Formation Work Flow

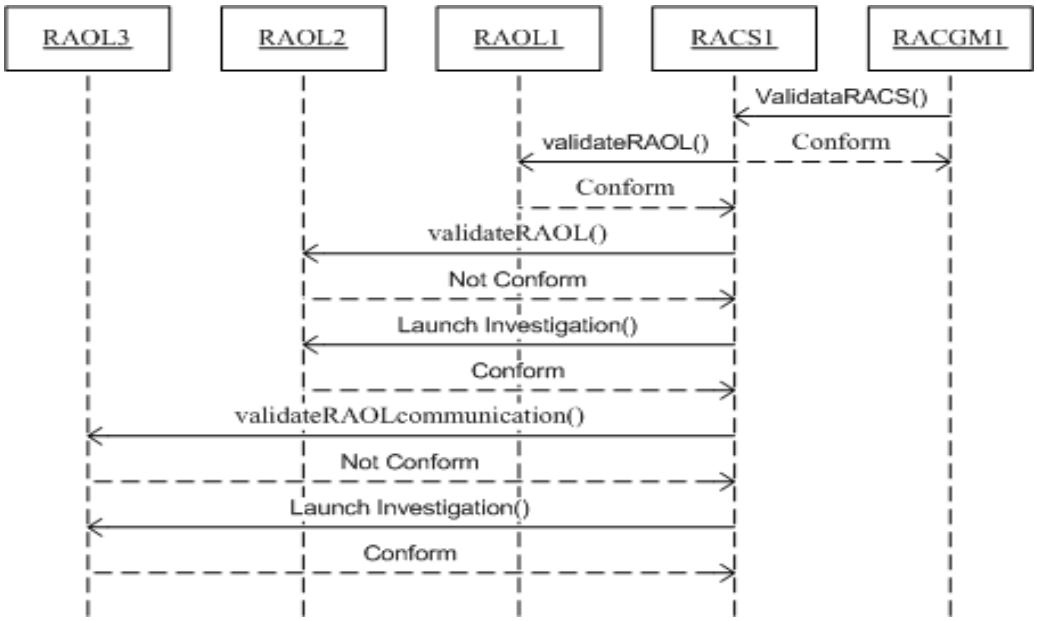


Figure 102: RACG Self-Configuration Work Flow

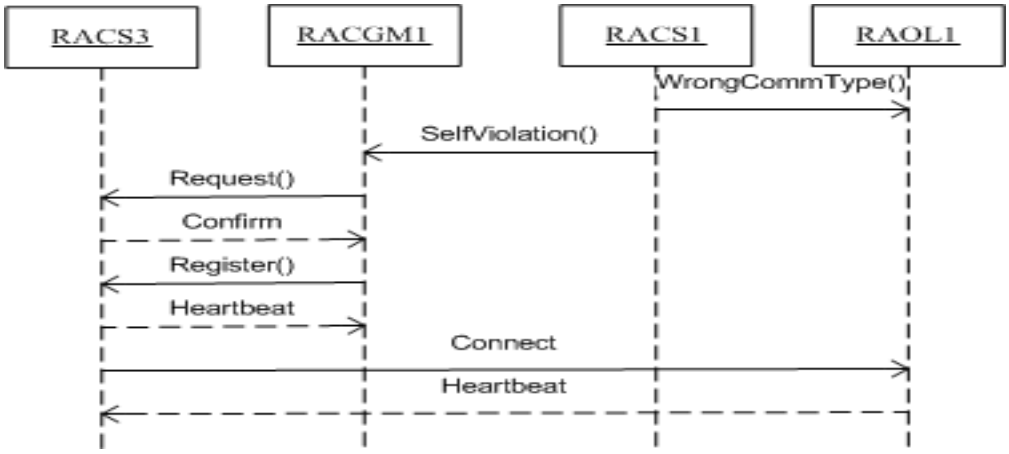


Figure 103: RACG Self-Configuration Work Flow by Substitution

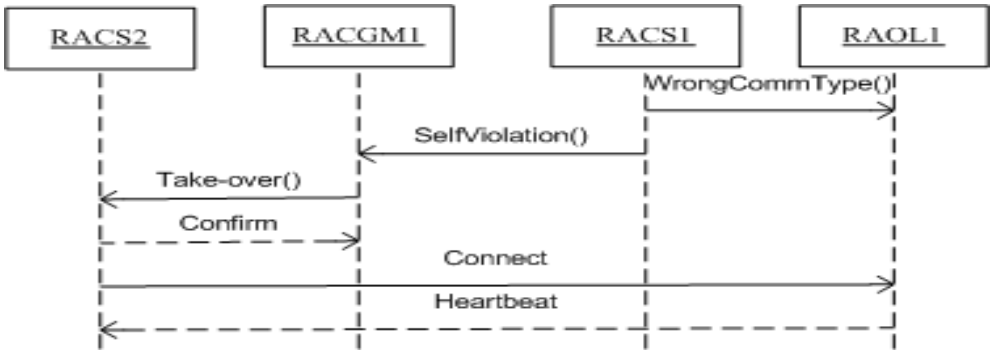


Figure 104: RACG Self-Configuration Work Flow by Take-over

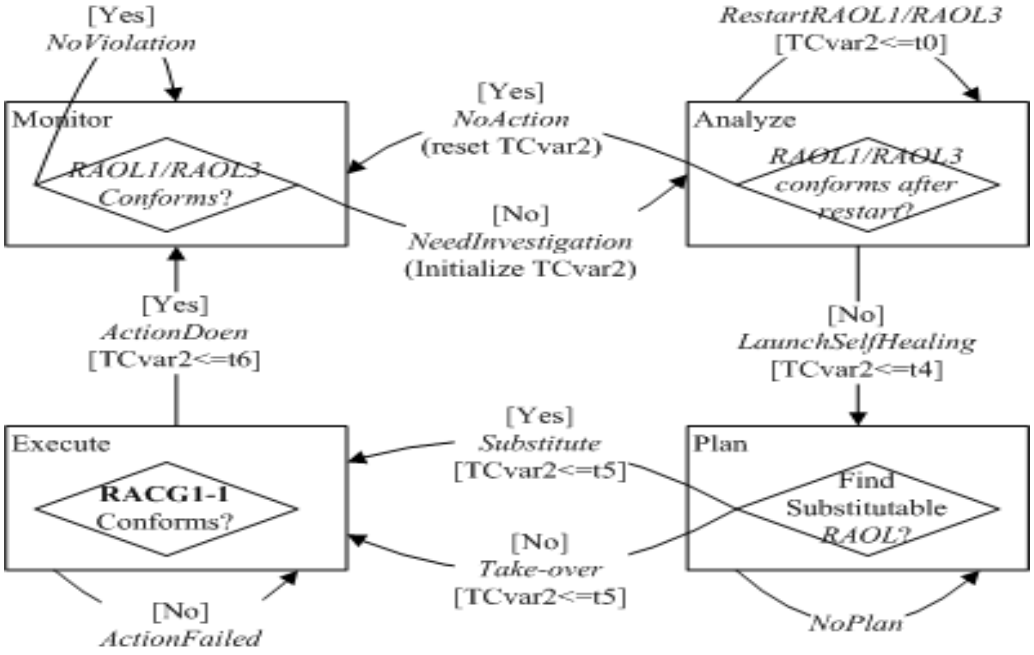


Figure 105: Intelligent Control Loop in RACS1 for Self-Configuration

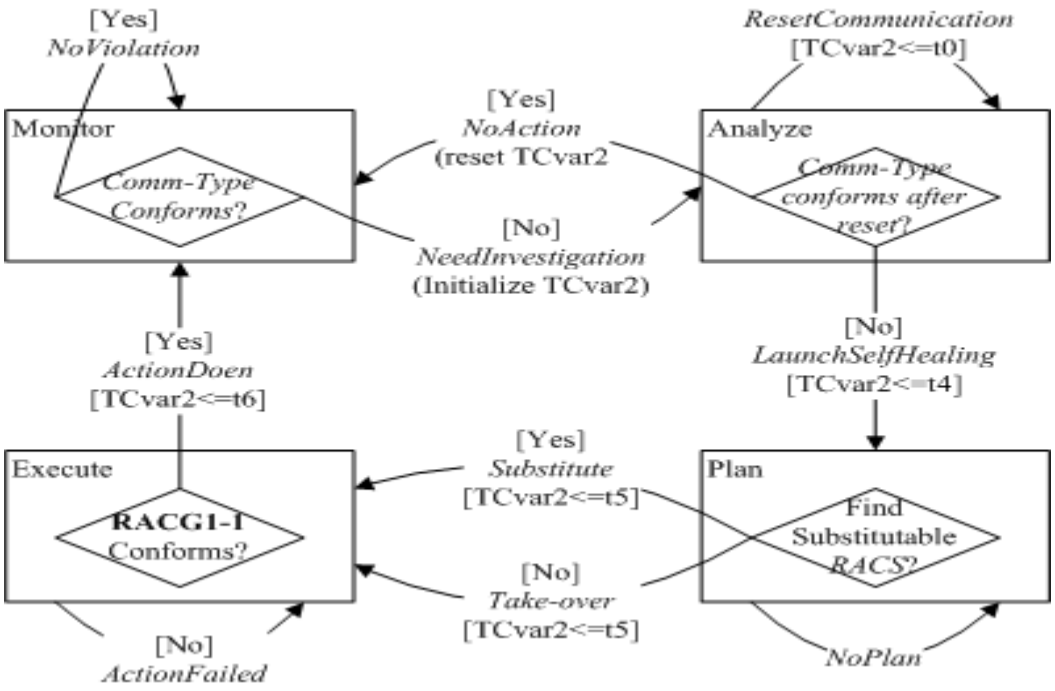


Figure 106: ICL in RACS1 for Communication Self-Configuration

After *RACS1* enters *Analyze* state, 1) it sends a *Restart* message to *RAOL1* in *t0* ticks where the violation is caused by the incorrect RAOL type or incorrect communication type from *RAOL1* to *RACS1*. If *RAOL1* conforms to its type and communication type, *NoAction* event occurs and *RACS1* goes back to *Monitor* state, while the *TCvar2* is reset; otherwise, *RACS1* transits to *Plan* state triggered by *LaunchSelfHealing* event in *t4* ticks. 2) If the violation is caused by incorrect communication type from other RAOL (*RAOL3*) to *RAOL1*, *RACS1* sends a *Restart* message to *RAOL3*. If the communication conforms to its type, *NoAction* event occurs and *RACS1* goes back to *Monitor* state, while the *TCvar2* is reset; otherwise, *RACS1* transits to *Plan* state triggered by the *LaunchSelfHealing* event within *t4* ticks. 3) If the violation is caused by the incorrect communication type from *RACS1* to *RAOL1*, *RACS1* resets that communication. If it conforms to the correct one

specified in the index category **RACG-Formation** (see Property 8.7.4), *NoAction* event occurs and *RACS1* goes back to the *Monitor* state, while the *TCvar2* is reset; otherwise, *RACS1* transits to *Plan* state triggered by *LaunchSelfHealing* event in *t4* ticks.

When *RACS1* is in *Plan* state, it chooses either *Substitute* plan or *Take-over* plan, based on the availability of substitutable RAOL for *RAOL1* (scenario 1 in the paragraph above) or for *RAOL3* (scenario 2). *RACS1* transits to *Execute* state triggered by the *Substitute* event or *Take-over* event respectively in *t5* ticks. For scenario 3, *RACS1* sends a *selfViolation* message to *RACGM1*, and the latter chooses either *Substitute* plan or *Take-over* plan based on the availability of substitutable RACS for *RACS1*. It transits to *Execute* state triggered by *Substitute* or *Take-over* event in *t5* ticks.

When *RACS1* is in *Execute* state and *Substitute* plan is applicable, it sends a *register* message to the substitutable RAOL of *RAOL1* (scenario 1) or *RAOL3* (scenario 2) and initialize it to the status of *RAOL1* or *RAOL3* according to the checkpoint made before. When the *take-over* plan is applicable, *RACS1* sends a *take-over* message to *RAOL2* (scenario 1 or scenario 2) and update it to the status of the synchronous product machine of *RAOL1* and *RAOL2*, or *RAOL3* and *RAOL2* according to the checkpoint. After the plan execution, *RACS1* validates the configuration of **RACG1'**, an evolution of **RACG1** (see Property 6.1.3) against the index category **RACG-Formation** based on their categorical specifications. If that configuration conforms to the index category (see Property 8.7.4), *ActionDone* event occurs and then *RACS1* transits to the *Monitor* state within *t6* ticks; otherwise, *ActionFailed* event keeps it in *Execute* state for *RACGM1*'s intervention. For

scenario 3, *RACS1* is substituted by *RACS3* or taken over by *RACS2*.

8.4 Categorical Illustration of Forming a RACG

The actions in the formation work flow, self-configuration work flow, substitution work flow and take-over work flow of RACG1 can be specified as the categories where objects are the actions (*InitializeRAOL*, *ValidateRAOL*, *ValidateRAOLcommunication*, etc.), and morphisms are their preorder relationship *before*. Each object (action) in those categories is a quadruple (see Property 6.2.28). For example, *LaunchInvestigation* = (*RACS1*, *NotConfrom-RAOL*, *InvestigateRAOL*, *RAOL1*), and the sequences of those actions can be specified as the categories in which objects are those sequences (<*InitializeRACS*, Heartbeat, *InitializeRAOL*, *Heartbeat*>, <*ValidateRACS*, *Conform*, *ValidateRAOL*, *NotConform*>), and morphisms are the *equivalence* relationship between those sequences (see Property 6.2.31).

The transitions in the intelligent control loop of RACS1 for self-configuration can be specified as the category in which objects are those transitions (*NoViolation, NeedInvestigation*, *RestartRAOL*, *NoAction*, etc.), and morphisms are their preorder relations *before*. Each object (transition) in that category is a triple (see Property 6.2.23). For example, *NeedInvestigation* = (*Monitor*, *NotConform-RAOL*, *Analyze*); the sequences of those transitions can be specified as a category in which objects are those sequences (<*NoViolation*, *NeedInvestigation*, *RestartRAOL1*, *NoAction*>, <*RestartRAOL1*, *LaunchSelfHealing*, *Substitute*, *ActionDone*>), and morphisms are *equivalence* relations between those sequences (see Property 6.2.26).

Let **RACG1** be a subcategory (consisting of the objects RACS1, RAOL1, RAOL2, RAOL3 and the morphisms among them) of **RACG1-0** (a category consisting of all the potential RAE for the self-configuration in RACG1). If **RACG1** is conformed to the index category **RACG-Formation** by restarting violated *RAOL1* or *RAOL3*, it evolves to **RACG1-1** (consisting of the objects RACS1, RAOL1 or RAOL1-1, RAOL2, RAOL3 or RAOL3-1 and the morphisms among them in **RACG1-0**) that has the same configuration and categorical structure as **RACG1** except for the different initial status of *RAOL1* or *RAOL3*. This evolution is specified by a *Restart* functor (a structure-preserving mapping) from **RACG1-1** to **RACG1-0**. If **RACG1** is conformed to the **RACG-Formation** by substituting *RAOL1* or *RAOL3* with their isomorphic objects *RAOL7* or *RAOL9* (see Definition 3.1.3), it will evolve to **RACG1-2** (consisting of objects RACS1, RAOL1 or RAOL7, RAOL2, RAOL3 or RAOL9 and the morphisms among them in **RACG1-0**), which has the same configuration and categorical structure as the **RACG1** but replacing *RAOL1* or *RAOL3* with *RAOL7* or *RAOL9*. The above is specified by a *Substitute* functor, a structure-preserving mapping. If **RACG1** is conformed to **RACG-Formation** by asking *RAOL2* to take over the responsibilities of *RAOL1* or *RAOL3*, it evolves to **RACG1-3** (consisting of the objects RACS1-1, SPM, RAOL1 or RAOL3 and the morphisms among them in **RACG1-0**), which has the different categorical structure, but both of them have the equivalent configuration (see Property 8.7.5 and the figure below).

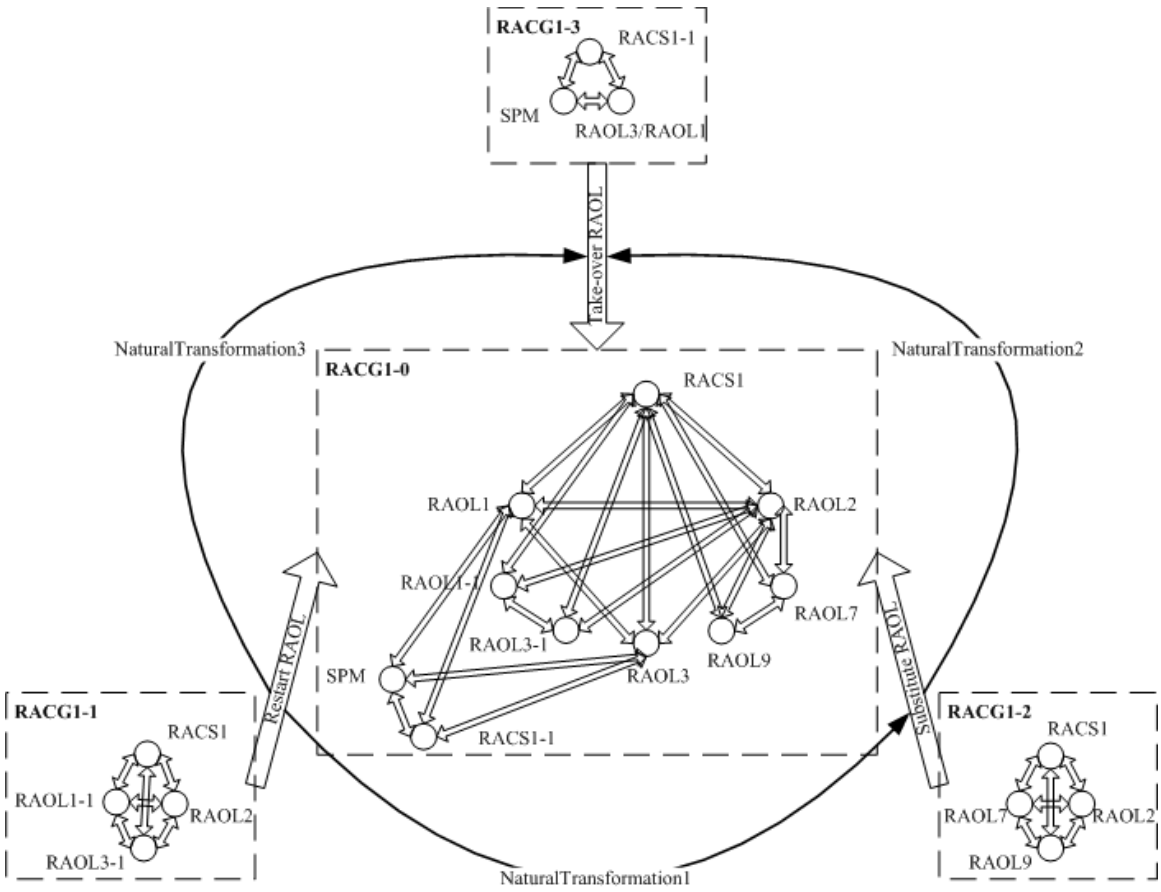


Figure 107: Evolution for Self-Configuration in RACG1

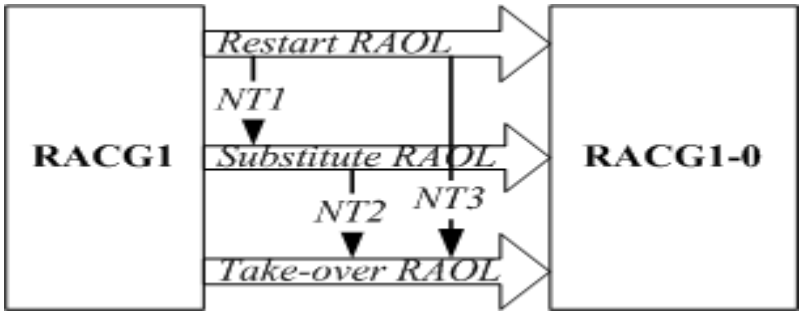


Figure 108: Natural Transformation for Self-Configuration in RACG1

The mapping among those evolutions *RestartRAOL*, *SubstituteRAOL* and *Take-over-RAOL* of the **RACG1** can be interpreted as natural transformations (see Property 6.1.4). The functor category having those functors as objects and their natural transformations as morphisms illustrate all possible evolutions with their relations. In addition, the natural

transformation3 is a composition of natural transformation1 and natural transformation2, which may be interpreted as the following: the result of the evolution *RestartRAOL -> SubstituteRAOL -> Take-over-RAOL* is equivalent to the evolution *RestartRAOL -> Take-over-RAOL*. Figure 109, 110 and 111 illustrate those natural transformations and their composition respectively.

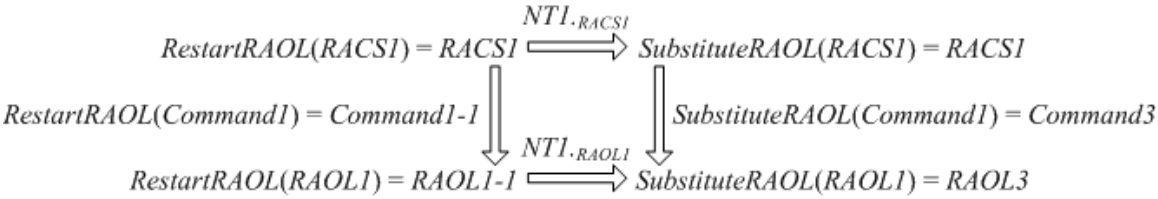


Figure 109: Natural Transformation *RestartRAOL -> SubstituteRAOL* in RACG1

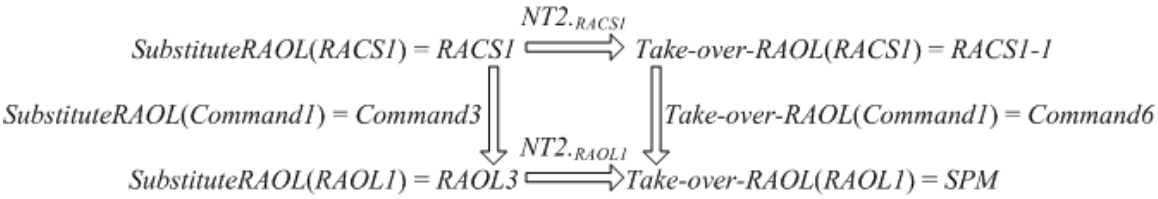


Figure 110: Natural Transformation *SubstituteRAOL -> Take-over-RAOL* in RACG1

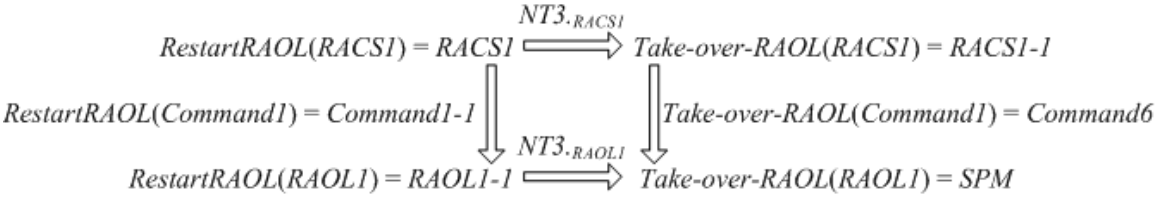


Figure 111: Natural Transformation *RestartRAOL -> Take-over-RAOL* in RACG1

Let **RACG1** be a subcategory (consisting of the objects RACS1, RAOL1, RAOL2, RAOL3 and the morphisms among them) of **RACG1-0'** (a category consisting of all the potential RAE for the communication self-configuration within RACG1). If **RACG1** is conformed to the **RACG-Formation** by restarting the communication from *RACS1* to *RAOL1* or restarting *RACS1*, it evolves to **RACG1-4** (consisting of objects RACS1-1,

RAOL1, RAOL2, RAOL3 and the morphisms among them in the **RACG1-0'**), which has the same configuration and categorical structure as **RACG1** except for the different initial status of *RACS1*. This evolution is specified by the *RestartRACS* functor (a structure-preserving mapping) from **RACG1-4** to **RACG1-0'**. If **RACG1** is conformed to **RACG- Formation** by substituting *RACS1* with its isomorphic objects *RACS3* (see Definition 3.1.3), it evolves to **RACG1-5** (consisting of objects RACS3, RAOL1, RAOL2, RAOL3 and the morphisms among them in the **RACG1-0'**), which has the same configuration and categorical structure as **RACG1** but replacing *RACS1* with *RACS3*. The above is specified by the *SubstituteRACS* functor, a structure-preserving mapping. If **RACG1** is conformed to the **RACG-Formation** by asking *RACS2* to take over the responsibilities of *RACS1*, it evolves to **RACG1-6** (consisting of objects SPM, RAOL1, RAOL2, RAOL3 and the morphisms among them in **RACG1-0'**), which has different categorical structure, but both of them have the equivalent configuration (see Property 8.7.5 and Figure 112).

The mapping among those evolutions *RestartRACS*, *SubstituteRACS* and *Take-over-RACS* of the **RACG1** is interpreted as natural transformations (see Property 6.1.4). The functor category having those functors as objects and their natural transformations as morphisms illustrate all possible evolutions with their relations. In addition, the natural transformation6 is a composition of natural transformation4 and natural transformation5, which may be interpreted as the following: the result of the evolution *RestartRACS -> SubstituteRACS -> Take-over-RACS* is equivalent to the evolution *RestartRACS -> Take-*

over-RACS. Figure 114, 115 and 116 illustrate those natural transformations and their composition respectively.

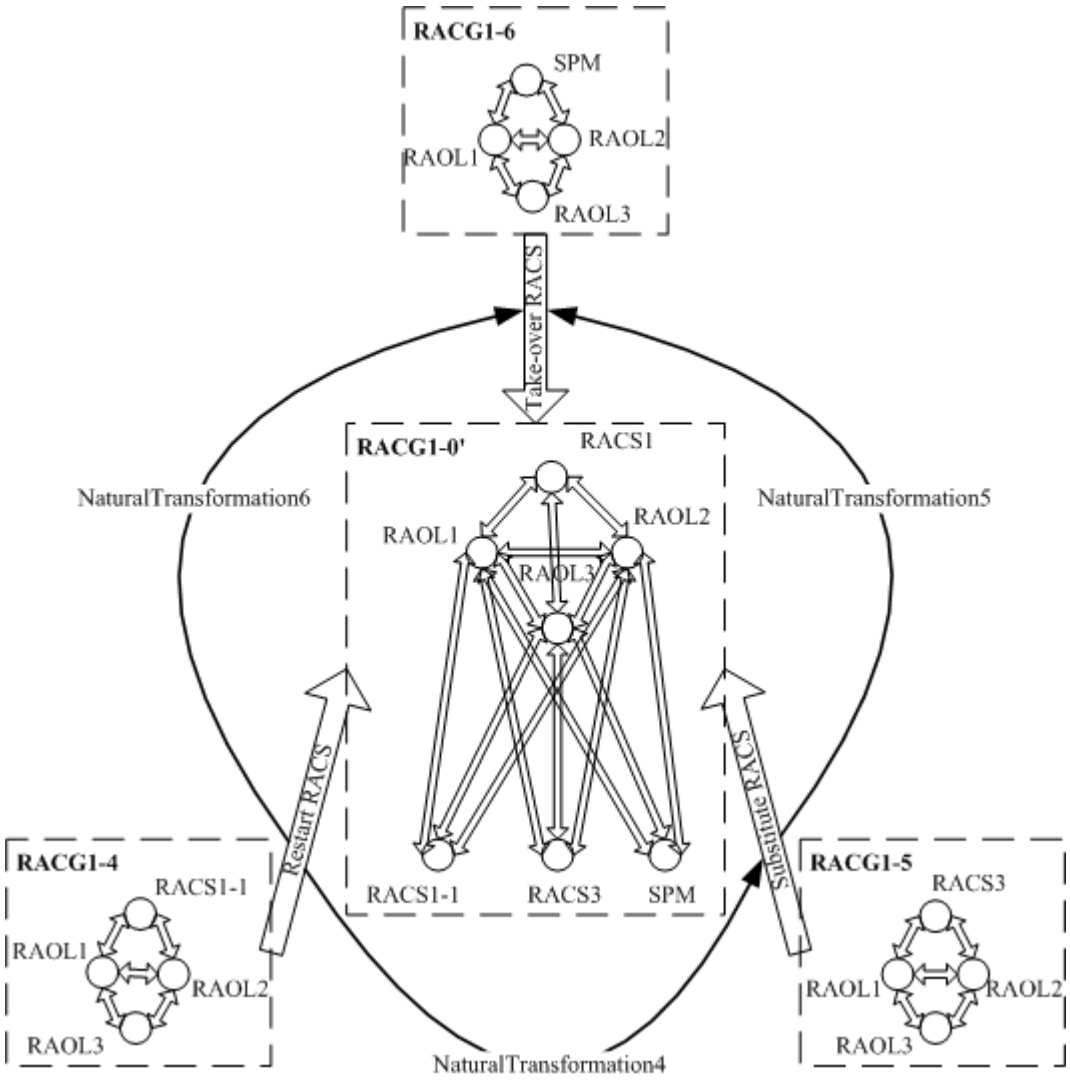


Figure 112: Evolution for Communication Self-Configuration in RACG1

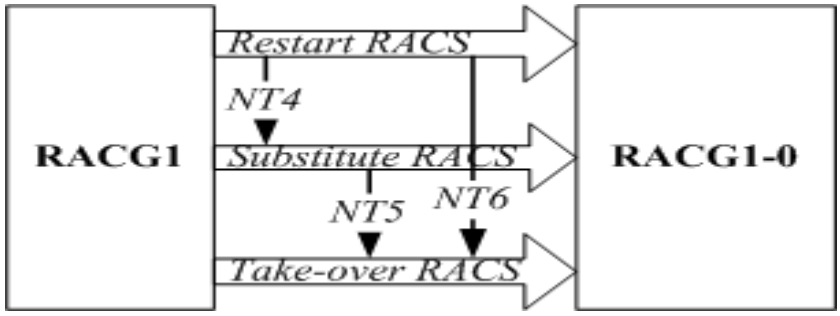


Figure 113: Natural Transformation of Communication Self-Configuration in RACG1

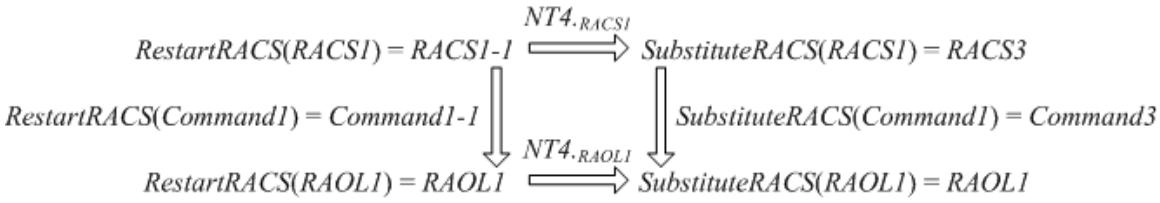


Figure 114: Natural Transformation *RestartRACS -> SubstituteRACS* in RACG1

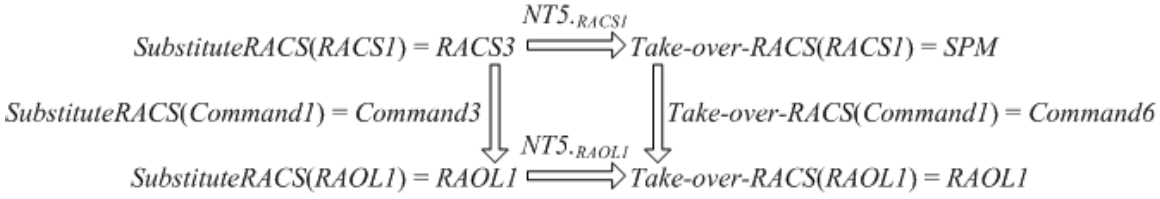


Figure 115: Natural Transformation *SubstituteRACS -> Take-over-RACS* in RACG1

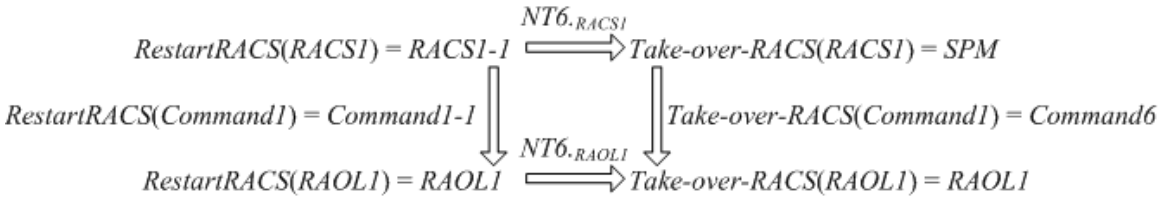


Figure 116: Natural Transformation *RestartRACS -> Take-over-RACS* in RACG1

When both *RACS1* and *RACS2* cannot conform to **RACG-Formation** at the same time, *RACGM1* tries to restart them first. If neither of them can conform to **RACG-Formation** after being restarted, *RACGM1* will substitute *RACS1* and *RACS2* with their isomorphic objects *RACS3* and *RACS4*; otherwise, the remaining process is the same as the illustration above. When *RACGM1* cannot find *RACS3* or *RACS4*, it will send an *action-required* message to *User Console*; otherwise, the description above may indicate the remaining process. If a **RACG** consists of more than two *RACS*, a similar categorical representation can be generated as we explained previously.

8.5 Forming a RAC

After receiving the task of forming a RAC from RACS, RAOL starts to create RAO and

establish corresponding connections between them based on the composition rules and communication protocols specified by the index category **RAC-Formation** (see Property 6.1.1). Figure 117 depicts an example of forming the categories **RAC1** and **RAC2** from their index category **RAC-Formation**. Figure 118 and 119 describe the detailed object as well as morphism mappings from **RAC1** and **RAC2** to **RAC-Formation** respectively.

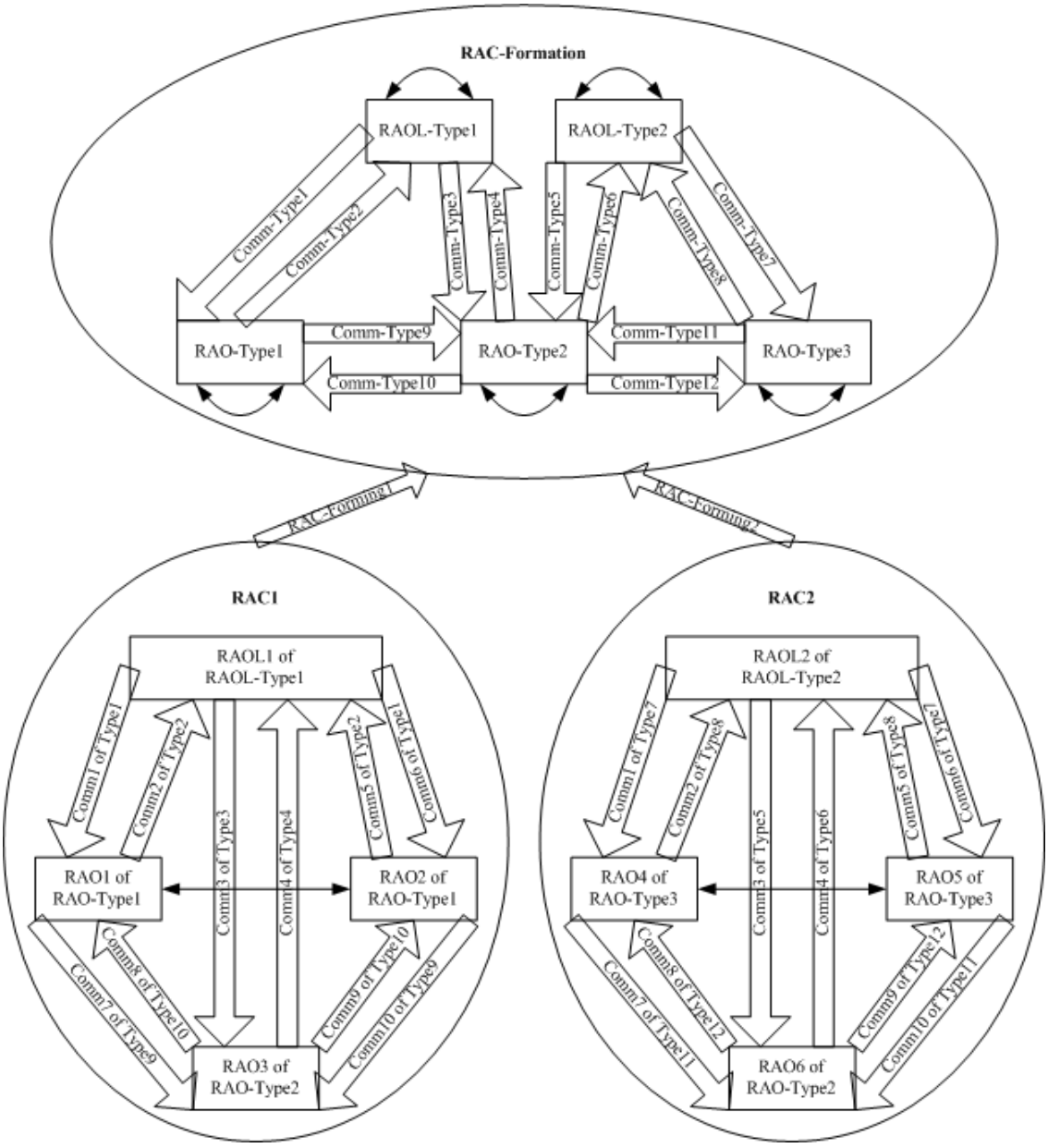


Figure 117: Example of Forming RAC from **RAC-Formation**

Object Mapping	Morphism Mapping (RAC-Formation <- RAC1)
RAOL->RAOL-Type1	Comm1->Comm-Type1, Comm2->Comm-Type2
RAO1->RAO-Type1	Comm3->Comm-Type3, Comm4->Comm-Type4
RAO2->RAO-Type1	Comm5->Comm-Type2, Comm6->Comm-Type1
RAO3->RAO-Type2	Comm7->Comm-Type9, Comm8->Comm-Type10
	Comm9->Comm-Type10, Comm10->Comm-Type9
	RAO1–RAO2-> Identity-Mor(RAO-Type1)

Figure 118: Object and Morphism Mapping of Functor RAC-Forming1

Object Mapping	Morphism Mapping (RAC-Formation <- RAC2)
RAOL2->RAOL-Type2	Comm1->Comm-Type7, Comm2->Comm-Type8
RAO4->RAO-Type3	Comm3->Comm-Type5, Comm4->Comm-Type6
RAO5->RAO-Type3	Comm5->Comm-Type8, Comm6->Comm-Type7
RAO6->RAO-Type2	Comm7->Comm-Type11, Comm8->Comm-Type12
	Comm9->Comm-Type12, Comm10->Comm-Type11
	RAO4–RAO5-> Identity-Mor(RAO-Type3)

Figure 119: Object and Morphism Mapping of Functor RAC-Forming2

After *RAOL1* initializes its *RAO* according to the requirements from *RACS1* and the capabilities of those *RAO*, it validates the configuration of those *RAO* against their types every *t* ticks (a tick is an abstraction of one time unit under a global clock in the **RAC1**), while *RAOL1* is in the first state of its intelligent control loop (monitoring). If the

configuration of those *RAO* conforms to their types, composition rules as well as communication protocols (see Property 8.7.4), *NoViolation* event keeps the *RAOL1* in *Monitor* state; otherwise, *NeedInvestigation* event is triggered and *RAOL1* transits to *Analyze* state, while a time constraint variable (*TCvar3*) is initialized to work as a local clock in terms of time constraints on each transition of the intelligent control loop. The value of *TCvar2* is *t0*, *t1*, *t2*, *t3*... where $t0 < t1 < t2 < t3$.

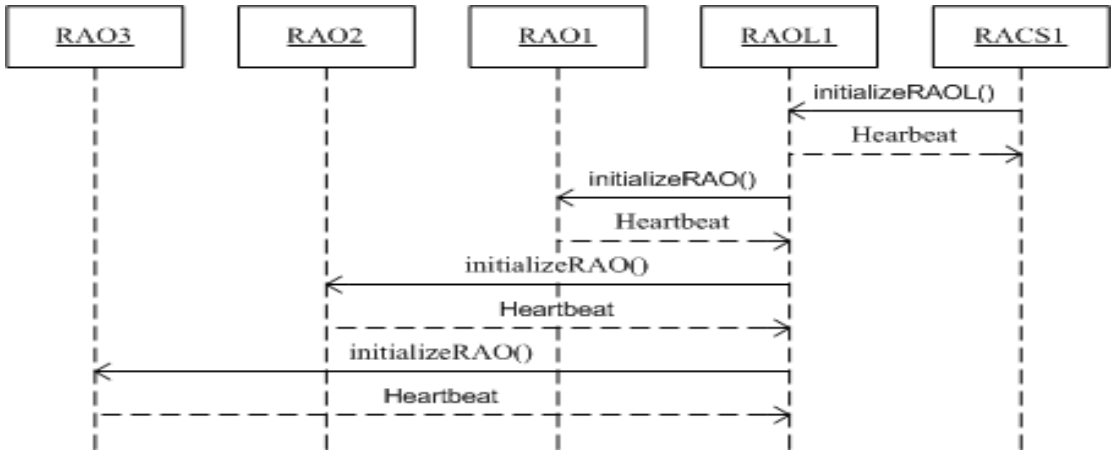


Figure 120: RAC Formation Work Flow

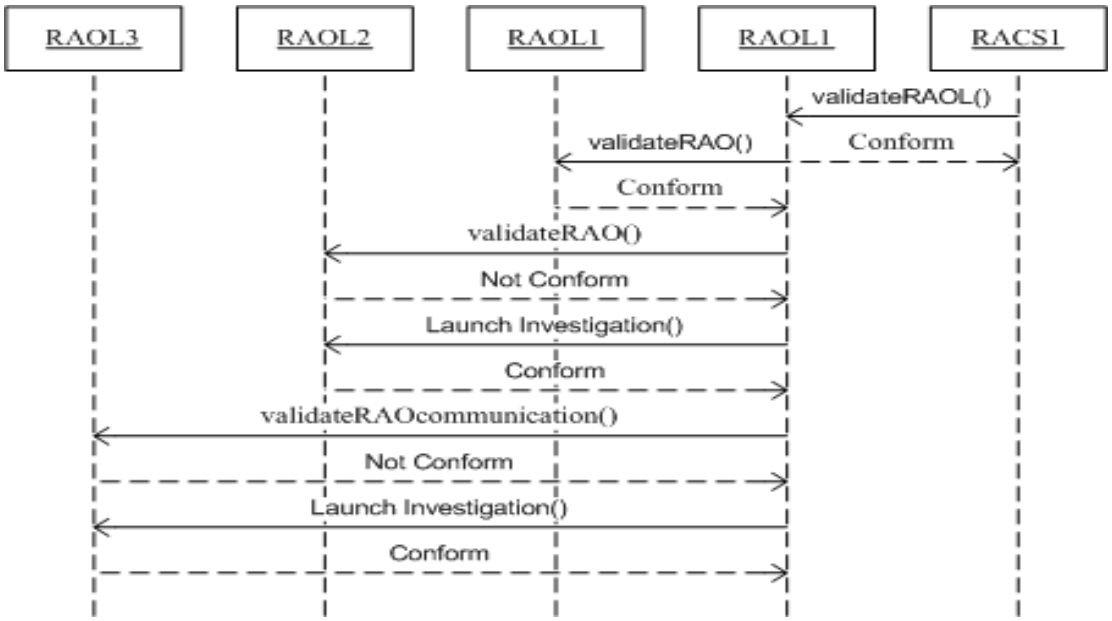


Figure 121: RAC Self-Configuration Work Flow

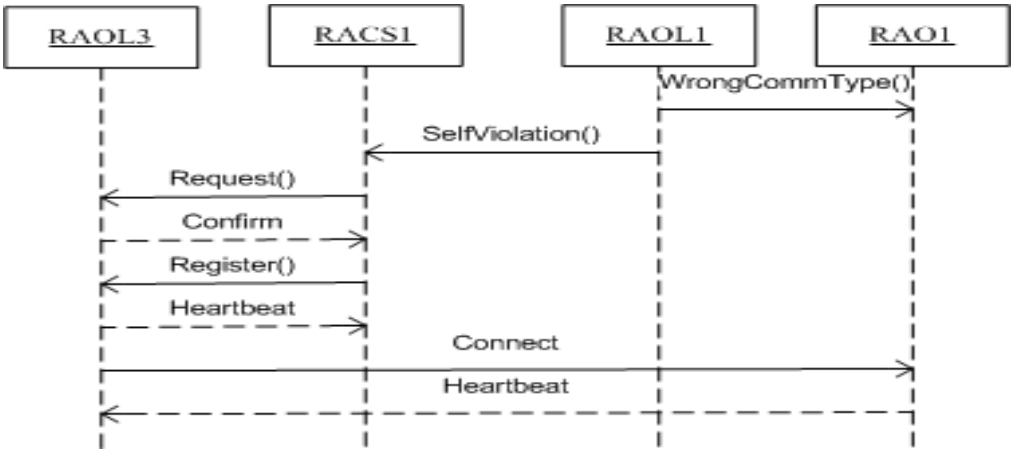


Figure 122: RAC Self-Configuration Work Flow by Substitution

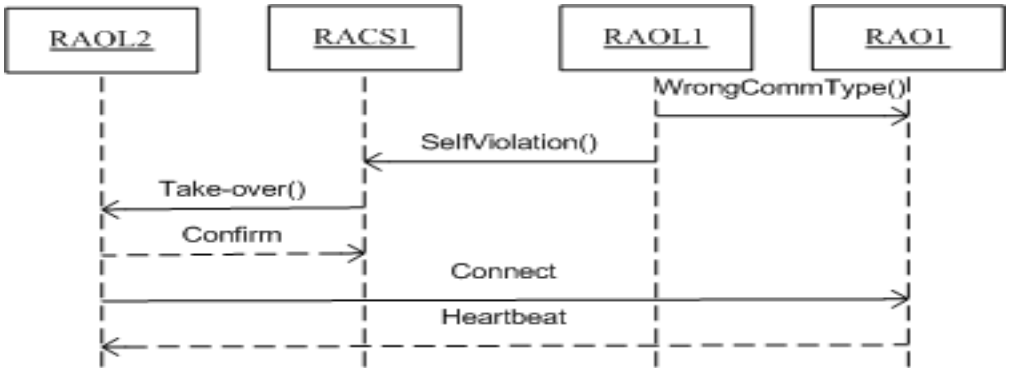


Figure 123: RAC Self-Configuration Work Flow by Take-over

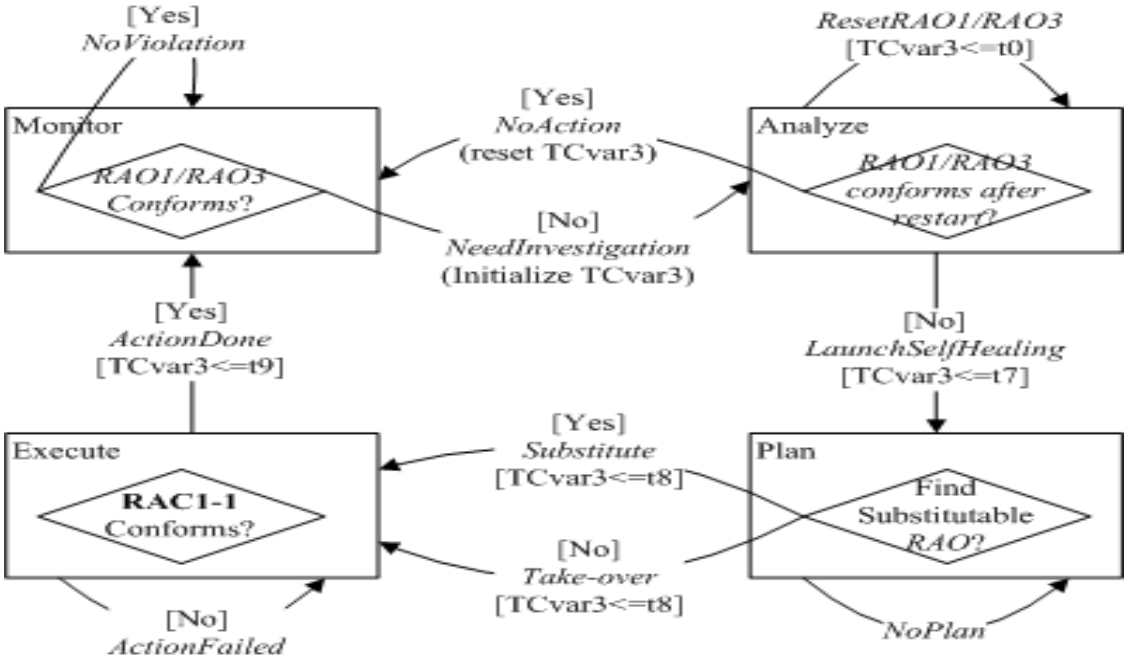


Figure 124: Intelligent Control Loop in RAC1 for Self-Configuration

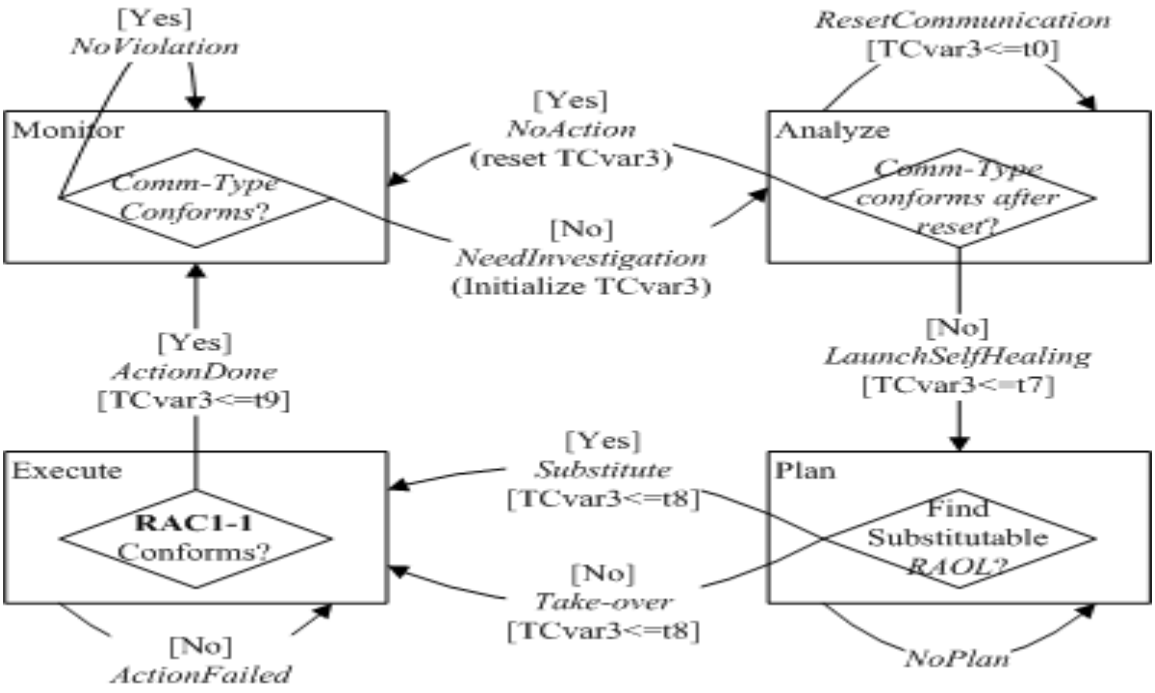


Figure 125: ICL in RAC1 for Communication Self-Configuration

After *RAOL1* enters *Analyze* state, 1) it sends a *Restart* message to *RAO1* in *t0* ticks where the violation is caused by the incorrect RAO type or incorrect communication type from *RAO1* to *RAOL1*. If *RAO1* conforms to its type or communication type, *NoAction* event occurs and *RAOL1* goes back to *Monitor* state, while the *TCvar3* is reset; otherwise, *RAOL1* transits to *Plan* state triggered by *LaunchSelfHealing* event in *t7* ticks. 2) If the violation is caused by incorrect communication type from other RAO (*RAO3*) to *RAO1*, *RAOL1* sends a *Restart* message to *RAO3*. If the communication conforms to its type, *NoAction* event occurs and *RAOL1* goes back to *Monitor* state, while the *TCvar3* is reset; otherwise, *RAOL1* transits to *Plan* state triggered by the *LaunchSelfHealing* event within *t7* ticks. 3) If the violation is caused by the incorrect communication type from *RAOL1* to *RAO1*, *RAOL1* resets that communication. If it conforms to the correct one specified in

the index category **RAC-Formation** (see Property 8.7.4), *NoAction* event occurs and *RAOL1* goes back to the *Monitor* state, while *TCvar3* is reset; otherwise, *RAOL1* transits to *Plan* state triggered by *LaunchSelfHealing* event in *t7* ticks.

When *RAOL1* is in *Plan* state, it chooses either *Substitute* plan or *Take-over* plan, based on the availability of substitutable RAO for *RAO1* (scenario 1 in the paragraph above) or for *RAO3* (scenario 2). *RAOL1* transits to *Execute* state triggered by the *Substitute* event or *Take-over* event respectively in *t8* ticks. For scenario 3, *RAOL1* sends a *selfViolation* message to *RACS1*, and the latter chooses either *Substitute* plan or *Take-over* plan according to the availability of substitutable RAOL for *RAOL1*. It transits to *Execute* state triggered by *Substitute* or *Take-over* event in *t8* ticks.

When *RAOL1* is in *Execute* state and *Substitute* plan is applicable, it sends a *register* message to the substitutable RAO of *RAO1* (scenario 1) or *RAO3* (scenario 2) and then initialize it to the status of *RAO1* or *RAO3* based on the checkpoint made before. When the *take-over* plan is applicable, *RAOL1* sends a *take-over* message to *RAO2* (scenario 1 or scenario 2) and update it to the status of the synchronous product machine of *RAO1* and *RAO2*, or *RAO3* and *RAO2* according to the checkpoint. After the plan execution, *RAOL1* validates the configuration of **RAC1'**, an evolution of **RAC1** (see Property 6.1.3) against the index category **RAC-Formation** based on their categorical specifications. If that configuration conforms to the index category (see Property 8.7.4), *ActionDone* event occurs and *RAOL1* transits to the *Monitor* state within *t9* ticks; otherwise, *ActionFailed* event keeps it in *Execute* state for *RACS1*'s intervention. For scenario 3, *RAOL1* is

substituted by *RAOL3* or taken over by *RAOL2*.

8.6 Categorical Illustration of Forming a RAC

The actions in the formation work flow, self-configuration work flow, substitution work flow and take-over work flow of RAC1 can be specified as the categories where objects are the actions (*InitializeRAO*, *ValidateRAO*, *ValidateRAOcommunication*, etc.), and morphisms are their preorder relationship *before*. Each object (action) in those categories is a quadruple (see Property 6.2.28). For example, *LaunchInvestigation* = (*RAOL1*, *NotConfrom-RAO*, *InvestigateRAO*, *RAO1*), and the sequences of those actions can be specified as the categories in which objects are those sequences (<*InitializeRAOL*, Heartbeat, *InitializeRAO*, *Heartbeat*>, <*ValidateRAOL*, *Conform*, *ValidateRAO*, *NotConform*>), and morphisms are the *equivalence* relationship between those sequences (see Property 6.2.31).

The transitions in the intelligent control loop of RAOL1 for self-configuration can be specified as the category in which objects are those transitions (*NoViolation, NeedInvestigation*, *RestartRAO*, *NoAction*, etc.), and morphisms are their preorder relations *before*. Each object (transition) in that category is a triple (see Property 6.2.23). For example, *NeedInvestigation* = (*Monitor*, *NotConform-RAO*, *Analyze*); the sequences of those transitions can be specified as a category in which objects are those sequences (<*NoViolation*, *NeedInvestigation*, *RestartRAO1*, *NoAction*>, <*RestartRAO1*, *LaunchSelfHealing*, *Substitute*, *ActionDone*>), and morphisms are *equivalence* relations between those sequences (see Property 6.2.26).

Let **RAC1** be a subcategory (consisting of objects RAOL1, RAO1, RAO2, RAO3 and the morphisms among them) of **RAC1-0** (a category consisting of all the potential RAE for the self-configuration in RAC1). If **RAC1** is conformed to the index category **RAC-Formation** by restarting the violated *RAO1* or *RAO3*, it will evolve to **RAC1-1** (consisting of the objects RAOL1, RAO1 or RAO1-1, RAO2, RAO3 or RAO3-1 and the morphisms among them in **RAC1-0**), which has the same configuration and categorical structure as **RAC1** except for the different initial status of *RAO1* or *RAO3*. This evolution is specified by a *Restart* functor (a structure-preserving mapping) from **RAC1-1** to **RAC1-0**. If **RAC1** is conformed to **RAC-Formation** by substituting *RAO1* or *RAO3* with their isomorphic objects *RAO7* or *RAO9* (see Definition 3.1.3), it will evolve to **RAC1-2** (consisting of objects RAOL1, RAO1 or RAO7, RAO2, RAO3 or RAO9 and the morphisms among them in the **RAC1-0**), which has the same configuration and categorical structure as **RAC1** but replacing *RAO1* or *RAO3* with *RAO7* or *RAO9*. The above is specified by a *Substitute* functor, a structure-preserving mapping. If **RAC1** is conformed to the **RAC-Formation** by asking *RAO2* to take over the responsibilities of *RAO1* or *RAO3*, it evolves to **RAC1-3** (consisting of objects RAOL1-1, SPM, RAO1 or RAO3 and the morphisms among them in **RAC1-0**), which has the different categorical structure, but both of them have the equivalent configuration (see Property 8.7.5 and the figure below).

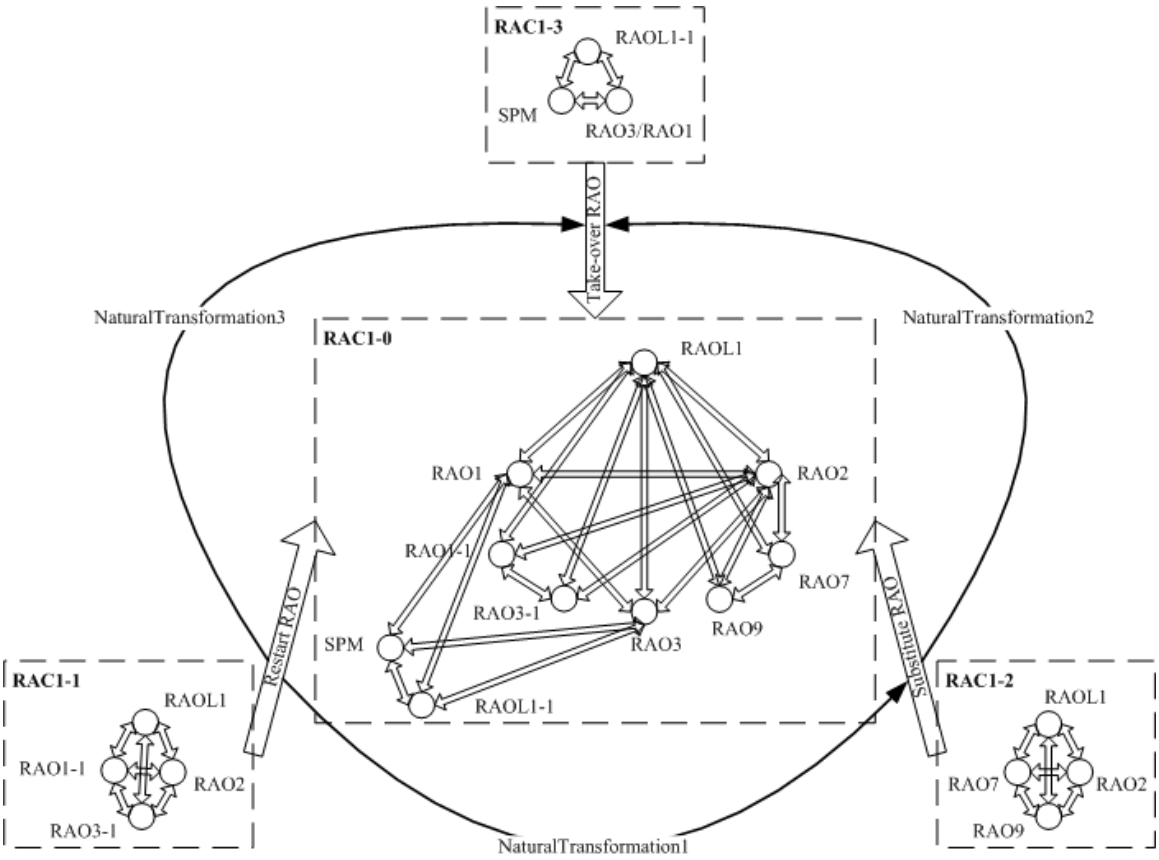


Figure 126: Evolution for Self-Configuration in RAC1

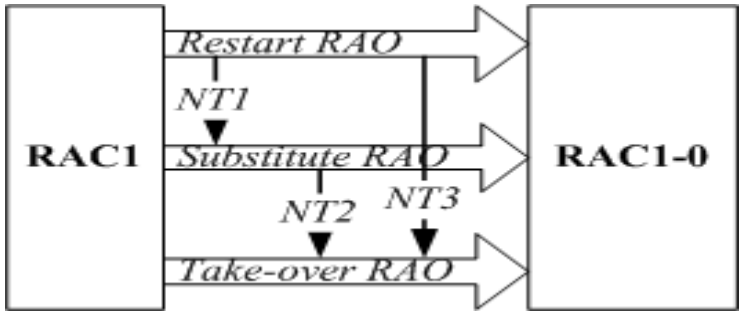


Figure 127: Natural Transformation for Self-Configuration in RAC1

The mapping among those evolutions *RestartRAO*, *SubstituteRAO* and *Take-over-RAO* of the **RAC1** can be interpreted as natural transformations (see Property 6.1.4). The functor category having those functors as objects and their natural transformations as morphisms illustrate all possible evolutions with their relations. In addition, the natural

transformation3 is a composition of natural transformation1 and natural transformation2, which may be interpreted as the following: the result of the evolution *RestartRAO -> SubstituteRAO -> Take-over-RAO* is equivalent to the evolution *RestartRAO -> Take-over-RAO*. Figure 128, 129 and 130 illustrate those natural transformations and their composition respectively.

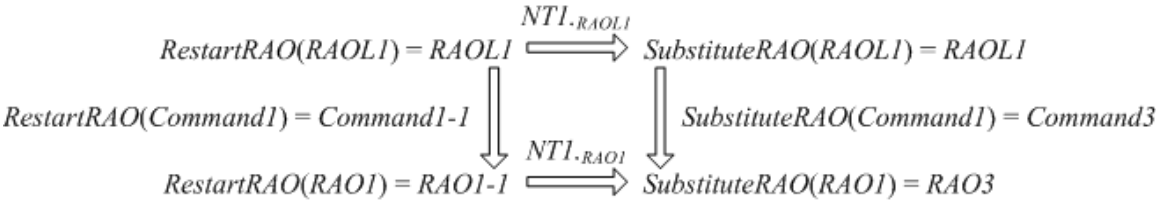


Figure 128: Natural Transformation *RestartRAO -> SubstituteRAO* in RAC1

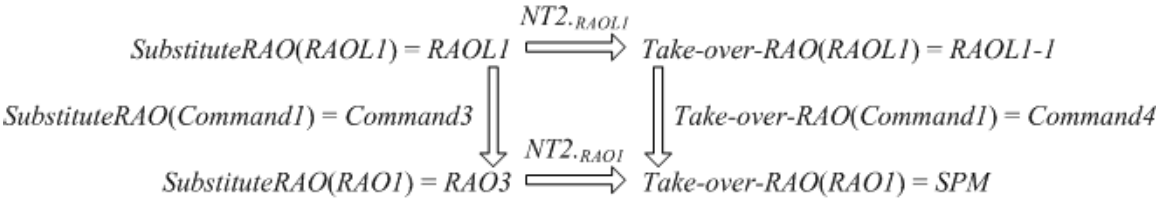


Figure 129: Natural Transformation *SubstituteRAO -> Take-over-RAO* in RAC1

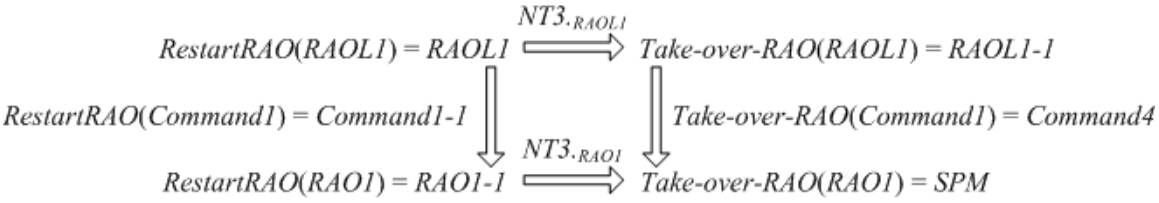


Figure 130: Natural Transformation *RestartRAO -> Take-over-RAO* in RAC1

Let **RAC1** be a subcategory (consisting of objects RAOL1, RAO1, RAO2, RAO3 and the morphisms among them) of **RAC1-0'** (a category consisting of all the potential RAE for the communication self-configuration in RAC1). If **RAC1** is conformed to **RAC- Formation** by restarting the communication from *RAOL1* to *RAO1* or restarting *RAOL1*, it will evolve to **RAC1-4** (consisting of objects RAOL1-1, RAO1, RAO2, RAO3

and the morphisms among them in **RAC1-0'**), which has the same configuration and categorical structure as **RAC1** except for the different initial status of *RAOL1*. This evolution is specified by a *RestartRAOL* functor from **RAC1-4** to **RAC1-0'**. If **RAC1** is conformed to the **RAC-Formation** by substituting *RAOL1* with its isomorphic objects *RAOL3* (see Definition 3.1.3), it evolves to **RAC1-5** (consisting of objects RAOL3, RAO1, RAO2, RAO3 and the morphisms among them in **RAC1-0'**), which has the same configuration and categorical structure as **RAC1** but replacing the *RAOL1* with *RAOL3*. The above is specified by the *SubstituteRAOL* functor, a structure-preserving mapping. If **RAC1** is conformed to **RAC-Formation** by asking *RAOL2* to take over the responsibilities of *RAOL1*, it evolves to **RAC1-6** (consisting of objects SPM, RAO1, RAO2, RAO3 and the morphisms among them in **RAC1-0'**), which has different categorical structure, but both of them have the equivalent configuration (see Property 8.7.5 and Figure 131).

The mapping among those evolutions *RestartRAOL*, *SubstituteRAOL* and *Take-over-RAOL* of the **RAC1** is interpreted as natural transformations (see Property 6.1.4). The functor category having those functors as objects and their natural transformations as morphisms illustrate all possible evolutions with their relations. In addition, the natural transformation6 is a composition of natural transformation4 and natural transformation5, which may be interpreted as the following: the result of the evolution *RestartRAOL -> SubstituteRAOL -> Take-over-RAOL* is equivalent to the evolution *RestartRAOL -> Take-over-RAOL*. Figure 133, 134 and 135 illustrate those natural transformations and their

composition respectively.

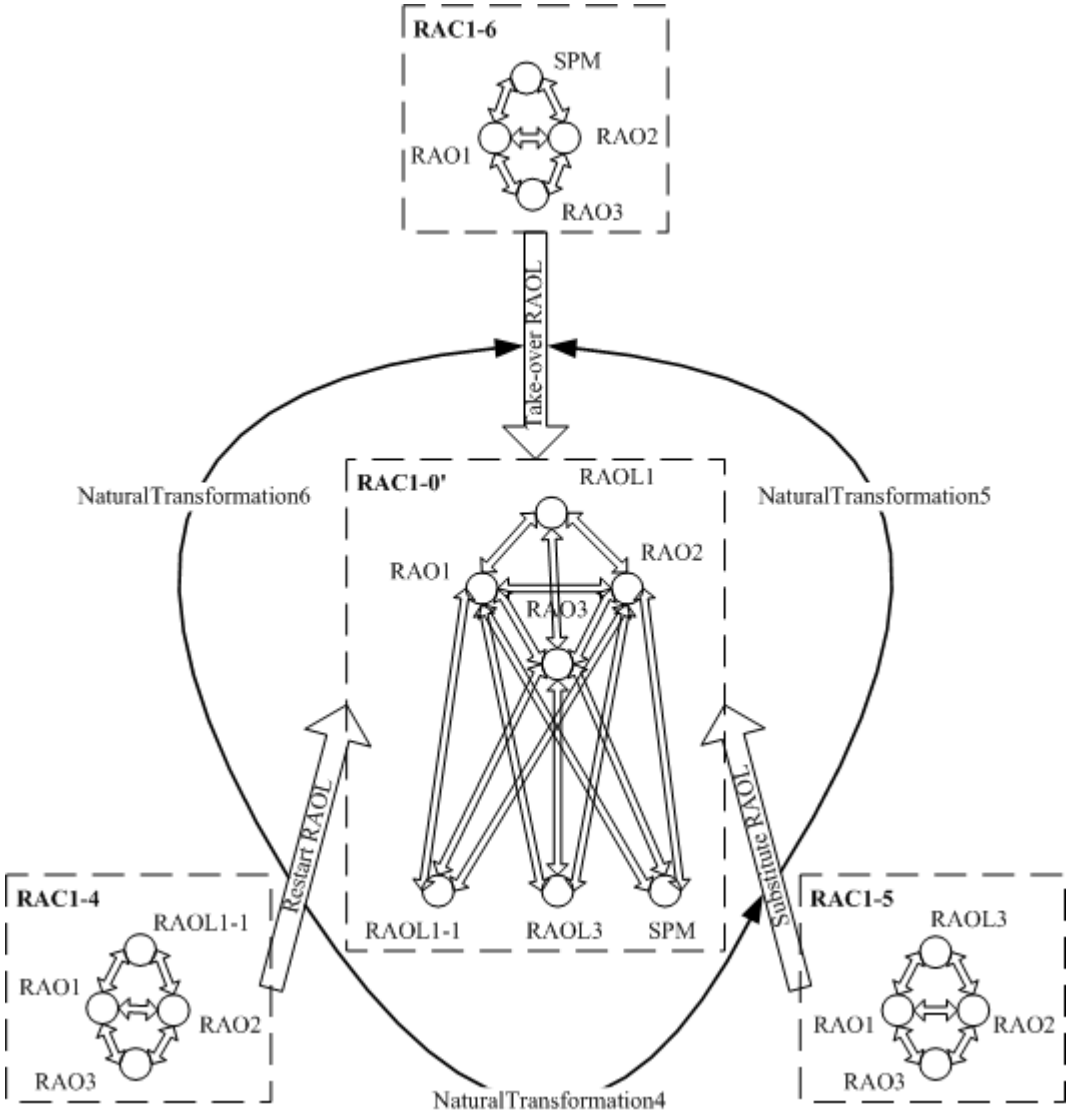


Figure 131: Evolution for Communication Self-Configuration in RAC1

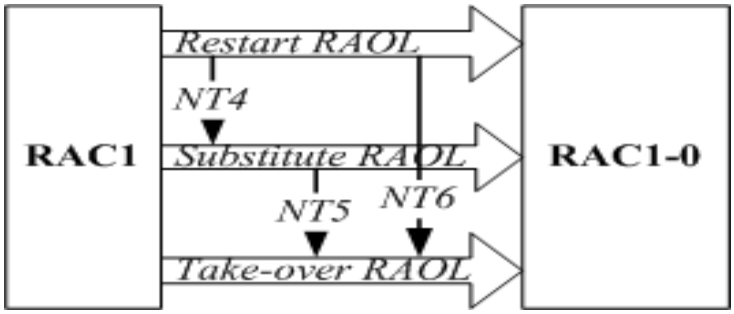


Figure 132: Natural Transformation of Communication Self-Configuration in RAC1

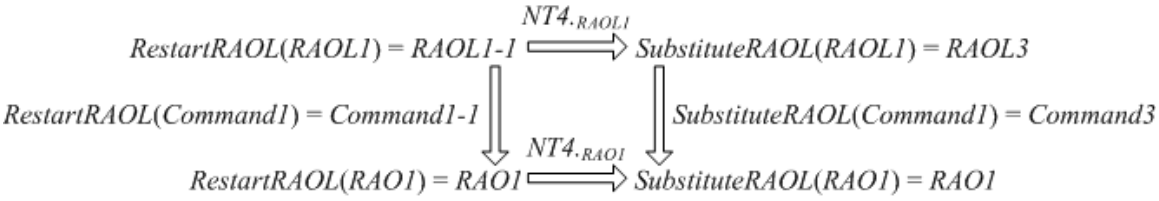


Figure 133: Natural Transformation *RestartRAOL -> SubstituteRAOL* in RAC1

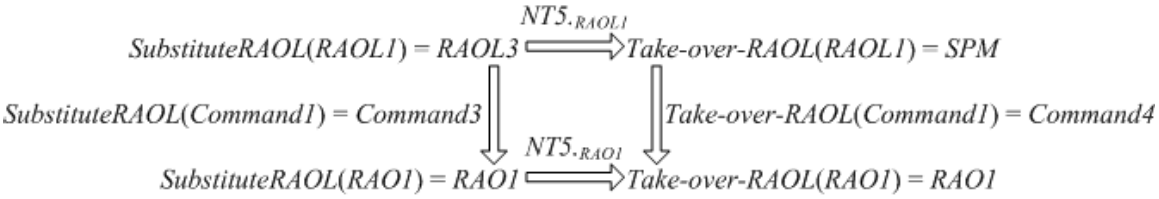


Figure 134: Natural Transformation *SubstituteRAOL -> Take-over-RAOL* in RAC1

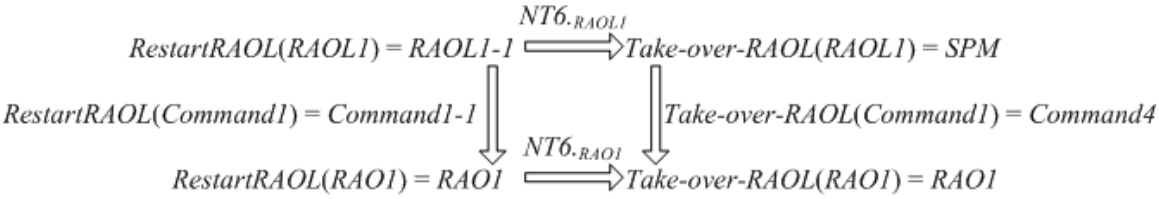


Figure 135: Natural Transformation *RestartRAOL -> Take-over-RAOL* in RAC1

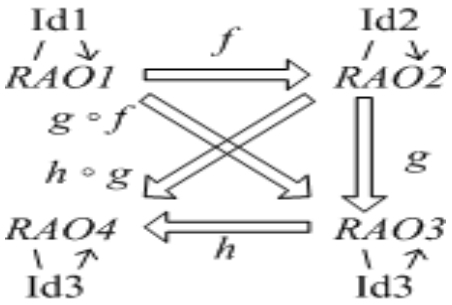
When both *RAC1* and *RAC2* cannot conform to **RAC-Formation** at the same time, *RACS1* tries to restart them first. If neither of them can conform to **RAC-Formation** after being restarted, *RACS1* will substitute *RAOL1* and *RAOL2* with their isomorphic objects *RAOL3* as well as *RAOL4*; otherwise, the remaining process is the same as the illustration above. When *RACS1* cannot find the *RAOL3* or *RAOL4*, it will send an *action-required* message to the *RACGM1*; otherwise, the description above may indicate the remaining process. If a **RAC** consists of more than two *RAOL*, the similar categorical representation can be generated as we explained previously.

8.7 Categorical Specifications of Self-Configuration

Property 8.7.1: The configuration of a *RAC* is a category denoted as **CONFIG**(*RAC*),

where objects are *RAO* and morphisms are connections between those *RAO* as **CONFIG** (*RAO*, *RAO*$'$) or **CONFIG**(*RAO*$'$, *RAO*).

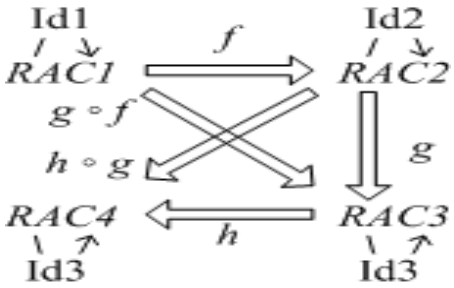
Proof. *All what we need is to prove: i) the existence of composition and identity morphism, and ii) prove associativity. Let* RAO_1, RAO_2 *and* RAO_3 *be three RAO such that* RAO_1 *connects to* RAO_2, *which connects to* RAO_3, *so* RAO_1 *connects to* RAO_3 *(indirectly through* RAO_2*), meaning that the existence of a composition of morphisms between* RAO_1 *and* RAO_3. *The identity morphism does exist as the natural representation of internal connections. Let f, g and h be the morphisms such that f:* $RAO_1 \rightarrow RAO_2$, *g:* $RAO_2 \rightarrow RAO_3$ *and h:* $RAO_3 \rightarrow RAO_4$. *It is clear that* $h \circ (g \circ f) = (h \circ g) \circ f$.■



Property 8.7.2: The configuration of a *RACG* is the category denoted as **CONFIG** (*RACG*), where objects are *RAC* and morphisms are connections between those *RAC* as **CONFIG** (*RAC*, *RAC*$'$) or **CONFIG**(*RAC*$'$, *RAC*).

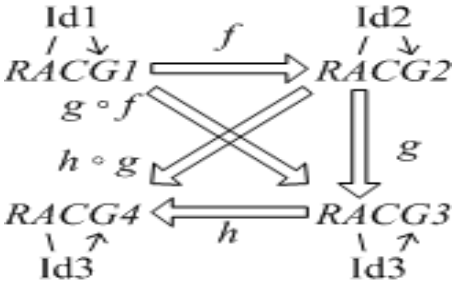
Proof. *All what we need is to prove: i) the existence of composition and identity morphism, and ii) prove associativity. Let* RAC_1, RAC_2 *and* RAC_3 *be three RAC such that* RAC_1 *connects to* RAC_2, *which connects to* RAC_3, *so* RAC_1 *connects to* RAC_3 *(indirectly through* RAC_2*), meaning that the existence of a composition of morphisms between* RAC_1 *and* RAC_3. *The identity morphism does exist as the natural representation of internal*

connections. Let f, g and h be the morphisms such that f: $RAC_1 \rightarrow RAC_2$, g: $RAC_2 \rightarrow RAC_3$ and h: $RAC_3 \rightarrow RAC_4$. It is clear that $h \circ (g \circ f) = (h \circ g) \circ f$.■



Property 8.7.3: The configuration of a *RAS* is a category denoted as **CONFIG** (*RAS*), where objects are *RACG* and morphisms are the connections between those *RACG* as **CONFIG** (*RACG*, *RACG*$'$) or **CONFIG**(*RACG*$'$, *RACG*).

Proof. *All what we need is to prove: i) the existence of composition and identity morphism, and ii) prove associativity. Let $RACG_1$, $RACG_2$ and $RACG_3$ be three RACG such that $RACG_1$ connects to $RACG_2$, which connects to $RACG_3$, so $RACG_1$ connects to $RACG_3$ (indirectly through $RACG_2$), meaning that the existence of a composition of morphisms between $RACG_1$ and $RACG_3$. The identity morphism does exist as the natural representation of internal connections. Let f, g and h be the morphisms such that f: $RACG_1 \rightarrow RACG_2$, g: $RACG_2 \rightarrow RACG_3$ and h: $RACG_3 \rightarrow RACG_4$. It is clear that $h \circ (g \circ f) = (h \circ g) \circ f$.*■



Property 8.7.4: The configuration of *RAE* is conformed to the configuration of *RAE-Formation* iff there exist a functor *F* from **CONFIG**(*RAE*) to **CONFIG**(*RAE-Formation*). The functor *F* guarantees all the objects and morphisms in **CONFIG**(*RAE*) have their mapped object types and morphism types in **CONFIG**(*RAE-Formation*).

Property 8.7.5: Two RAE's configurations are considered to be equivalent iff their social lives are equivalent (see Property 7.7.2) and they both conform to the configuration of RAE-Formation *configRAE-Formation*.

8.8 Representation of Categorical Specification of Self-Configuration

The figure below depicts an example of the representation for a categorical specification (in XML format) of the self-configuration property we present earlier in this chapter, and more XML representation can be found in Appendix E.

```
<CATEGORY name = "Formation-Work-Flow-in-RAS">
    <OBJECT>
        <OBJECT name = "InitializeRACGM" type = "Work-Flow-Action"/>
        <OBJECT name = "InitializeRACS" type = "Work-Flow-Action"/>
        <OBJECT name = "Heartbeat" type = "Work-Flow-Action"/>
    </OBJECT>
    <MORPHISM>
        <MORPHISM name = "Before" type ="Preorder"/>
            <FROM-OBJECT name = "InitializeRACGM"
                                type = "Work-Flow-Action"/>
            <TO-OBJECT name = "Heartbeat" type = "Work-Flow-Action"/>
        <MORPHISM>
        <MORPHISM name = "Before" type ="Preorder"/>
            <FROM-OBJECT name = "InitializeRACGM"
                                type = "Work-Flow-Action"/>
            <TO-OBJECT name = "InitializeRACS" type = "Work-Flow-Action"/>
        <MORPHISM>
        <MORPHISM name = "Before" type ="Preorder"/>
            <FROM-OBJECT name = "InitializeRACS"
```

```
                                type = "Work-Flow-Action"/>
            <TO-OBJECT name = "Heartbeat" type = "Work-Flow-Action"/>
        <MORPHISM>
    </MORPHISM>
</CATEGORY>
```

Figure 136: XML Specification of Category **Formation-Work-Flow-in-RAS**

8.9 Summary

In this chapter, we illustrated a prototype design of self-configuration property, prototype design of categorical specification for the self-configuration and transformation from the categorical self-configuration property to its XML specification.

We described three scenarios regarding the self-configuration that are forming a RAS (Section 8.1), forming a RACG (Section 8.3) and forming a RAC (Section 8.5) using intelligent control loops. In addition, we presented the categorical illustration for those scenarios in section 8.2, 8.4 & 8.6 respectively using functors, natural transformations and functor categories.

Finally, the categorical specifications for the self-configuration related properties together with their XML representations were illustrated in Section 8.7 & 8.8, such as the configuration of RAE, conformation between the configurations of RAE as well as the equivalence of RAE's configurations.

Chapter 9: RASF Integration Tool

This chapter states the research activity 13) & 14) in Figure 3, which are implementation of the RASF tool and integration of the MAS implementation to the RASF tool. I had one publication [85] and one in preparation during this stage. After describing the RASF process model, the models as well as specifications of the architecture, behavior and self-* properties in RASF, we have developed a tool, RASF Integration Tool (RASFIT), for supporting the RASF process methodology.

9.1 Architecture of RASFIT

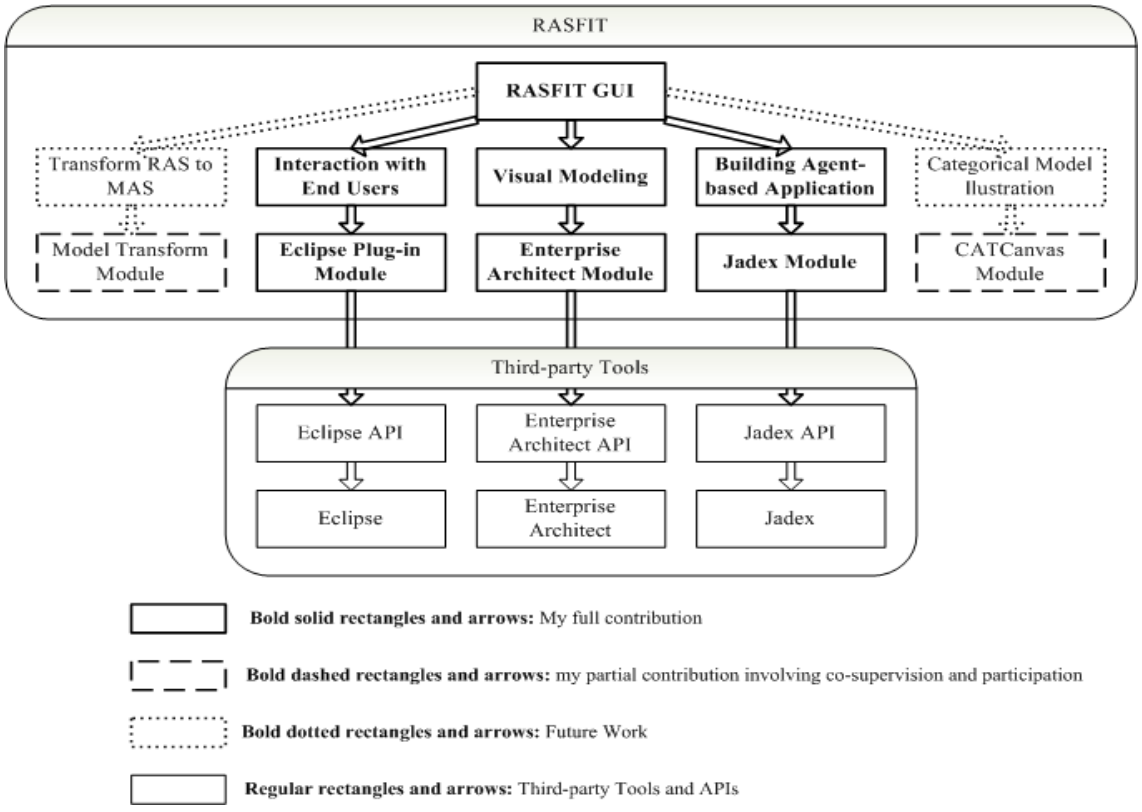


Figure 137: Architecture of RASF Integration Tool

Figure 137 depicts the architecture of RASFIT, which is an Eclipse [202] plug-in based solution that extends the Eclipse IDE with a UML design tool (Enterprise Architect [203])

for modeling and code generation, a framework for building the multi-agent applications named Jadex [204], a model transformation framework [148] to produce the multi-agent templates representing the RAS components that satisfy both reactive and autonomic properties, and a graphical tool in terms of illustrating categorical models [81]. The following figure illustrates an example of the user interface in RASFIT, and more examples can be found in Appendix F.

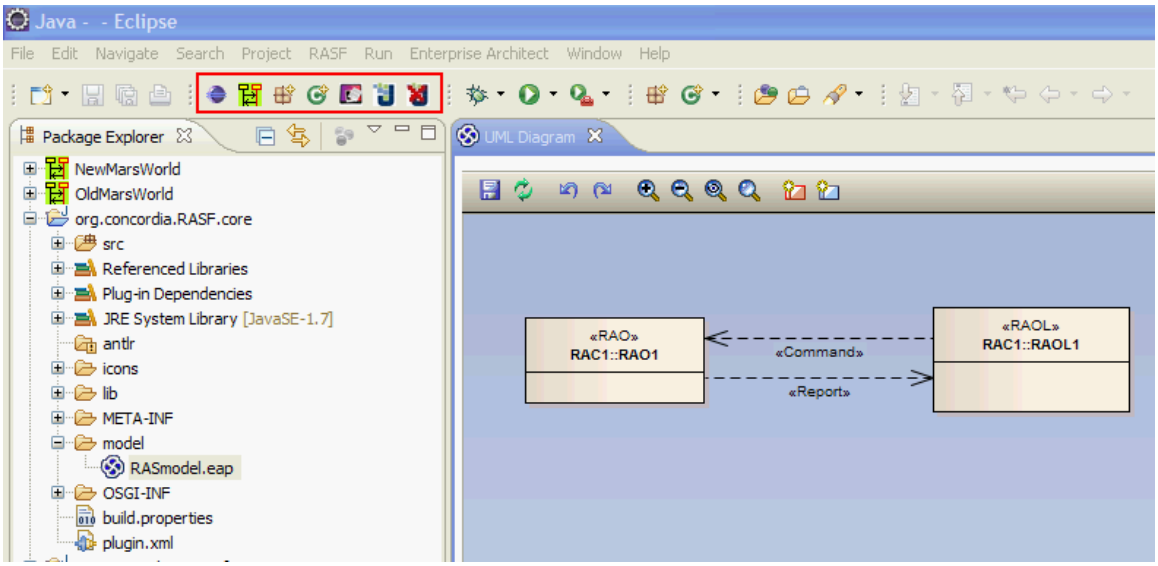


Figure 138: Toolbar Area for RASFIT

9.1.1 Eclipse Plug-in Module

The Eclipse plug-in module includes the projects "org.concordia.RASF.feature", "org. concordia.RASF.site" and some classes in the project "org.concordia.RASF.core". Figure 139 & 140 show some parts of the Eclipse plug-in module, which is responsible for the interactions from end users through the Eclipse IDE by using the Eclipse API. The users can use this module to configure RASFIT, create new RASF projects, packages and classes; they can also trigger other modules from it, such as the EA module for Phase 1 &

2 in Section 5.5, Jadex module for Phase 9 and model transformation module for Phase 3.

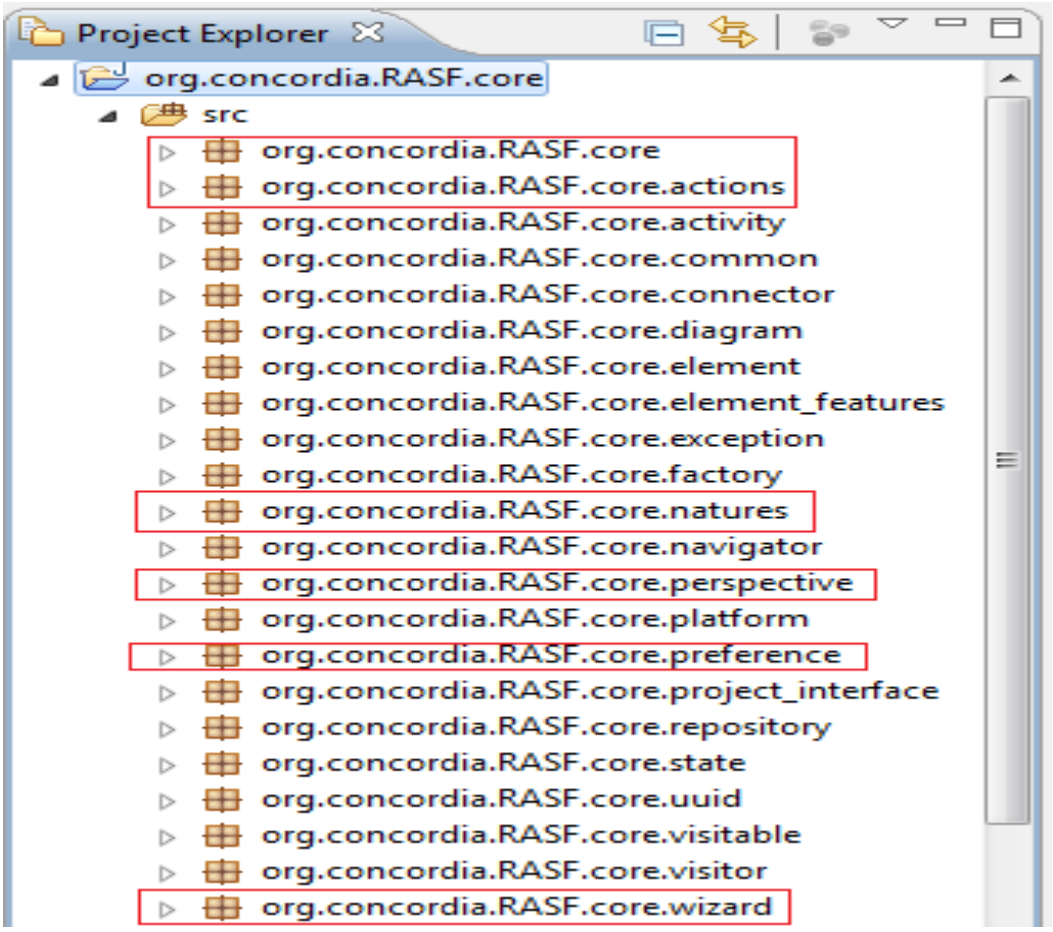


Figure 139: Part1 of the Eclipse Plug-in Module

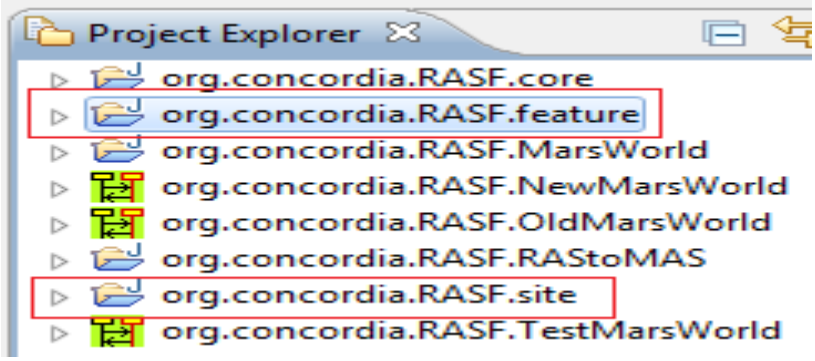


Figure 140: Part2 of the Eclipse Plug-in Module

9.1.2 EA Module

The EA module can be used for the Phase 1 & 2 we introduced in Section 5.5. This

module includes some packages and classes in the project "org.concordia.RASF. core". Figure 141 depicts some parts of the EA module that is responsible for modeling RAE of RAS through the integrated EA IDE in the Eclipse IDE by using the EA API. End users can use this module to draw UML diagrams, generate code template or XML templates and import predefined RASF model templates which are developed in the project "org.concordia.RASF.profile" (see Figure 142). Figure 143 shows the design and model of the RASF modeling profile that include stereotypes, meta-classes, RASF elements, interactions and diagrams for modeling the RAE in RAS.

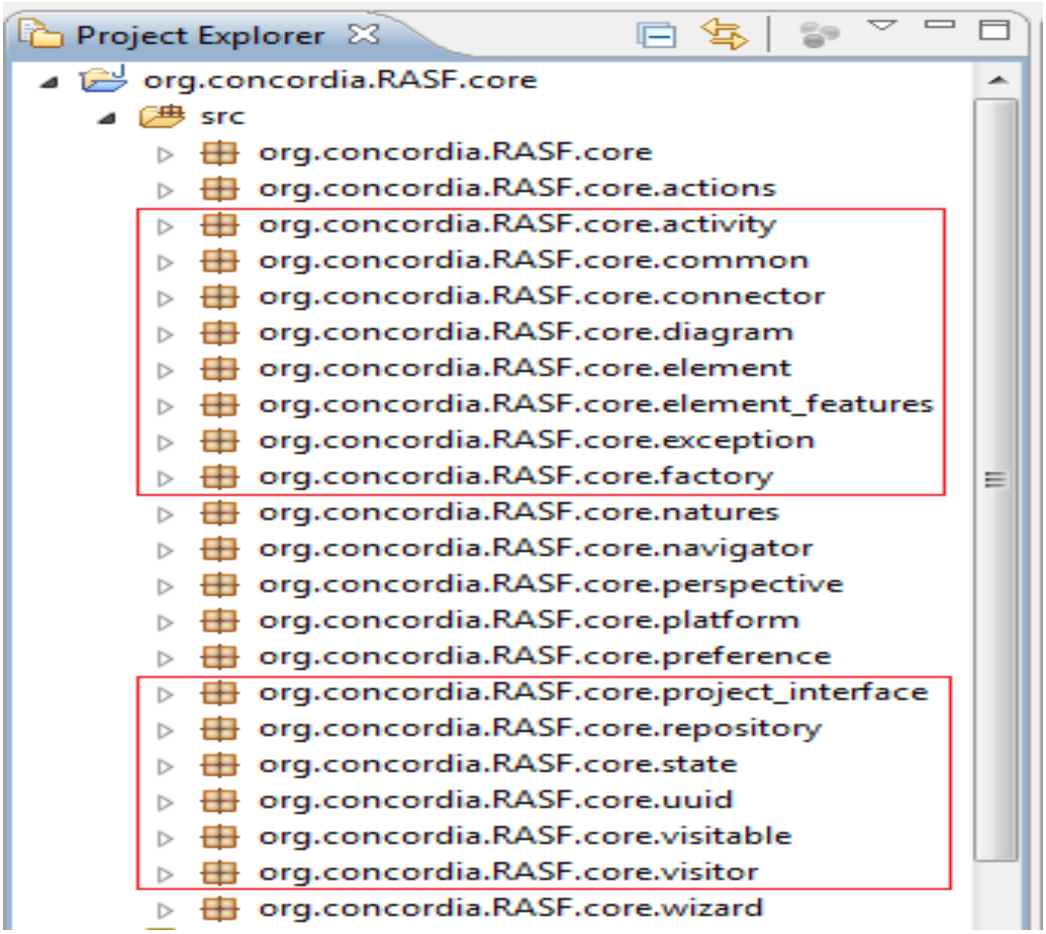


Figure 141: Part of the EA module

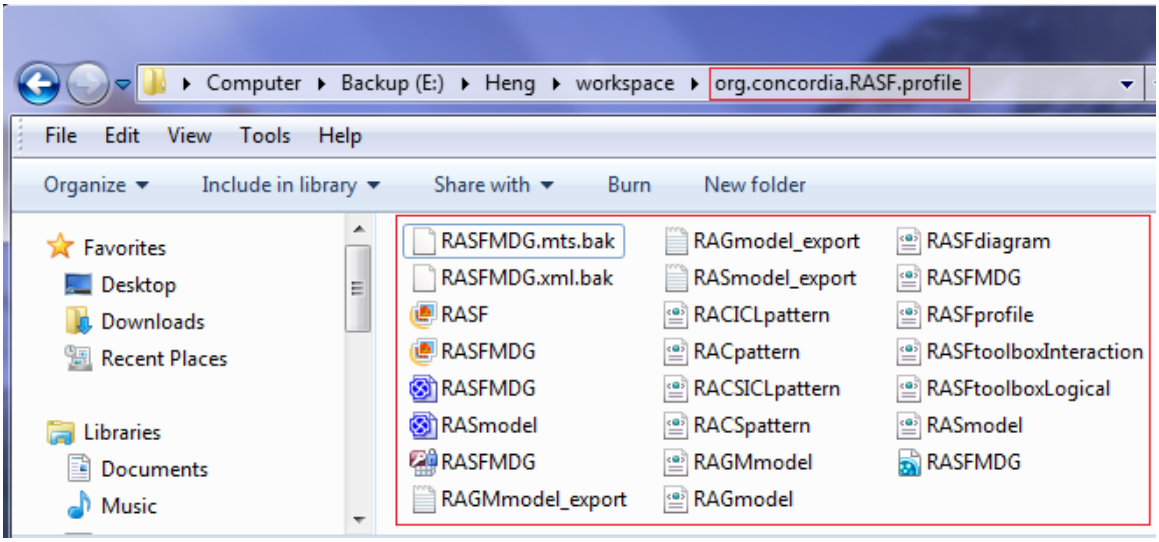


Figure 142: Project of RASF Modeling Profile

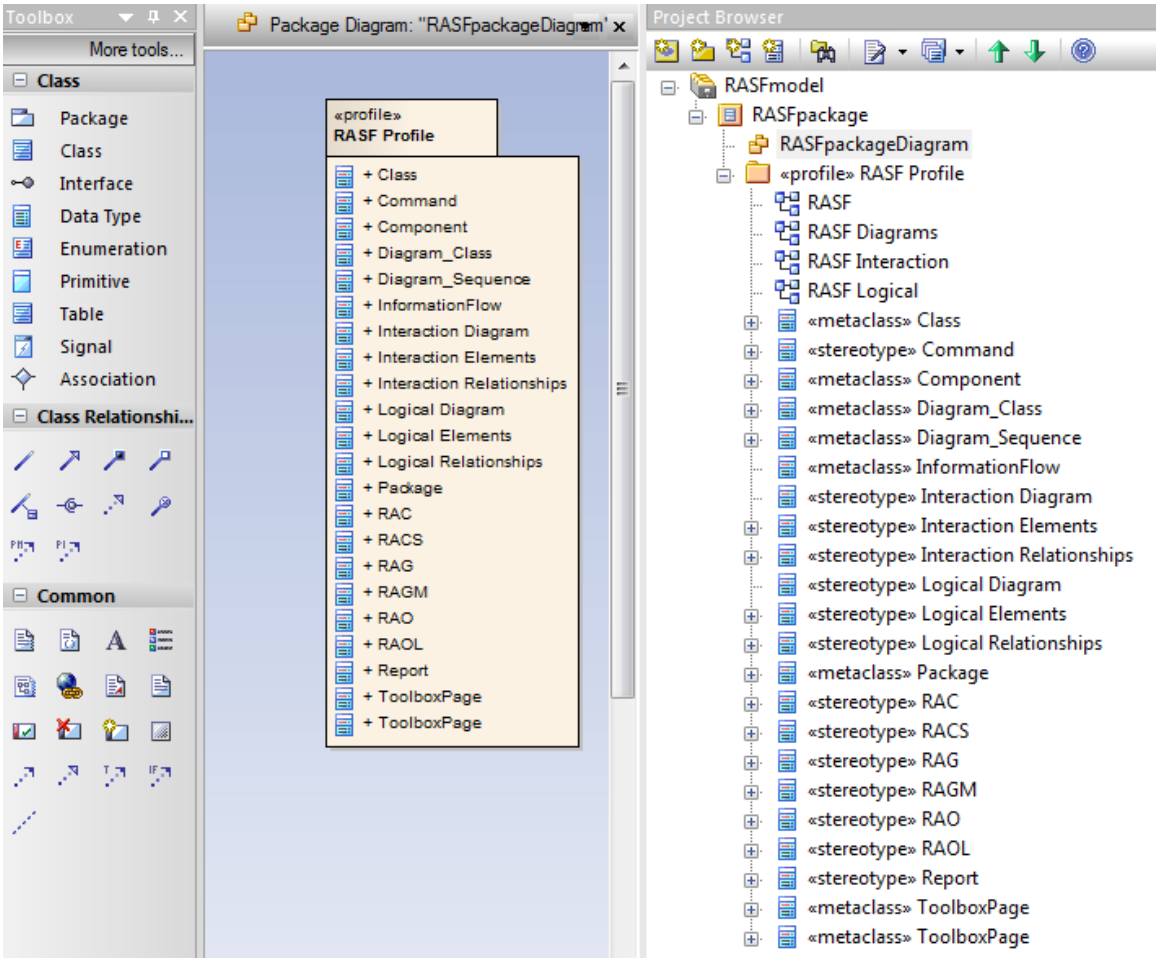


Figure 143: Design and Model of RASF Modeling Profile

Figure 144 illustrates a meta-model of the RASF elements and interactions for the RASF diagrams. For example, the stereotype <<RAGM>> with the meta-type of *RAGM* and the type of "RAGMtype" including its repository is extended from the meta-class of RASF package; the stereotype <<RACS>> with the meta-type of *RACS* and the type of "RACStype" having its repository is extended from the meta-class of RASF component; the stereotype <<RAOL>> with the meta-type of *RAOL* and the type of "RAOLtype" containing its repository is extended from the meta-class of RASF class; the stereotype <<Report>> with the meta-type of *Report* and the type of "reportType" is extended from the meta-class of RASF information flow.

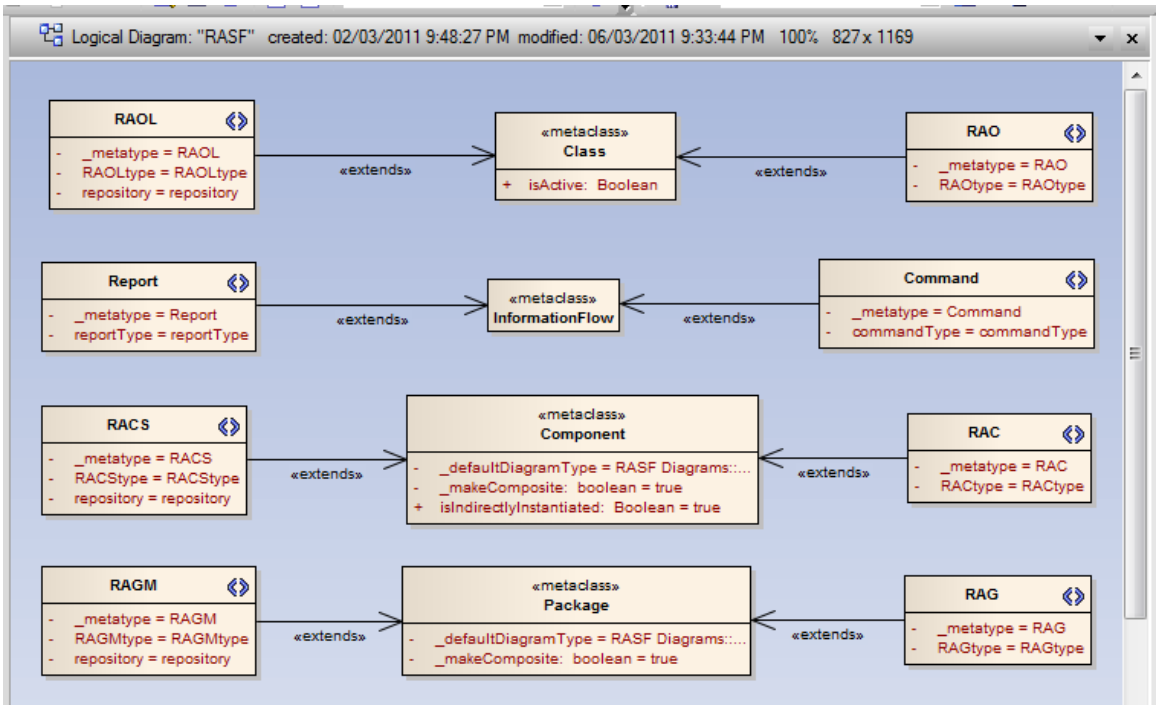


Figure 144: Meta-model of the RASF Elements and Interactions

Figure 145 depicts a meta-model of the RASF diagrams with corresponding toolbox. For instance, the RASF interaction diagram is extended from the meta-class of the EA

sequence diagram with the toolbox "RASF Interaction"; the RASF logical diagram is extended from the meta-class of the EA class diagram with the toolbox "RASF Logical".

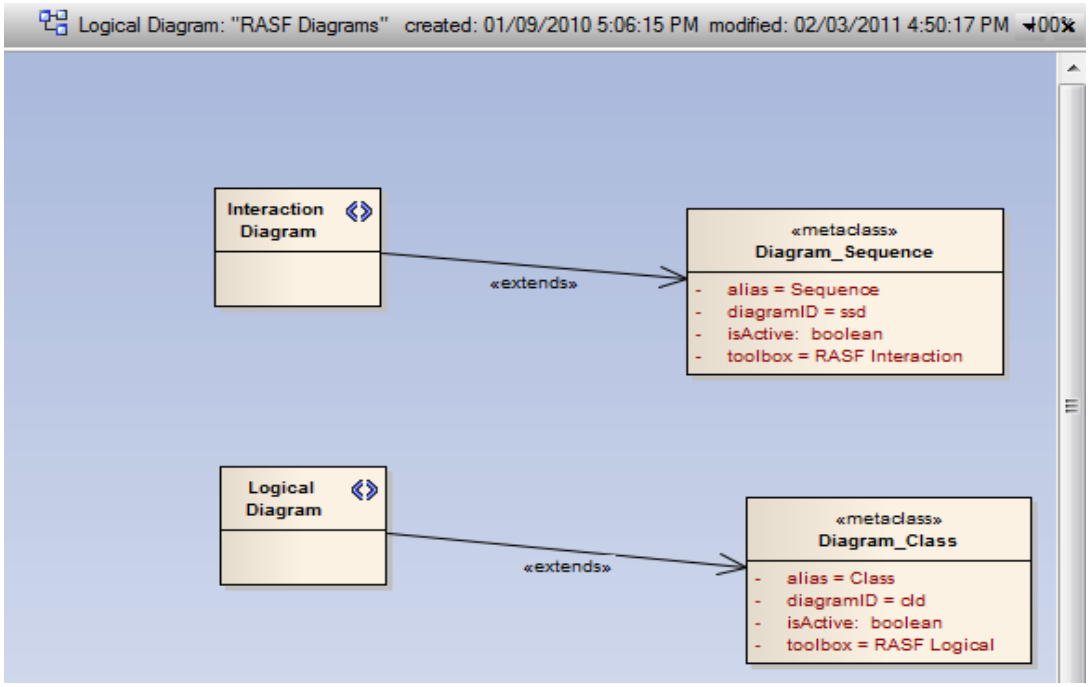


Figure 145: Meta-model of the RASF Diagrams

Figure 146 shows a meta-model of the RASF toolbox "Logical". For example, the "Logical Elements" is extended from the meta-class of the EA toolbox page with RASF elements, such as RAO, RAOL, RAC, RACS and RAGM.

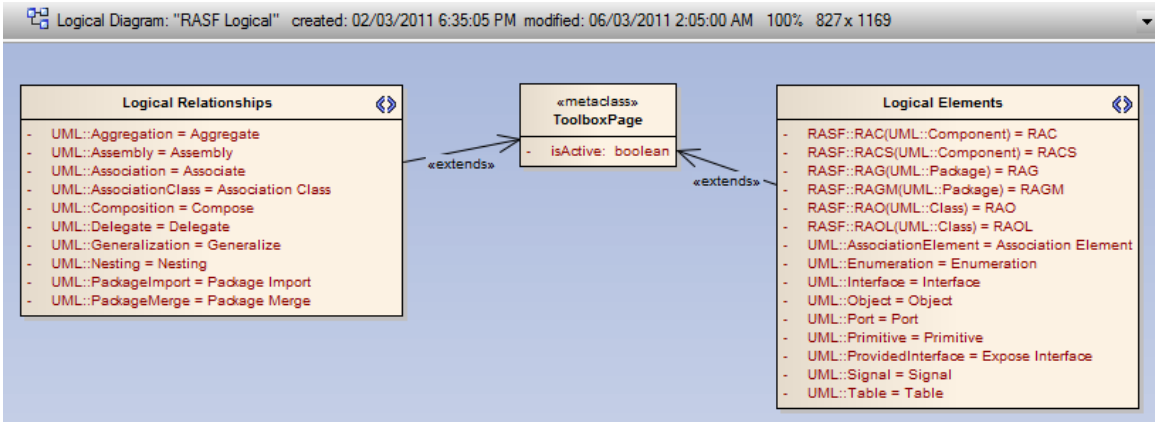


Figure 146: Meta-model of the RASF Toolbox "Logical"

9.1.3 Jadex Module

The Jadex module can be used for the Phase 9 we introduced in Section 5.5. This module includes some packages and classes in the project "org.concordia.RASF.core". Figure 147 illustrates some parts of the Jadex module which is responsible for modeling the MAS implementation from the RAS model through the integrated Jadex IDE in the Eclipse IDE by using the Jadex API. End users can use this module to create a RASF project with the Jadex nature, create new agents with their capabilities, beliefs, goals, and start the Jadex platform in terms of running or debugging the MAS applications.

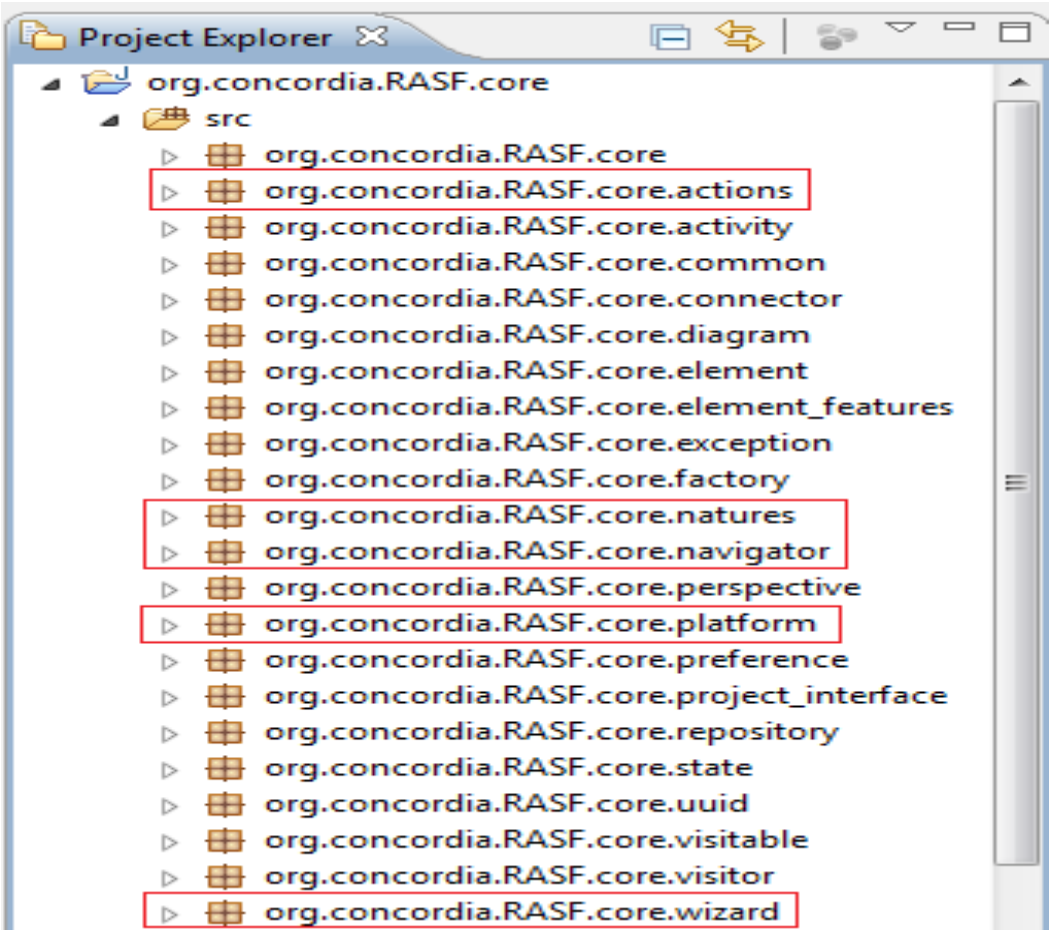


Figure 147: Part of the Jadex Module

9.1.4 Model Transformation Module

The model transformation module can be used for the Phase 3 & 4 we introduced in Section 5.5. This module includes project "org.concordia.RASF.RAStoMAS" (see Figure 148), which is responsible for transforming the RAS model represented in a XML format (generated from the EA module in Section 9.1.2) to the MAS implementation represented in format of agent definition files, defining beliefs, goals, message events, plan headers and related plan files in Java that contain the body of executable plans [148].

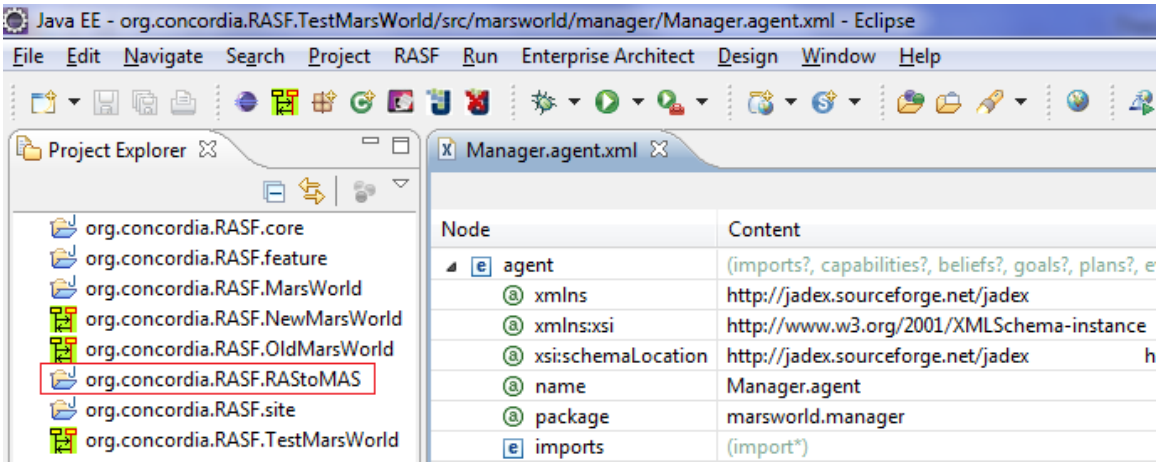


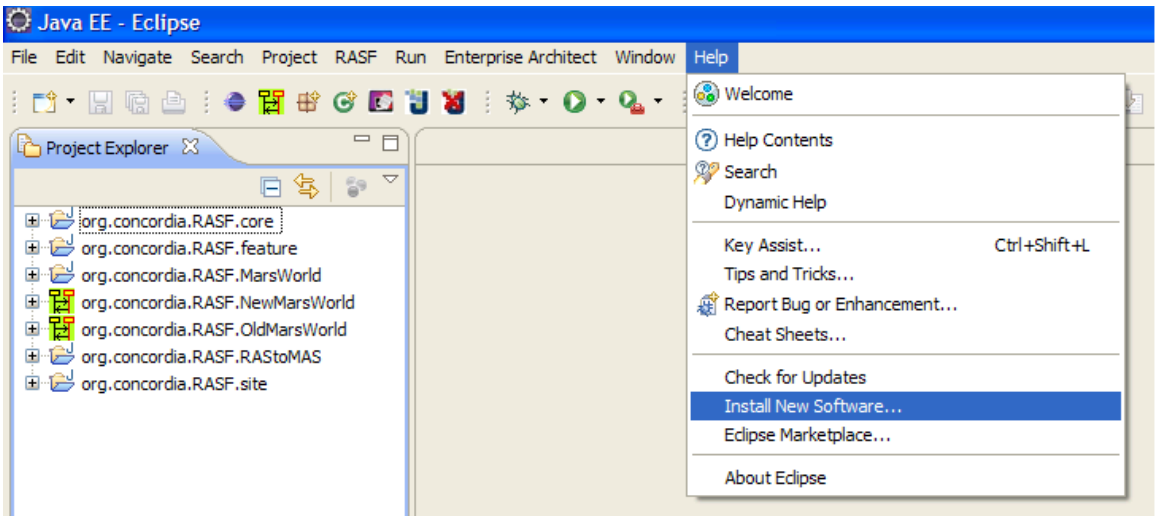
Figure 148: Part of the Model Transformation Module

9.1.5 CATCanvas Module

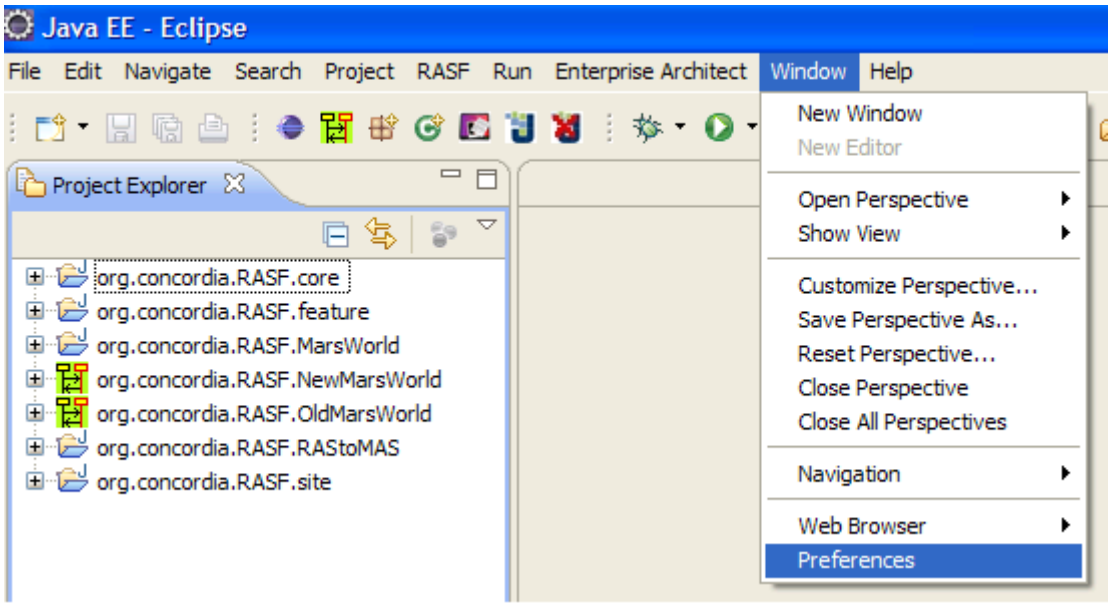
The CATCanvas module can be used for the Phase 5, 6 & 11. This module is a standalone application described in [81]. End users may use this module to graphically illustrate a RAS model represented in a XML format by importing its XML file. Moreover, the users can draw new categorical diagrams using the CATCanvas, export those diagrams to XML files and then import them to generate the RAS and MAS models.

9.2 Installation and Configuration of RASFIT

Installing RASFIT is similar to other Eclipse plug-ins as the figures in Appendix G illustrating. The following figure depicts the starting point.



After installing RASFIT successfully and restarting Eclipse, the next step is to configure RASFIT as the figures in Appendix G illustrating. The following figure depicts the starting point.

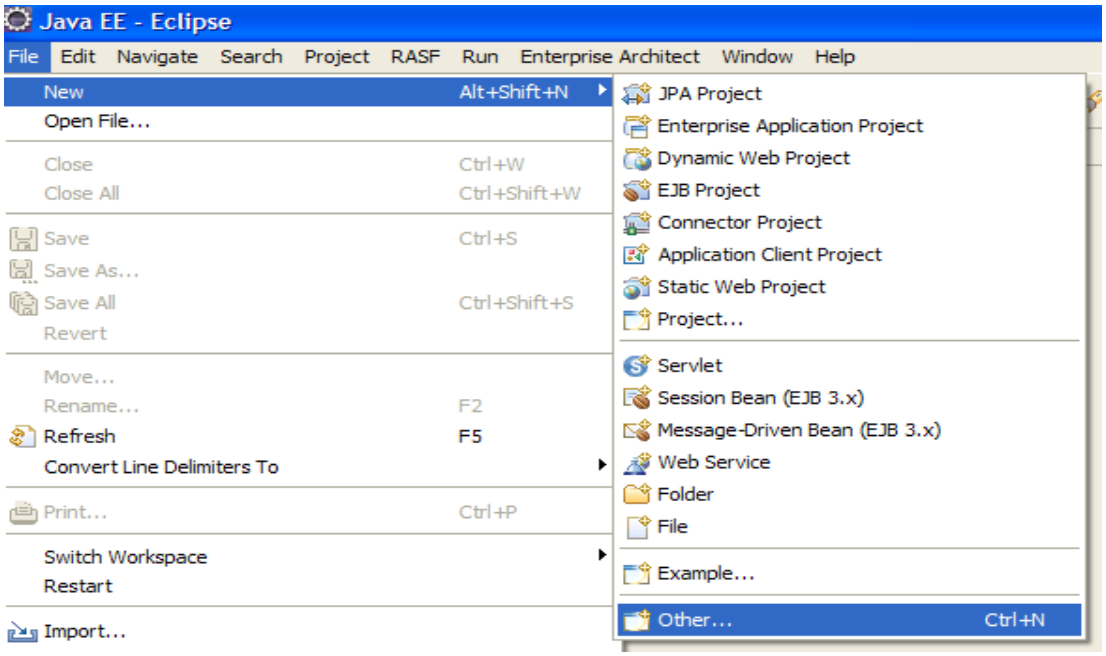


9.3 Applying RASF Methodology with RASFIT

This section describes how to use RASFIT to develop a RAS based on the RASF process model (Section 5.5) that includes modeling and specification. The model transformation and implementation will be addressed in the next chapter together with case studies. We only list the first step for each stage, and more details can be found in Appendix H.

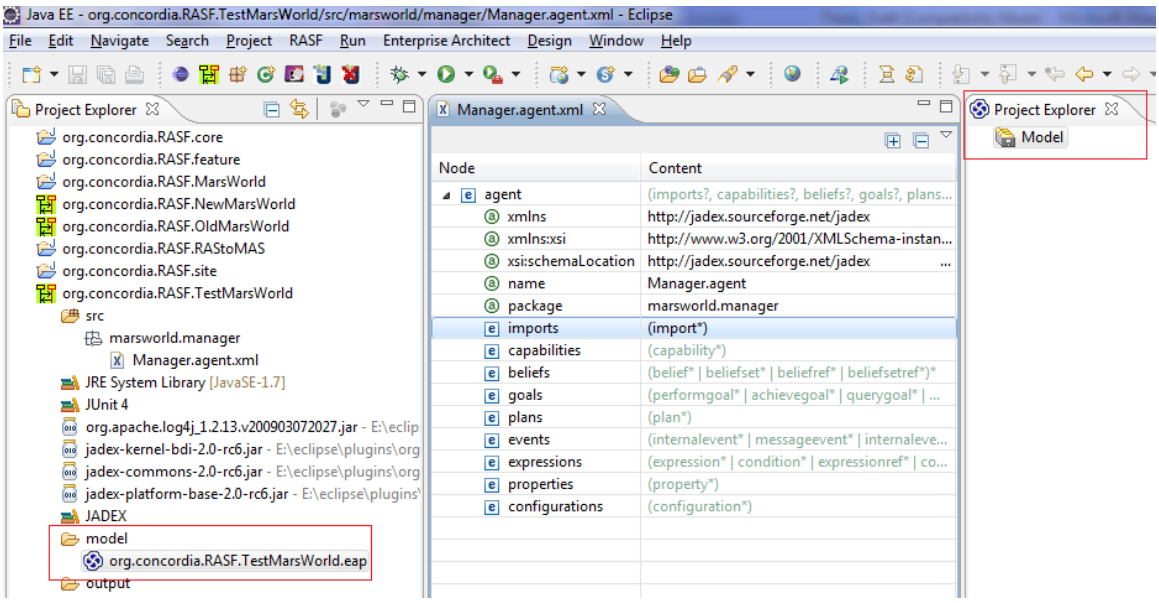
9.3.1 Creation of RASF Project

Step 1 (Phase 1 in Section 5.5): In the Eclipse IDE, click "File" → "New" → "Other…"



9.3.2 Modeling in RASF Project

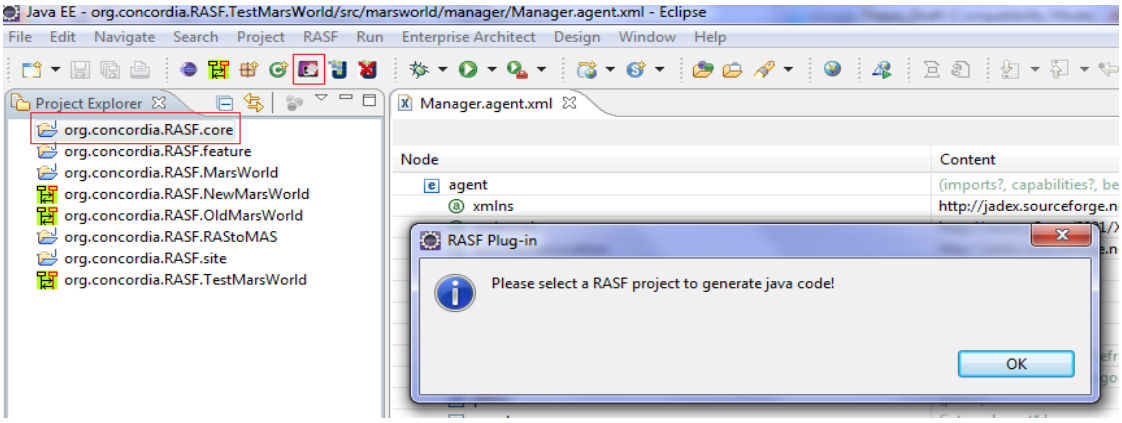
Step 11 (Phase 1): Double click the model file of Enterprise Architect (EA) under the folder "model", and the project explorer of EA is opened (see the figure below).



9.3.3 XML File and Code Template Generation in RASF Project

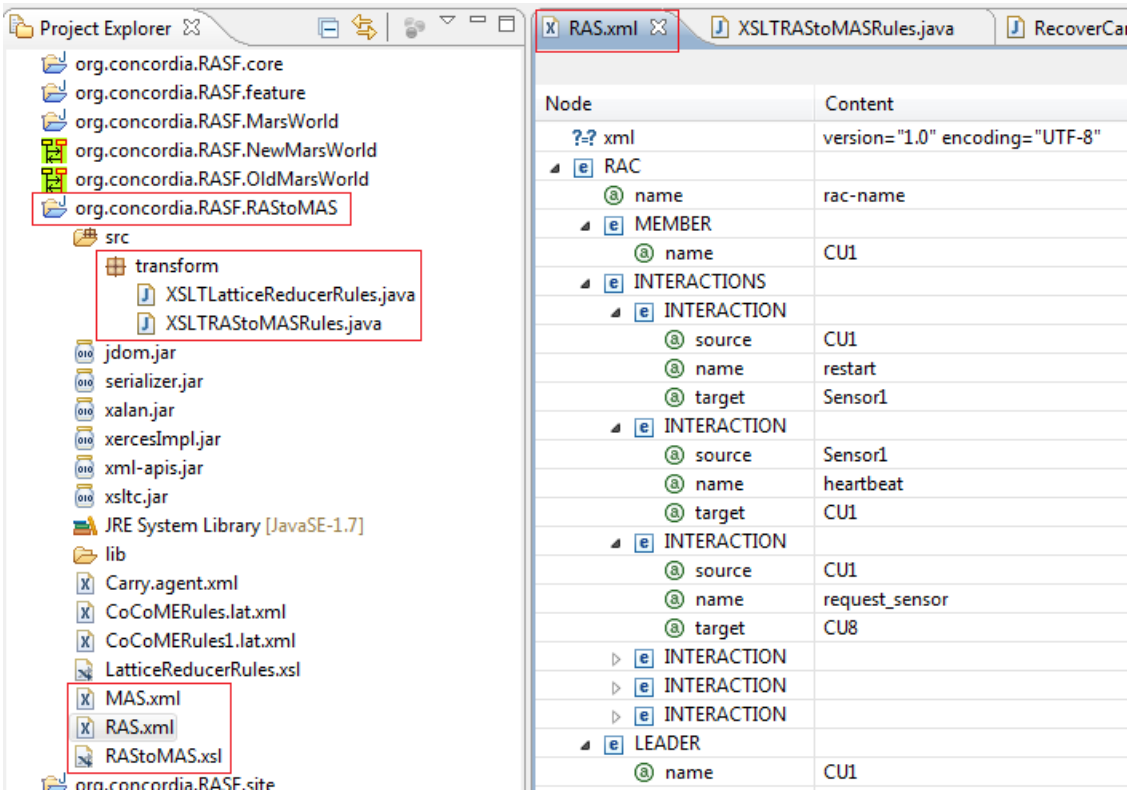
After completing the RAS modeling, we can generate the XML specification file and code template for the RAS model automatically in terms of transforming it to the MAS model as the following:

Step 22 (Phase 1 & 2): The XML specification file generation is triggered by clicking the button "RASF Code Generation" either from the RASF toolbar or from the RASF menu. The generation only applies to the RASF projects; otherwise, it will throw an error message (see the figure below).

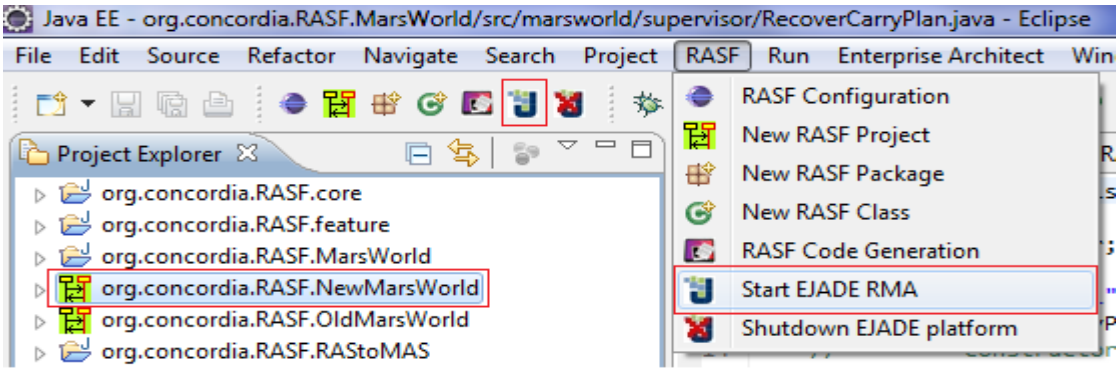


9.3.4 Model Transformation and Application Deployment

After having the XML specification files and source code templates, we can extract all necessary information from them according to the input model (see Section 5.4.2) of the model transformation module (see Section 9.1.4). The following figure depicts an example of the output model and source code for the Mars-World case study (see Section 4.1).



Step 29 (Phase 9): In order to deploy the MAS implementations that are transformed and developed from the XML specification files as well as source code templates, we can start a Jade and Jadex platform by clicking the button "Start EJADE RMA" either from the RASF toolbar or from the RASF menu (see the figure below).



9.4 Summary

In this chapter, we gave an introduction to the implementation of RASFIT and integration of the MAS implementation to RASFIT, which includes the Eclipse plug-in module, Enterprise Architect module, Jadex module, CATCanvas module as well as the model transformation module.

RASFIT is and Eclipse plug-in based solution that extends the Eclipse IDE with a UML design tool (EA) for modeling and code generation, a framework for building the multi-agent applications (Jadex), a model transformation framework to produce the multi- agent templates representing the RAS components that satisfy autonomic properties, and a graphical tool in terms of illustrating categorical models.

We presented the architecture, installation and configuration of RASFIT. We also illustrated how to apply RASF methodology with RASFIT, which includes the creation, modeling, XML file and code template generation, model transformation and application deployment in RASF project.

We will introduce our case studies to support the RASF methodology and approach in next chapter.

Chapter 10: RASF Case Studies

This chapter states the research activity 9), 12), 17) in Figure 3, which state the prototype design of self-healing and self-configuration in case studies using RASF. The background and introduction of those case studies can be found in Chapter 4. I had one publication [86] and one in preparation for this chapter.

10.1 Mars-World

In Mars-world, the objective for a group of robots is to mine ore; the mining process is composed of locating the ore, mining it, and transporting the mined ore to a home base.

10.1.1 Architecture Model of Mars-World

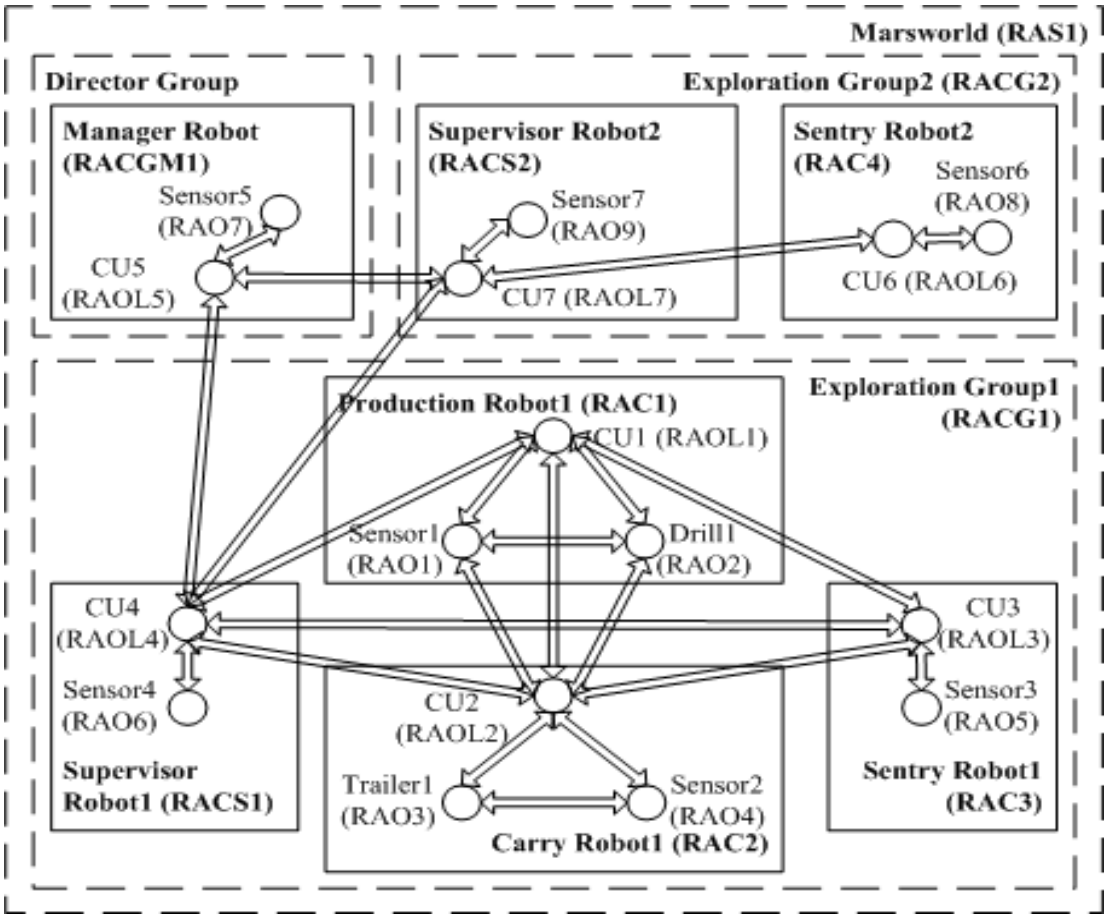


Figure 149: Example of Mars-world Modeled using RASF

Figure 149 depicts an example of the architecture model of Mars-world built from the RAS architecture model (see Figure 21 in Section 5.1) for a simplified scenario presented above (Phase 1 in Section 5.5), where every circle represents the component of a robot and each arrow specifies the communication between those components.

In this example, an exploration group (RACG1) has a *supervisor robot* (RACS1) and its backup (RACS1'), a *production robot* (RAC1), a *carry robot* (RAC2) and a *sentry robot* (RAC3). A *control unit* (RAOL) and a *sensor* (RAO1) are two common devices of each robot. Moreover, different types of robots have their particular equipments. For instance, a *production robot* has a drill (RAO2); a *carry robot* has a trailer (RAO3); a *sentry robot* has an enhanced sensor (RAO5) instead of the standard one. The figures below illustrate the specifications of production robot, exploration group and Mars-world according to the specifications of RAC (Figure 22), RACG (Figure 23) and RAS (Figure 24) in Section 5.1.2, 5.1.3 & 5.1.4.

```
RAC <Production Robot1>
 Member: <Drill1, Sensor1, CU1>
 Interaction: <(Drill1, Sensor1), (Drill1, CU1), (Sensor1, CU1),
                (Drill1, CU2), (Sensor1, CU2)>
 Leader: <CU1>
 Supervisor: <Supervisor Robot1>
 Neighbor: <Carry Robot1, Sentry Robot1>
 Repository: <Component Repository>
End RAC
```

Figure 150: Specification of Production Robot

RACG <Exploration Group1>
Member: <Production Robot1, Carry Robot1, Sentry Robot1,
Supervisor Robot1>
Interaction: <(CU1, CU4), (CU2, CU4), (CU3, CU4),
(CU1, CU2), (CU1, CU3), (CU2, CU3)>
Supervisor: <Supervisor Robot1>
Manager: <Manager Robot1>
Neighbor: <Exploration Group2>
Repository: <Group Repository>
End RACG

Figure 151: Specification of Exploration Group

RAS <Marsworld>
Member: <Exploration Group1, Exploration Group2, Director Group>
Interaction: <(CU4, CU5), (CU4, CU7), (CU7, CU5)>
Manager: <Manager Robot>
User Console: <Ground Station>
Neighbor: <Other Marsworld>
Repository: <System Repository>
End RAS

Figure 152: Specification of Mars World

The following figures illustrate some examples of using the RASFIT we introduced earlier in Chapter 9 in terms of specifying the architecture model of Mars-world.

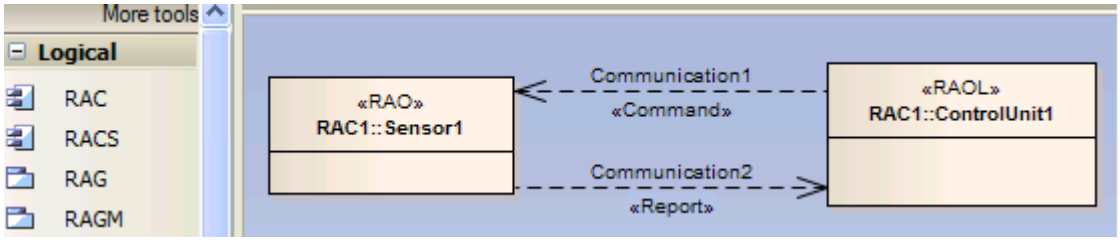


Figure 153: Class Diagram of Sentry Robot1

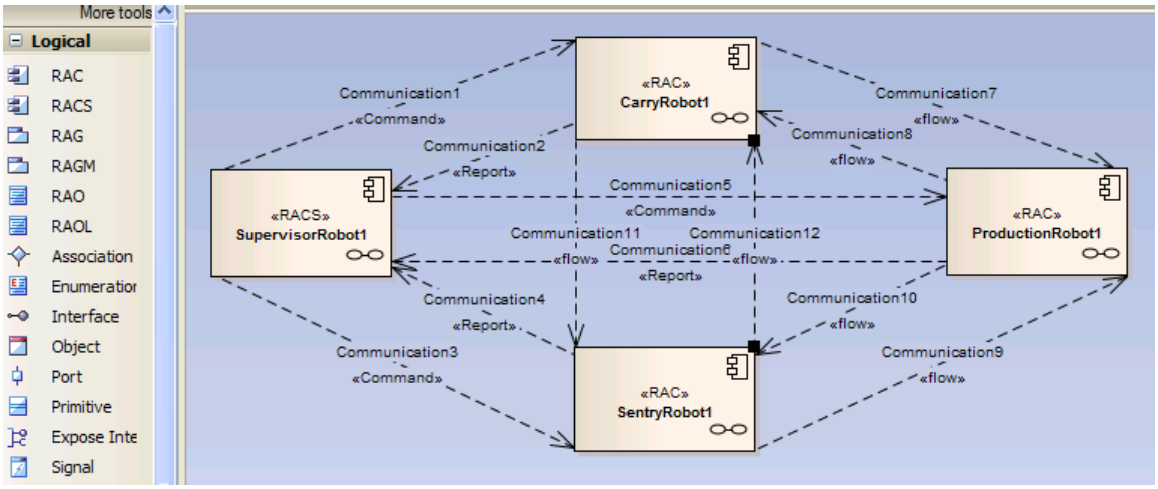


Figure 154: Component Diagram of Exploration Group1

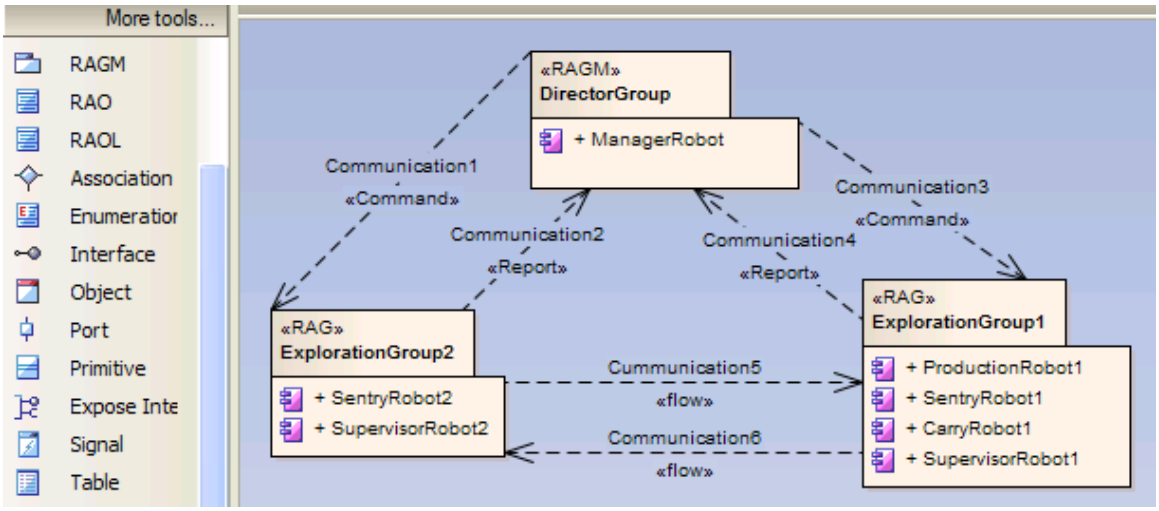


Figure 155: Package Diagram of Mars-world

10.1.2 Self-Healing in Mars-World

Crashed Sensor. After *sensor1* is started by *CU1*, it starts to send its heartbeat messages to *CU1* every *t* ticks, while *CU1* is in *Monitor* state, monitoring the status of *sensor1*. If *CU1* receives the heartbeat messages from *sensor1*, *NoChange* event keeps it in *Monitor* state; otherwise, *Sensor1-Crashed* event is triggered and *CU1* transits to *Analyze* state, while a time constraint variable (*TCvar1*) is initialized to work as a local clock in terms

of time constraints on each transition of the intelligent control loop. After *CU1* enters *Analyze* state, it sends the *Restart* message to *sensor1* in *t0* ticks. If *sensor1* is recoverable, *NoAction* event occurs and *CU1* goes back to *Monitor* state, while *TCvar1* is reset; otherwise, *CU1* transits to *Plan* state triggered by *HasAction* event in *t1* ticks. When *CU1* is in *Plan*, it broadcasts *RequestSensor1* messages with type information of *sensor1* to all other *robots* for replacing it by an available *sensor* which is equivalent to *sensor1*, such as *sensor8*, since equivalent objects behave in the same way (see Property 7.7.3). If at least one *sensor* is available for switching, *CU1* chooses *Substitute* plan and transits to *Execute* state triggered by *Substitute* event in *t2* ticks; otherwise, it selects *Take-over* plan and enters *Execute* triggered by *Take-over* event in *t2* ticks. In this plan, *drill1* takes the responsibilities of *sensor1* by its backup sensor and works as the product object of original *drill1* and *sensor1*, because of their synchronous communication. When *CU1* is in *Execute* state and *Substitute* plan is applicable, *CU1* sends a *register* message to *sensor8* and then initializes it to the status of *sensor1* based on the checkpoint made before. When *Take-over* plan is applicable, *CU1* sends a *Take-over* message to *drill1* and update it to the status of synchronous product machine of *drill1* and *sensor1* based on the checkpoint made before. After executing the plan, *CU1* validates the original as well as evolutionary behaviors of *production robot1* based on their categorical specifications. If they are equivalent, *ActionDone* event occurs and *CU1* transits to *Monitor* in *t3* ticks; otherwise, *ActionFailed* event keeps it in *Execute* for the user intervention from *ground station* through *supervisor robot1* and *manager robot* (see the figures below).

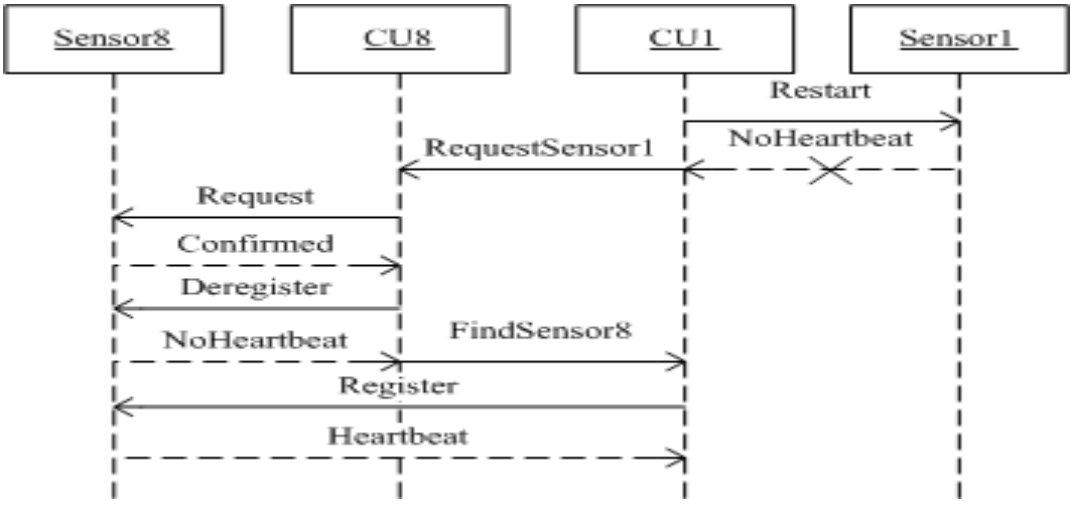


Figure 156: Sensor Substitution Work Flow in Production Robot

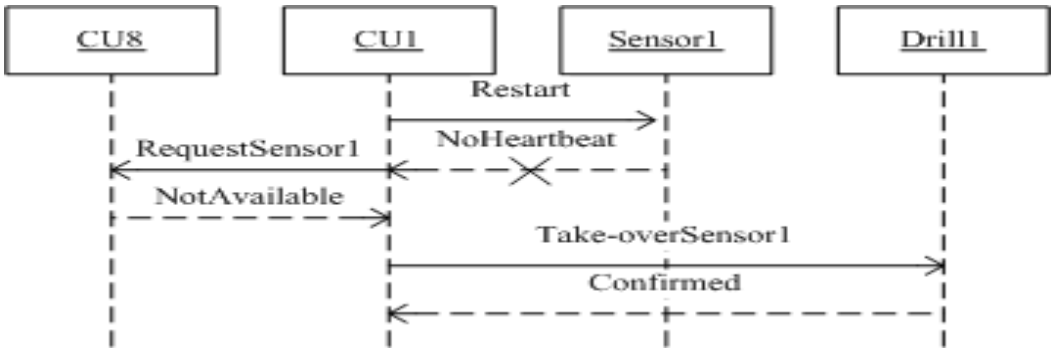


Figure 157: Sensor Take-over Work Flow in Production Robot

When both *sensor1* and *drill1* are crashed at the same time, *CU1* tries to restart them first. If neither of them can be recovered, *CU1* broadcasts messages to all other *robots* for requesting the equivalent *sensor* and *drill* of them; otherwise, the remaining process is the same as the illustration before. If none of *sensor* or *drill* is available, *CU1* broadcasts messages to all other *robots* for requesting the equivalent *production robot* of the original one, or the description before is applicable for remaining process (Phase 1 in Section 5.5).

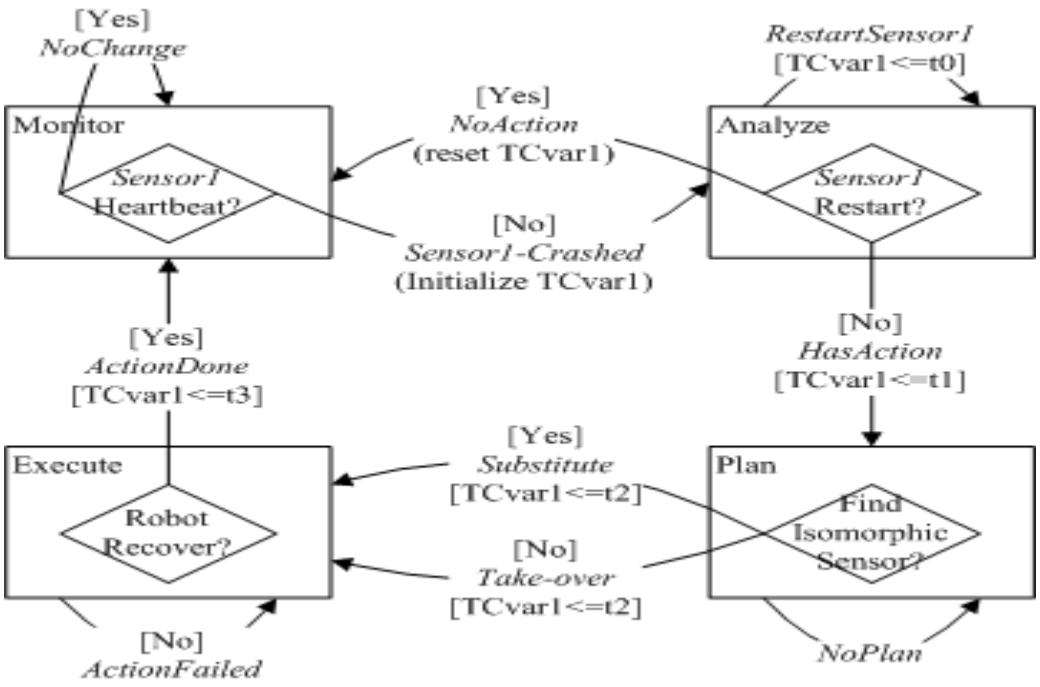


Figure 158: Intelligent Control Loop of Control Unit in Production Robot

Crashed Control Unit. As the scenario above, after a *sentry robot* is started by its *supervisor robot*, its control unit (*CU3*) begins to send heartbeat messages to *supervisor robot*'s control unit (*CU4*) every *t* ticks. Figures 159 and 160 show the work flows of substituting or taking over crashed *CU3*. If *CU1*, *CU2* and *CU3* are crashed at the same time, *CU4* tries to restart them first. If none of them can be restarted, *CU4* can broadcast messages to all other *robots* in terms of requesting the equivalent objects of *CU1*, *CU2* and *CU3*; otherwise, the remaining process is the same as the description before. If none of *CU* is available for substituting, *CU4* may broadcast messages to all other *robots* for requesting the equivalent *production robot*, *carry robot* as well as *sentry robot* of the original ones; otherwise, the illustration before may indicate the remaining process.

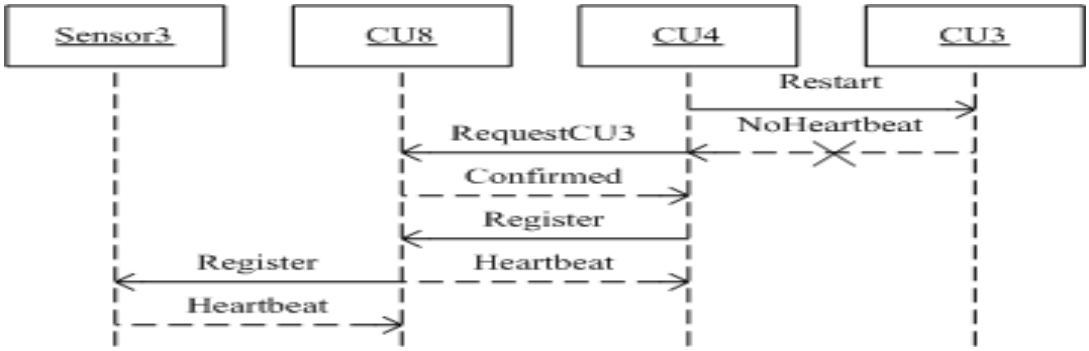


Figure 159: Control Unit Substitution Work Flow in Sentry Robot

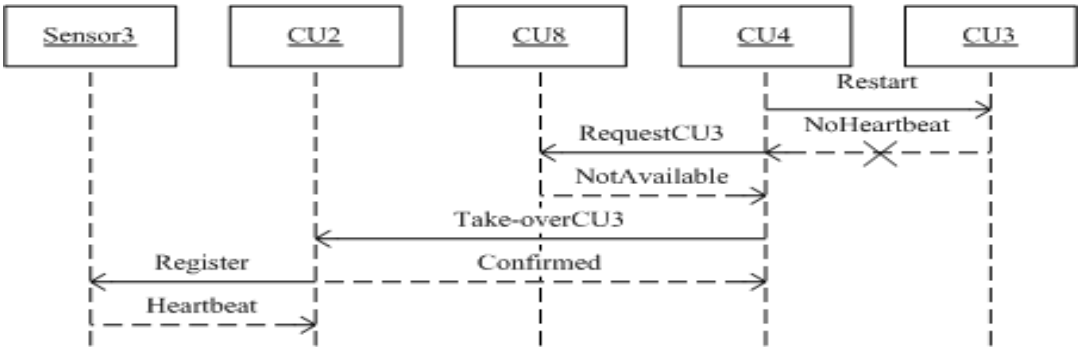


Figure 160: Control Unit Take-over Work Flow in Sentry Robot

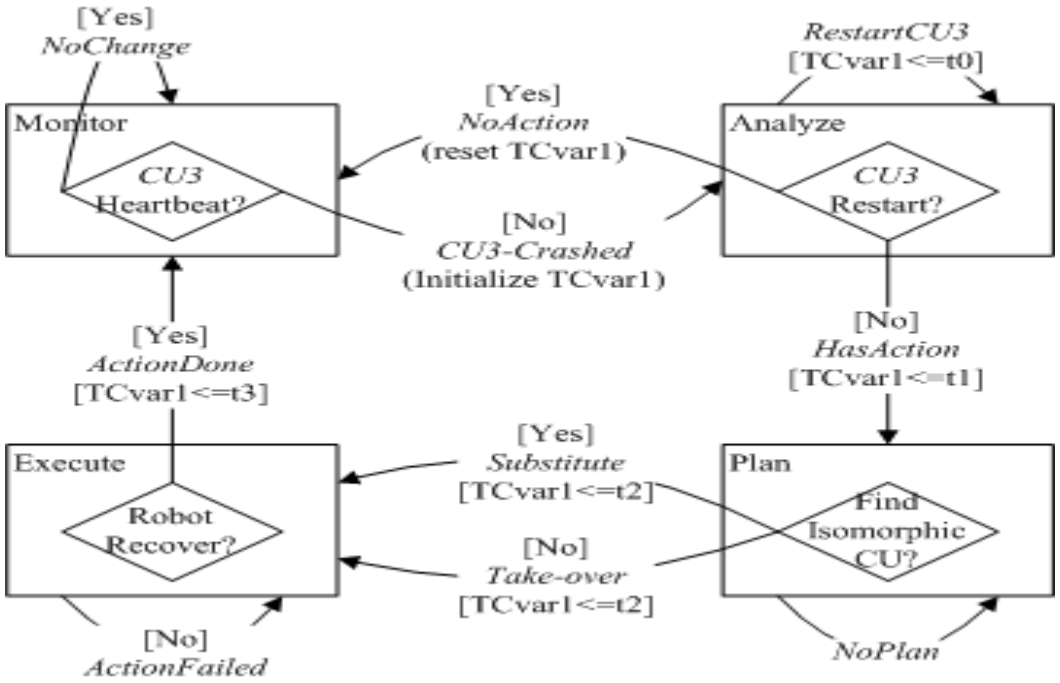


Figure 161: Intelligent Control Loop of Control Unit in Supervisor Robot

Crashed Robot. Similarly as the scenarios discussed above, after a *specialist robot* is started by a *supervisor robot*, it begins to send its heartbeat messages to the *supervisor robot*. If any part of the *specialist robot* is crashed and it cannot be restarted, substituted, or took over, the *supervisor robot* identifies it as a crashed *robot* and broadcasts messages to all other *specialist robots* for requesting an equivalent *robot* of the original one (see Figure 162). If none of the *specialist robot* is available for substituting, the *supervisor robot* waits for the user intervention from *ground station* through *manager robot* (see Figure 163, 164).

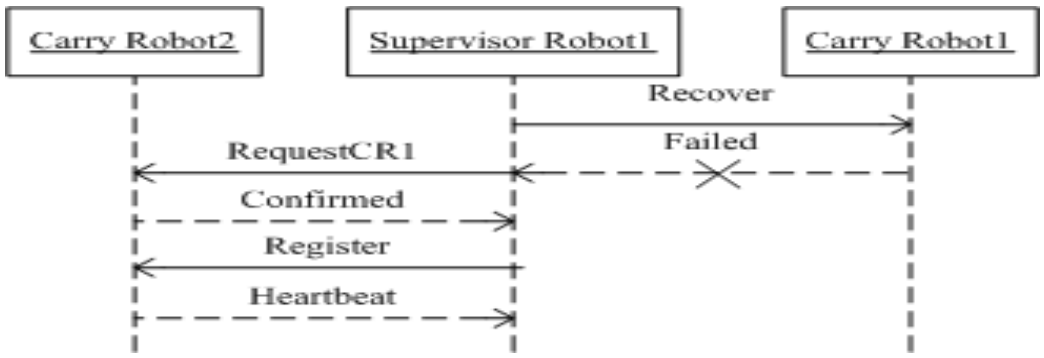


Figure 162: Carry Robot Substitution Work Flow in Exploration Group

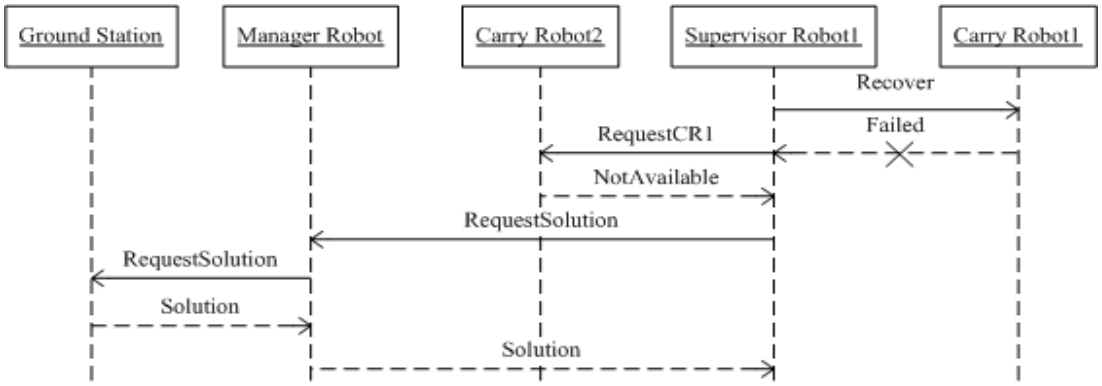


Figure 163: User Intervention Request Work Flow in Exploration Group

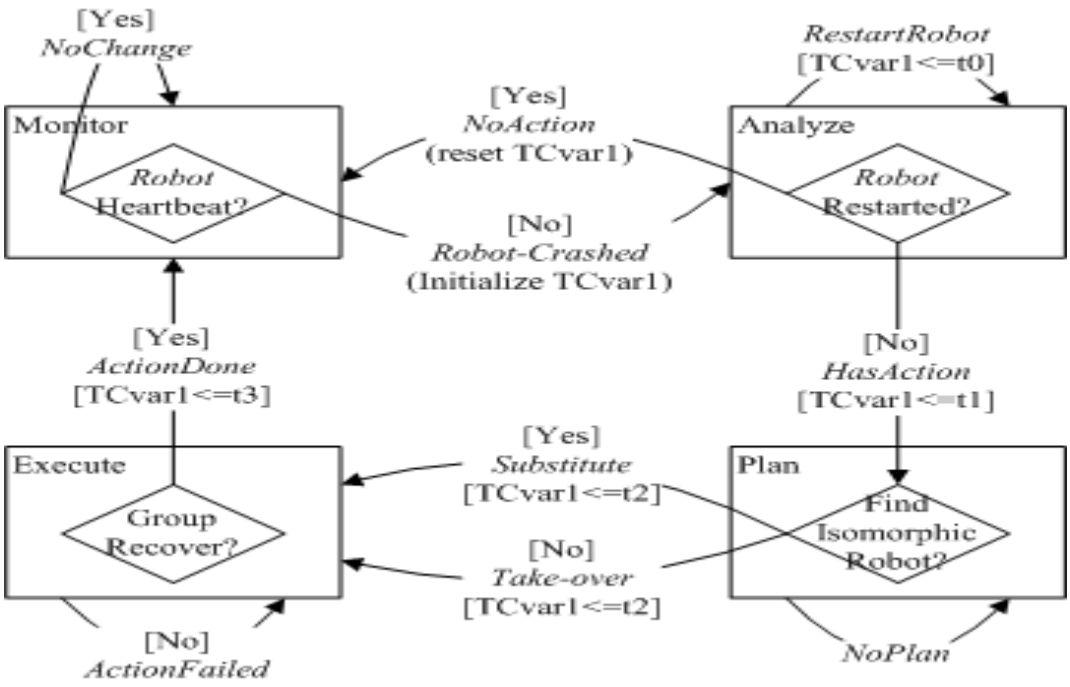


Figure 164: Intelligent Control Loop of Control Unit in Manager Robot

10.1.3 Self-Configuration in Mars-World

Forming Mars-world. After receiving the task of forming a Mars-world from Console, *manager robot* starts to create *supervisor robots* and establish corresponding connections among them based on the composition rules and communication protocols specified by the index category **Mars-world-Formation**. Figure 165 depicts an example of forming the category **Mars-world** from its index category **Mars-world-Formation**. Figure 166 illustrates the detailed object as well as morphism mappings from **Mars-world** to **Mars-world-Formation**. The *manager robot* is initialized and validated by an initialization manager in the Console (Phase 1 in Section 5.5).

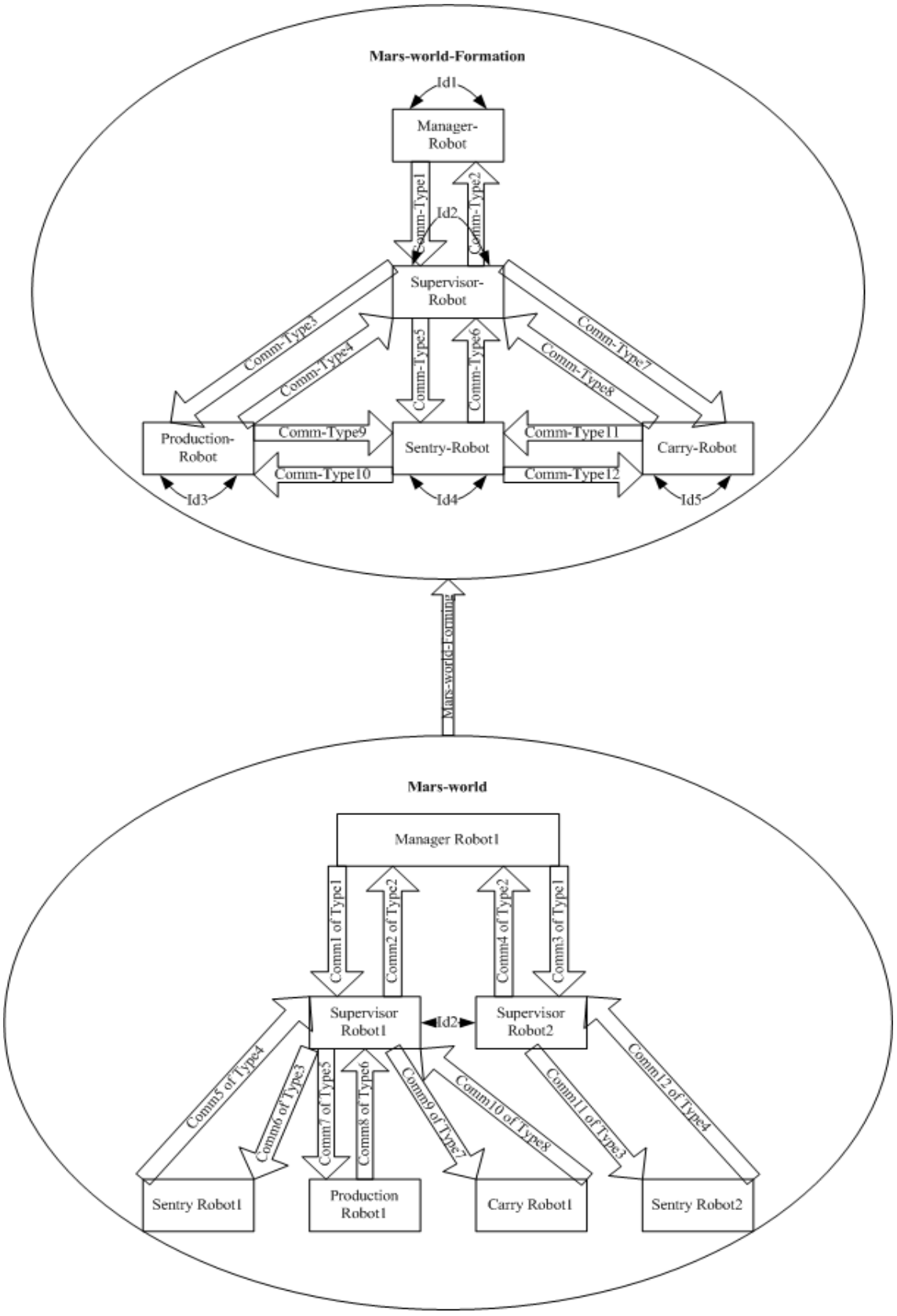


Figure 165: Example of Forming **Mars-world** from **Mars-world-Formation**

Object Mapping	Morphism Mapping (RAS-Formation <- RAS1)
Manager Robot1 -> Manager Robot	Comm1->Comm-Type1, Comm2->Comm-Type2
Supervisor Robot1 -> Supervisor Robot	Comm3->Comm-Type1, Comm4->Comm-Type2
Supervisor Robot2 -> Supervisor Robot	Comm5->Comm-Type3, Comm6->Comm-Type4
Sentry Robot1 -> Sentry Robot	Comm7->Comm-Type5, Comm8->Comm-Type6
Production Robot1 -> Production Robot	Comm9->Comm-Type7, Comm10->Comm-Type8
Carry Robot1 -> Carry Robot	Comm11->Comm-Type3, Comm12->Comm-Type4
Sentry Robot2 -> Sentry Robot	Supervisor Robot1 – Supervisor Robot2 -> Identity-Mor(Supervisor Robot)

Figure 166: Object and Morphism Mapping of Functor Mars-world-Forming

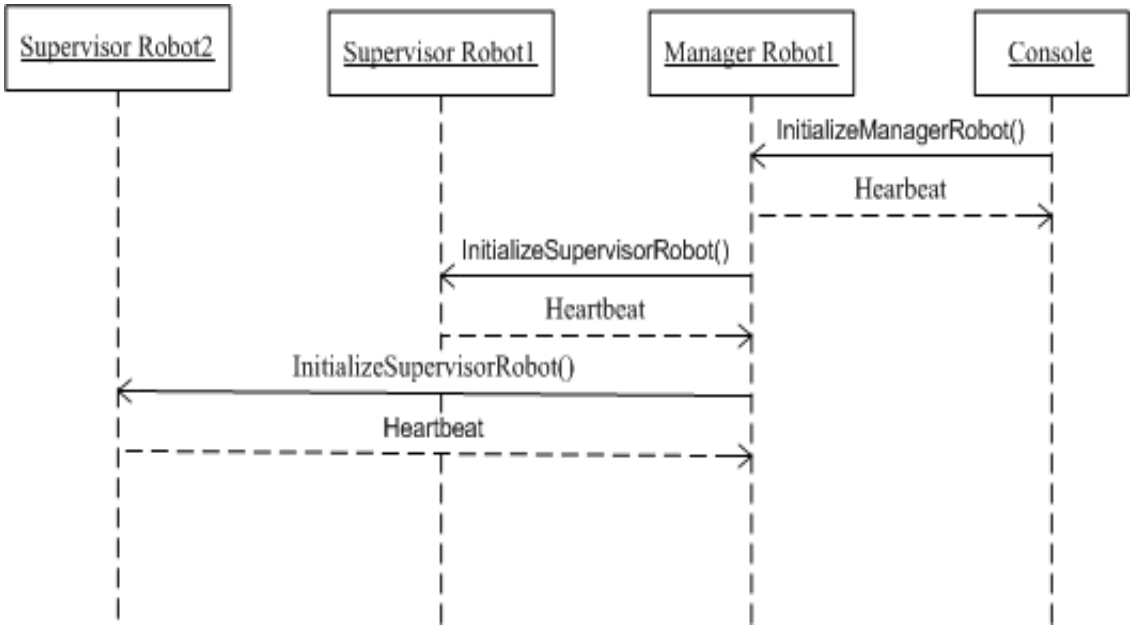


Figure 167: Formation Work Flow in Mars-world

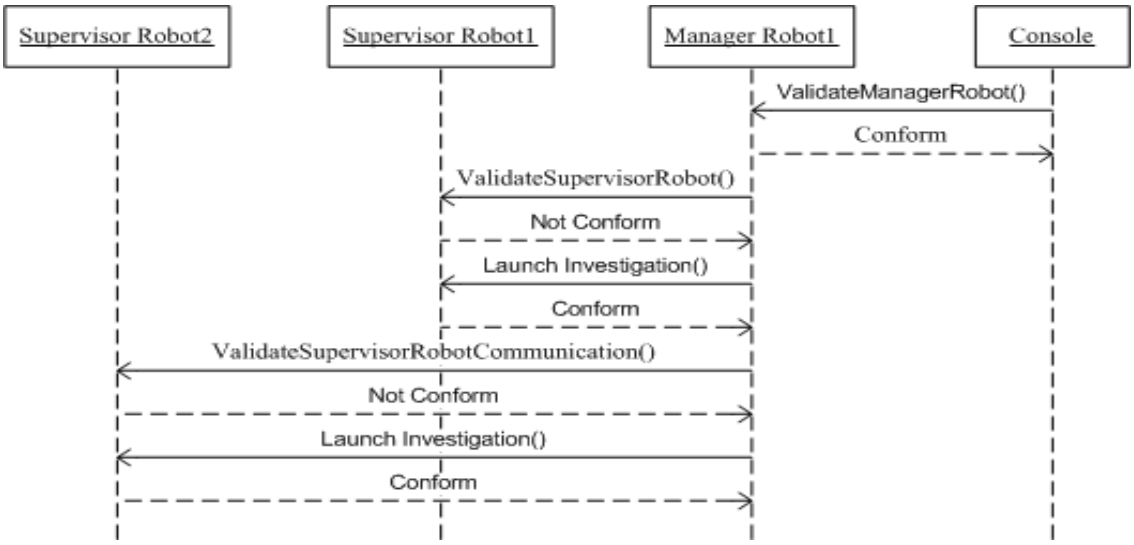


Figure 168: Self-Configuration Work Flow in Mars-world

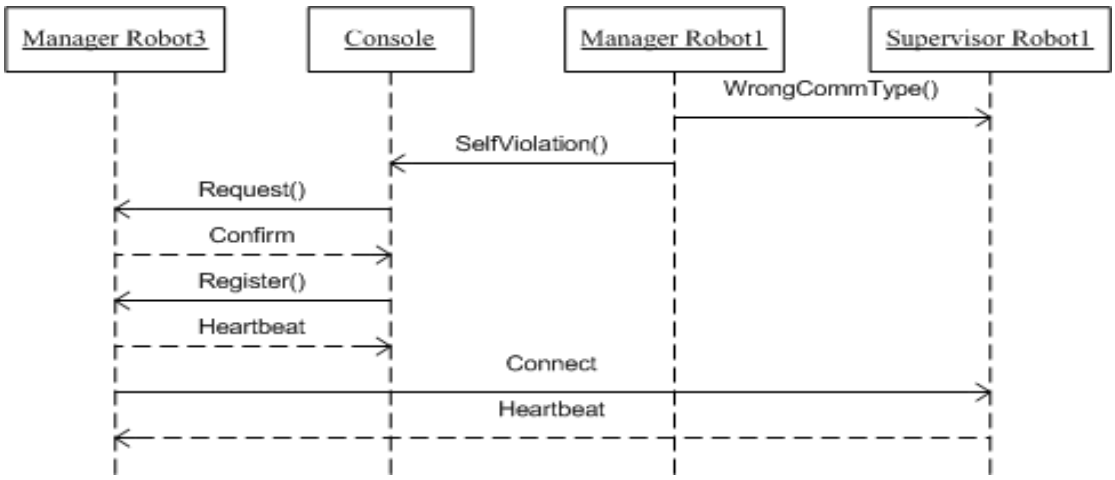


Figure 169: Self-Configuration Work Flow of Substitution in Mars-world

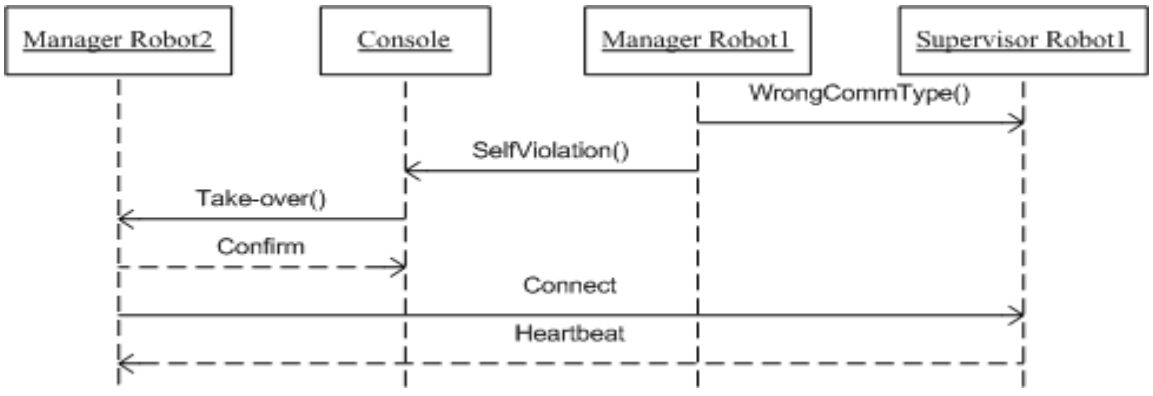


Figure 170: Self-Configuration Work Flow of Take-over in Mars-world

After *manager robot1* initializes its *supervisor robots* according to the requirements from *Console* as well as the capabilities of those *supervisor robots*, it validates the configuration of those *supervisor robots* against their types every *t* ticks, while *manager robot1* is in the first state of its intelligent control loop (monitoring). If the configuration of those *supervisor robots* conforms to their types, composition rules and communication protocols (see Property 8.7.4), *NoViolation* event keeps *manager robot1* in *Monitor* state; otherwise, *NeedInvestigation* event is triggered and *manager robot1* transits to *Analyze* state, while a time constraint variable (*TCvar1*) is initialized to work as a local clock in terms of time constraints on each transition of the intelligent control loop. The value of *TCvar1* is *t0*, *t1*, *t2*, *t3*... where $t0 < t1 < t2 < t3$.

After *manager robot1* enters the *Analyze* state, 1) it sends the *Restart* message to *supervisor robot1* in *t0* ticks where the violation is caused by incorrect supervisor robot type or incorrect communication type from the *supervisor robot1* to *manage robot1*. If *supervisor robot1* conforms to its type and communication type, *NoAction* event occurs and *manager robot1* goes back to *Monitor* state, while the *TCvar1* is reset; otherwise, *manager robot1* transits to *Plan* state triggered by *LaunchSelfHealing* event in *t1* ticks. 2) If the violation is caused by incorrect communication type from other supervisor robot (*supervisor robot3*) to *supervisor robot1*, *manager robot1* sends the *Restart* message to *supervisor robot3*. If the communication conforms to its type, *NoAction* event occurs and *manager robot1* goes back to the *Monitor* state, while the *TCvar1* is reset; otherwise, *manager robot1* transits to *Plan* state triggered by the *LaunchSelfHealing* event within *t1*

ticks. 3) If the violation is caused by the incorrect communication type from *manager robot1* to *supervisor robot1*, *manager robot1* resets that communication. If it conforms to the correct one specified in the index category **Mars-world-Formation** (see Property 8.7.4), *NoAction* event occurs and *manager robot1* goes back to the *Monitor* state, while the *TCvar1* is reset; otherwise, *manager robot1* transits to *Plan* state triggered by the *LaunchSelfHealing* event in *t1* ticks.

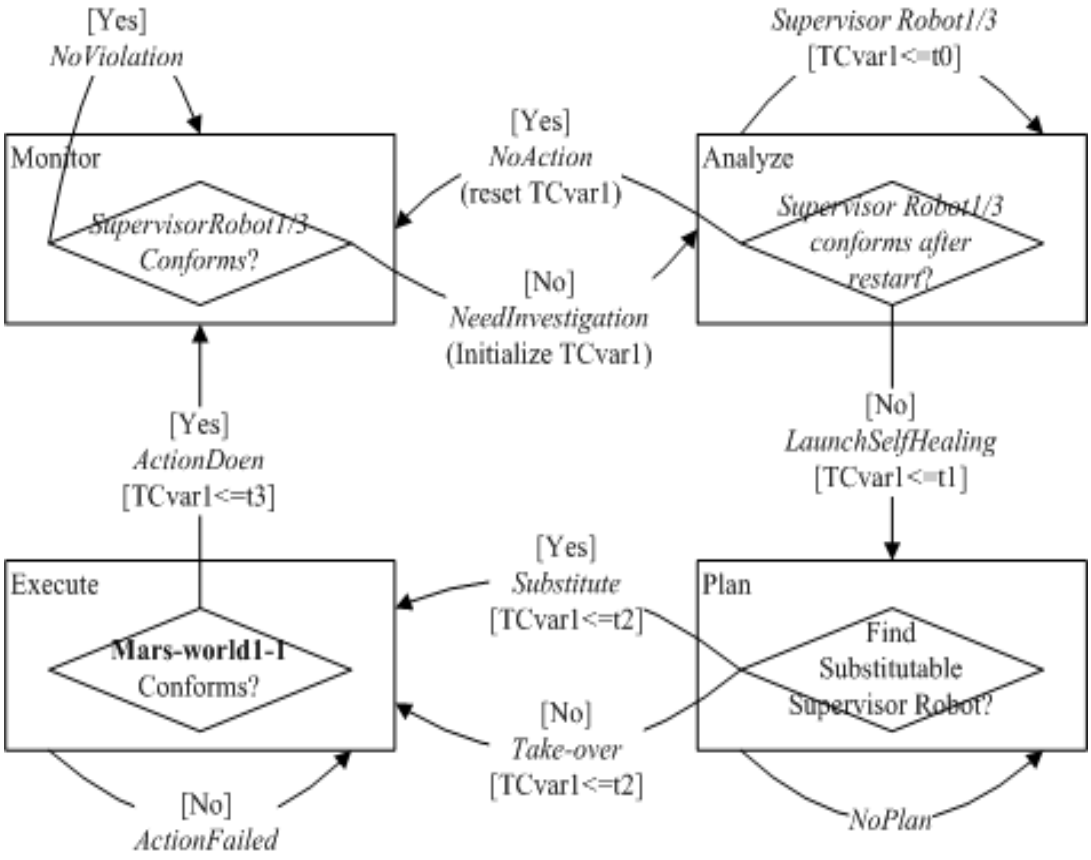


Figure 171: Intelligent Control Loop in Manager Robot1 for Self-Configuration

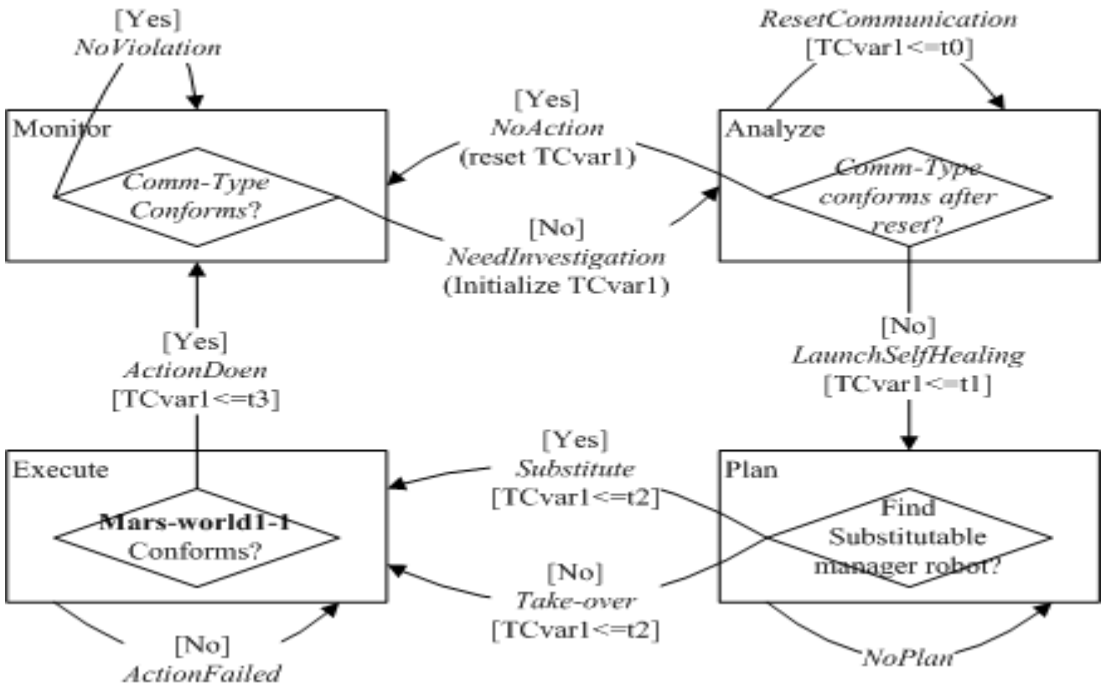


Figure 172: ICL in Manager Robot1 for Communication Self-Configuration

When *manager robot1* is in *Plan* state, it chooses either *Substitute* plan or *Take-over* plan, based on the availability of substitutable supervisor robot for *supervisor robot1* (scenario 1 in the paragraph above) or for *supervisor robot3* (scenario 2). *Manager robot1* transits to *Execute* state triggered by the *Substitute* event or *Take-over* event respectively in *t2* ticks. For scenario 3, *manager robot1* sends a *selfViolation* message to *Console*, and the latter chooses either *Substitute* plan or *Take-over* plan based on the availability of substitutable manager robot for *manager robot1*. It transits to *Execute* state triggered by *Substitute* or *Take-over* event in *t2* ticks.

When *manager robot1* is in *Execute* state and *Substitute* plan is applicable, it sends a *register* message to the substitutable supervisor robot of *supervisor robot1* (scenario 1) or *supervisor robot3* (scenario 2) and initialize it to status of *supervisor robot1* or *supervisor*

robot3 according to the checkpoint made before. When the *take-over* plan is applicable, *manager robot1* sends a *take-over* message to *supervisor robot2* (scenario 1 or scenario 2) and update it to the status of the synchronous product machine of *supervisor robot1* and *supervisor robot2*, or *supervisor robot3* and *supervisor robot2* based on the checkpoint. After the plan execution, *manager robot1* validates the configuration of **Mars-world'**, an evolution of **Mars-world** (see Property 6.1.3) against its index category **Mars-world-Formation** according to their categorical specifications. If that configuration conforms to the index category (see Property 8.7.4), *ActionDone* event occurs and *manager robot1* transits to the *Monitor* state in *t3* ticks; otherwise, *ActionFailed* event keeps it in *Execute* state for the user intervention from the *Console*. For scenario 3, the *manager robot1* is substituted by *manager robot3* or taken over by *manager robot2*.

10.1.4 Categorical Model of Structure in Mars-World

According to the Property 6.1.2 in Chapter 6, every robot in Mars-world, a production robot, for instance, is a category **Production-Robot1** (**PR1**) consisting of objects *Drill1*, *Sensor1*, *Control-Unit1* (*CU1*) and their interactions **PR1**(*Drill1*, *Sensor1*), **PR1**(*CU1*, *Drill1*) and **PR1**(*CU1*, *Sensor1*). Similarly, for an *exploration group* (see Property 6.1.7), the category **Exploration Group1** (**EG1**) includes the full sub-categories **Production-Robot1** (**PR1**), **Carry Robot1** (**CR1**), **Sentry Robot1** (**SR1**), and **Supervisor Robot1** (**Supervisor1**). Moreover, the types of robots, their parts and groups can be specified by their corresponding type categories as **Robot-Formation**, **Robot-Group-Formation** and **Mars-World-Formation** (Phase 2 in Section 5.5).

The evolution of a robot, for example, from **PR1** to **PR1'**, because of the new configuration for its *drill1* or *sensor1*, is a functor F: **PR1'** → **PR1-0** (see Property 6.1.3). Moreover, the evolution of an *exploration group*, for instance, from **EG1** to **EG1'** due to the new organization for its **PR1**, **CR1**, or **SR1** may be modeled as F: **EG1'** → **EG1-0** (see Property 6.1.9). The relationship between two solutions in terms of fault-tolerance for an *exploration group*, *Solution1*: **EG1'** → **EG1-0** as well as *Solution2*: **EG1"** → **EG1-0**, can be modeled by a natural transformation *convert1*: *Solution1* → *Solution2* (see Property 6.1.10). All those solutions (functors) along with their conversions (natural transformations) may be specified by the functor category **Solutions** (**EG1'**, **EG1-0**) as the Property 6.1.11.

The figure below depicts the representation of the index category **Robot-Formation** in a XML format (Phase 2 in Section 5.5); more details can be found in Appendix I.

```
<CATEGORY name = "Robot-Formation">
    <OBJECT>
        <OBJECT name = "Sensor"/>
        <OBJECT name = "Drill"/>
        <OBJECT name = "Trailer"/>
        <OBJECT name = "Control-Unit"/>
    </OBJECT>
    <MORPHISM>
        <MORPHISM name = "Communication-from-CU-to-Sensor"/>
            <FROM-OBJECT name = "Control-Unit"/>
            <TO-OBJECT name = "Sensor"/>
        </MORPHISM>
        <MORPHISM name = "Communication-from-CU-to-Drill"/>
            <FROM-OBJECT name = "Control-Unit"/>
            <TO-OBJECT name = "Drill"/>
        </MORPHISM>
        <MORPHISM name = "Communication-from-CU-to-Trailer"/>
```

```
            <FROM-OBJECT name = "Control-Unit"/>
            <TO-OBJECT name = "Trailer"/>
        </MORPHISM>
        <MORPHISM name = "Communication-from-Sensor-to-CU"/>
            <FROM-OBJECT name = "Sensor"/>
            <TO-OBJECT name = "Control-Unit"/>
        </MORPHISM>
        <MORPHISM name = "Communication-from-Drill-to-CU"/>
            <FROM-OBJECT name = "Drill"/>
            <TO-OBJECT name = "Control-Unit"/>
        </MORPHISM>
        <MORPHISM name = "Communication-from-Trailer-to-CU"/>
            <FROM-OBJECT name = "Trailer"/>
            <TO-OBJECT name = "Control-Unit"/>
        </MORPHISM>
        <MORPHISM name = "Communication-from-Sensor-to-Drill"/>
            <FROM-OBJECT name = "Sensor"/>
            <TO-OBJECT name = "Drill"/>
        </MORPHISM>
        <MORPHISM name = "Communication-from-Sensor-to-Trailer"/>
            <FROM-OBJECT name = "Sensor"/>
            <TO-OBJECT name = "Trailer"/>
        </MORPHISM>
        <MORPHISM name = "Communication-from-Drill-to-Sensor"/>
            <FROM-OBJECT name = "Drill"/>
            <TO-OBJECT name = "Sensor"/>
        </MORPHISM>
        <MORPHISM name = "Communication-from-Trailer-to-Sensor"/>
            <FROM-OBJECT name = "Trailer"/>
            <TO-OBJECT name = "Sensor"/>
        </MORPHISM>
    </MORPHISM>
</CATEGORY>
```

Figure 173: XML Specification of Index Category **Robot-Formation**

10.1.5 Categorical Model of Behavior in Mars-World

The synchronous communication and asynchronous between the parts of a robot, such as the control units, sensors or drills can be specified as their products and coproduct

correspondingly (see Property 6.2.6 & 6.2.7).

```
<CATEGORY name = "Robot-Part-Behavior">
    <OBJECT>
        <OBJECT name = "Function-Pair_i" type = "Function-Pair-Type_i"/>
        <OBJECT name = "Function-Pair_j" type = "Function-Pair-Type_j"/>
    </OBJECT>
    <MORPHISM>
        <MORPHISM name = "Interaction_n" type ="Interaction-Type_n"/>
            <FROM-OBJECT name = "Function-Pair_i"
                                type = "Function-Pair-Type_i"/>
            <TO-OBJECT name = "Function-Pair_j" type = "Function-Pair-Type_j"/>
        <MORPHISM>
    </MORPHISM>
</CATEGORY>
```

Figure 174: XML Specification of Category **Robot-Part-Behavior**

```
<PRODUCT name = "Sync-Communication-between-Sensor-and-Drill">
    <PRODUCT-OBJECT name = "Synchronous-Communication_m"
                        type = "Sync-Communication-between-Robot-Parts"/>
    <BETWEEN-OBJECT name = "Sensor_i" type = "Sensor"/>
    <BETWEEN-OBJECT name = "Drill_j" type = "Drill"/>
</PRODUCT>
<PRODUCT name = "Sync-Communication-between-Sensor-and-Trailer">
    <PRODUCT-OBJECT name = "Synchronous-Communication_n"
                        type = "Sync-Communication-between-Robot-Parts"/>
    <BETWEEN-OBJECT name = "Sensor_x" type = "Sensor"/>
    <BETWEEN-OBJECT name = "Trailer_y" type = "Trailer"/>
</PRODUCT>
```

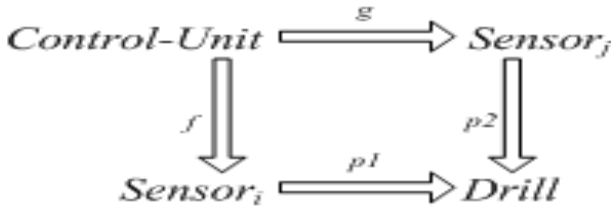
Figure 175: XML Specification of Synchronous Communication in Robot

```
<COPRODUCT name = "Async-Communication-between-Sensor-and-CU">
    <COPRODUCT-OBJECT name = "Asynchronous-Communication_x"
                        type = "Async-Communication-between-Robot-Parts"/>
    <BETWEEN-OBJECT name = "Sensor_i" type = "Sensor"/>
    <BETWEEN-OBJECT name = "CU_j" type = "Control-Unit"/>
</COPRODUCT>
<COPRODUCT name = "Async-Communication-between-Drill-and-CU">
    <COPRODUCT-OBJECT name = "Asynchronous-Communication_y"
                        type = "Async-Communication-between-Robot-Parts"/>
```

```
    <BETWEEN-OBJECT name = "Drill_i" type = "Drill"/>
    <BETWEEN-OBJECT name = "CU_j" type = "Control-Unit"/>
<COPRODUCT name = "Async-Communication-between-Trailer-and-CU">
    <COPRODUCT-OBJECT name = "Asynchronous-Communication_z"
                          type = "Async-Communication-between-Robot-Parts"/>
    <BETWEEN-OBJECT name = "Trailer_i" type = "Drill"/>
    <BETWEEN-OBJECT name = "CU_j" type = "Control-Unit"/>
</COPRODUCT>
```

Figure 176: XML Specification of Asynchronous Communication in Robot

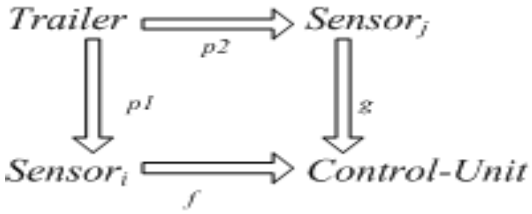
The next relay of the outgoing communication from the same source object *Control-Unit* to the same destination object *Drill* by two relay objects *Sensors* can be specified as the pushout $Drill = Sensor_i +_{Control\text{-}Unit} Sensor_j$ (see Property 6.2.8 and the figure below).




```
<PUSHOUT name = "Next-Communication-Relay-from-CU-to-Drill">
    <SOURCE-OBJECT name = "CU_n" type = "Control-Unit"/>
    <RELAY-OBJECT name = "Sensor_i" type = "Sensor"/>
    <RELAY-OBJECT name = "Sensor_j" type = "Sensor"/>
    <DESTINATION-OBJECT name = "Drill_pushout" type = "Drill"/>
</PUSHOUT>
```

Figure 177: XML Specification of Pushout *Next Communication Relay in Robot*

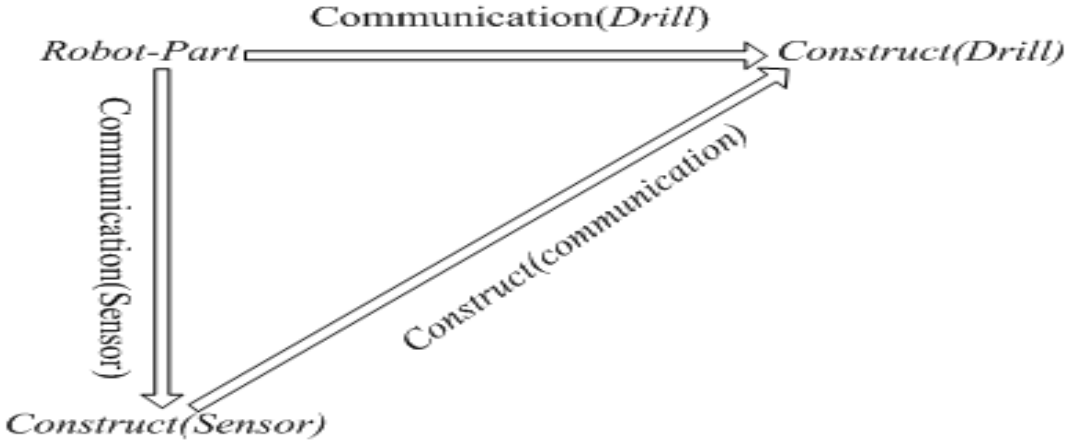
The previous relay of the incoming communication toward the same destination object *Control-Unit* from the same source object *Trailer* by two replay object *Sensors* can be specified as the pullback: $Trailer = Sensor_i \times_{Control\text{-}Unit} Sensor_j$ (see property 6.2.9).




```
<PULLBACK name = "Previous-Communication-Relay-from-Trailer-to-CU">
    <SOURCE-OBJECT name = "Trailer_pullback" type = "Trailer"/>
    <RELAY-OBJECT name = "Sensor_i" type = "Sensor"/>
    <RELAY-OBJECT name = "Sensor_j" type = "Sensor"/>
    <DESTINATION-OBJECT name = "CU_n" type = "Control-Unit"/>
</PULLBACK>
```

Figure 178: XML Specification of Pullback *Previous Communication Relay in Robot*

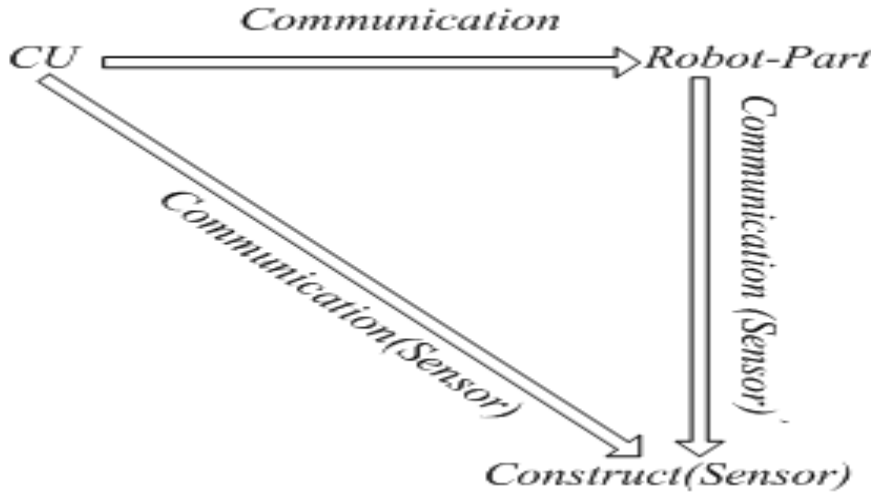
The designated behavior of a robot, such as the sentry robot, production robot, carry robot or supervisor robot, can be specified as a category of cones, where objects are the cones consisting of an object *Robot-Part* with a family of *Communications* in the diagram from **Robot-Formation** to **Robot**, and morphisms are incoming communications among those cones (see Property 6.2.10 and the figure below).




```
<CATEGORY name = "Robot-Behavior-Designated">
    <OBJECT>
        <OBJECT name = "Cone-to-Diagram_i"
                type = "Cone-to-Diagram-Robot-Part"/>
        <OBJECT name = "Cone-to-Diagram_j"
                type = "Cone-to-Diagram-Robot-Part"/>
```

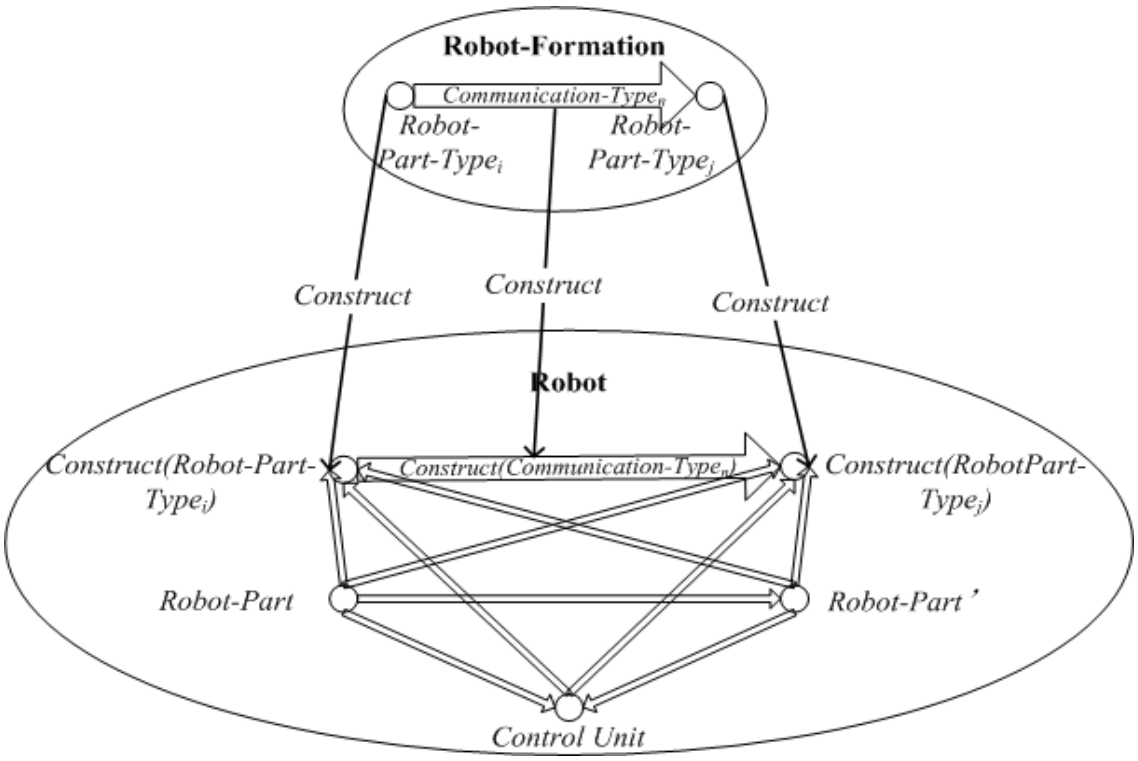
```
    </OBJECT>
    <MORPHISM>
        <MORPHISM name = "Incoming-Communication_n"
                        type ="Incoming-Communication-Robot-Part"/>
            <FROM-OBJECT name = "Cone-to-Diagram_i"
                                type = "Cone-to-Diagram-Robot-Part"/>
            <TO-OBJECT name = "Cone-to-Diagram_j"
                              type = "Cone-to-Diagram-Robot-Part"/>
        <MORPHISM>
    </MORPHISM>
</CATEGORY>
```

Figure 179: XML Specification of Category **Robot-Behavior-Designated**

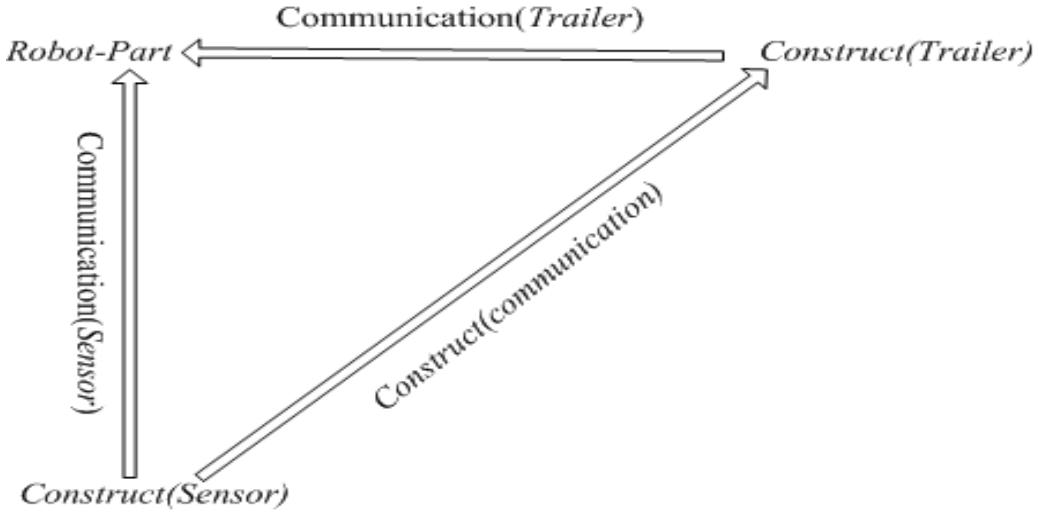



```
<LIMIT name = "Limit-of-Robot-Behavior-Designated">
    <DIAGRAM name = "Construct_n" source-category = "Robot-Formation"
                    destination-category = "Robot_n"/>
    <BEHAVIOR-CATEGORY name = "Robot_n-Behavior-Designated"/>
    <TERMINAL-OBJECT name = "CU_n" type = "Control-Unit"/>
</LIMIT>
```

Figure 180: XML Specification of Limit *Limit-of-Robot-Behavior-Designated*

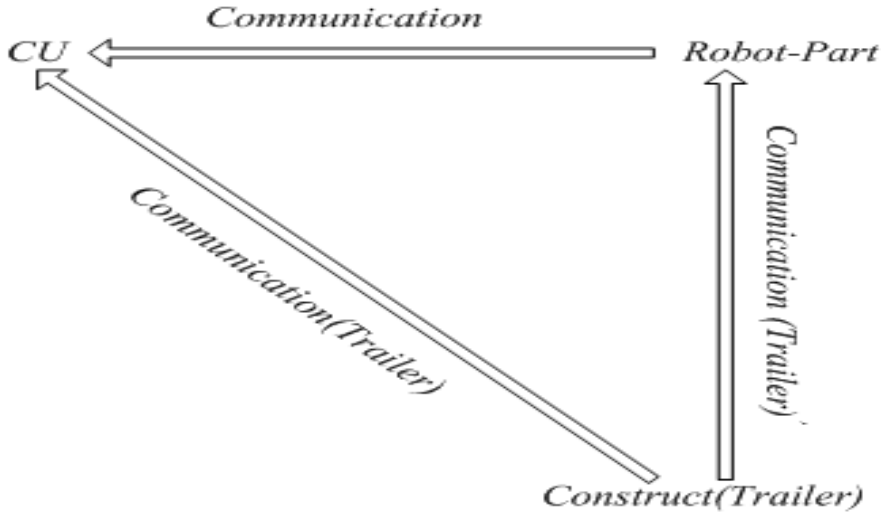


The achieved behavior of a robot in the runtime can be specified as a category of cocones, where objects are the cocones consisting of an object *Robot-Part* with a family of *Communications* in the diagram from **Robot-Formation** to **Robot**, and morphisms are outgoing communications among the cocones (see Property 6.2.11 and the figure below).



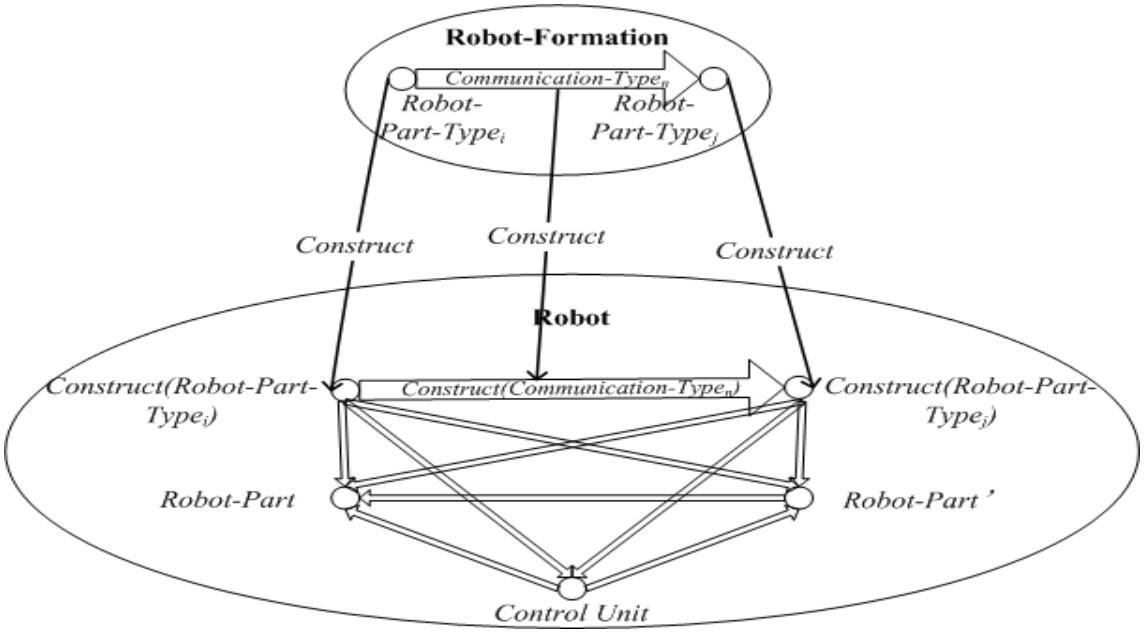
```
<CATEGORY name = "Robot-Behavior-Achieved">
    <OBJECT>
        <OBJECT name = "Cocone-to-Diagram_i"
                type = "Cocone-to-Diagram-Robot-Part"/>
        <OBJECT name = "Cocone-to-Diagram_j"
                type = "Cocone-to-Diagram-Robot-Part"/>
    </OBJECT>
    <MORPHISM>
        <MORPHISM name = "Outgoing-Communication_n"
                  type ="Outgoing-Communication-Robot-Part"/>
            <FROM-OBJECT name = "Cocone-to-Diagram_i"
                         type = "Cocone-to-Diagram-Robot-Part"/>
            <TO-OBJECT name = "Cocone-to-Diagram_j"
                       type = "Cocone-to-Diagram-Robot-Part"/>
        <MORPHISM>
    </MORPHISM>
</CATEGORY>
```

Figure 181: XML Specification of Category **Robot-Behavior-Achieved**

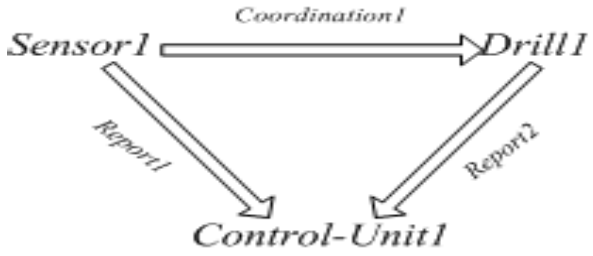



```
<COLIMIT name = "Colimit-of-RAC-Behavior-Achieved">
    <DIAGRAM name = "Construct_n" source-category = "RAO-Type"
             destination-category = "RAC_n"/>
    <BEHAVIOR-CATEGORY name = "RAC_n-Behavior-Achieved"/>
    <INITIAL-OBJECT name = "RAOL_n" type = "RAOL-Type_n"/>
</COLIMIT>
```

Figure 182: XML Specification of Colimit *Colimit-of-Robot-Behavior-Achieved*



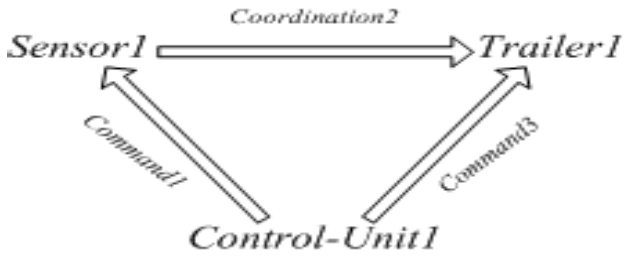
The outgoing communication from the *Sensor*, *Drill* and *Trailer* to their *Control Unit* in a *Robot* can be specified by a slice category as **Robot**/*CU* (see Property 6.2.12).




```
<SLICE-CATEGORY name = "Production-Robot1/Control-Unit1">
    <OBJECT>
        <OBJECT name = "Report1" type = "Report"/>
        <OBJECT name = "Report2" type = "Report"/>
    </OBJECT>
    <MORPHISM>
        <MORPHISM name = "Coordination1" type ="Coordination"/>
            <FROM-OBJECT name = "Report1" type = "Report"/>
            <TO-OBJECT name = "Report2" type = "Report"/>
        <MORPHISM>
    </MORPHISM>
</SLICE-CATEGORY>
```

Figure 183: XML Specification of Slice Category **Production-Robot1**/*CU1*

The incoming communication from the *Control Unit* to the *Sensor*, *Drill* and *Trailer* in a *Robot* can be specified by a coslice category as *CU*/**Robot** (see Property 6.2.13).




```
<COSLICE-CATEGORY name = "Control-Unit1/Production-Robot1">
    <OBJECT>
        <OBJECT name = "Command1" type = "Command"/>
        <OBJECT name = "Command3" type = "Command"/>
    </OBJECT>
    <MORPHISM>
        <MORPHISM name = "Coordination2" type ="Coordination"/>
            <FROM-OBJECT name = "Command1" type = "Command"/>
            <TO-OBJECT name = "Command3" type = "Command"/>
        <MORPHISM>
    </MORPHISM>
</CATEGORY>
```

Figure 184: XML Specification of Coslice Category *CU1*/**Production-Robot1**

The social life of any *Robot* in the category **Mars-world** is a subcategory of **Mars-world** denoted as **SOCIAL**(*Robot*), where the objects are *Robot* and all other $Robot' \in$ |**Mars-world**| that have morphisms with *Robot*, and the morphisms are social connections between *Robot* and $Robot'$ as **Mars-world**(*Robot*, $Robot'$) or **Mars-world**($Robot'$, *Robot*) as the Definition 3.1.1.

10.1.6 Categorical Model of Self-Healing in Mars-World

Figure 185 illustrates a categorical specification of *production robot1* before *snesor1* is crashed. The category **PR1** consists of three objects *drill1*, *sensor1* and *CU1*. Therefore,

the bidirectional communications between those objects are six morphisms to specify the working collaboration between *drill1* and *sensor1* (*coordination1*, *coordiantion2*) as well as leadership among *CU1*, *drill1* and *sensor1* (*command1*, *report1*, *command2*, *report2*). Synchronous communication between *drill1* and *sensor1* is specified as their product. Slice category models *Report* actions (*report1*, *report2*) with the relations (*coordination1*) from *drill1* and *sensor1* to *CU1*; coslice category specifies *Command* actions (*command1*, *command2*) with their relations (*coordination2*) from *CU1* to *drill1* and *sensor1*.

CAT-RAC <Production Robot1>
Objects: <Drill1, Sensor1, CU1>
Morphisms: <Coordiantion1(Drill1, Sensor1), Coordination2(Sensor1, Drill1), Report1(Drill1, CU1), Command1(CU1, Drill1), Report2(Sensor1, CU1), Order2(CU1, Sensor1)>
Limit Object: <CU1>
Colimit Object: <CU1>
Product Objects: <SPM1(Drill1, Sensor1)>
Coproduct Objects: <(Drill1, CU1), (Sensor1, CU1)>
Pushout Objects: <Collect1(Command1, Command2)>
Pullback Objects: <Trance1(Report1, Report2)>
Slice Category: <(Report1, Report2), Coordination1>
Co*slice Category*: <(Command1, Command2), Coordination2>
Functors: <Restart(PR1-1, PR1-0), Substitute(PR1-2, PR1-0), Take-over(PR1-3, PR1-0)>
Natural Transformations: <Convert1(Restart, Substitute), Convert2(Substitute, Take-over), Convert3(Restart, Take-over)>
Functor Category: <(Restart, Substitute, Take-over), Convert1, Convert2, Convert3>
End CAT-RAC

Figure 185: Categorical Specification of Production Robot

If **PR1** is recovered by restarting crashed *sensor1*, it evolves to **PR1-1** consisting of the same composition and categorical specification as **PR1** except for the different initial status of *sensor1*, and this evolution is represented as the *Restart* functor from **PR1-1** to **PR1-0** (a category with all potential robot parts for the self-healing in a production robot).

If **PR1** is recovered by replacing *sensor1* with one of its equivalent object *sensor8*, it evolves to **PR1-2** having the same composition and categorical specification as **PR1** except for substituting each *sensor1* with *sensor8*, which evolution is specified as a *Substitute* functor. However, if **PR1** is recovered by asking *drill1* to take over the responsibilities of *sensor1*, it will evolve to **PR1-3** which has a different composition and categorical specification (see Figure 186), but both of them have the equivalent behavior, since *drill1'* works as the product object of original *drill1* and *sensor1* through its backup sensor and *drill1'* ~ *drill1* × *sensor1*.

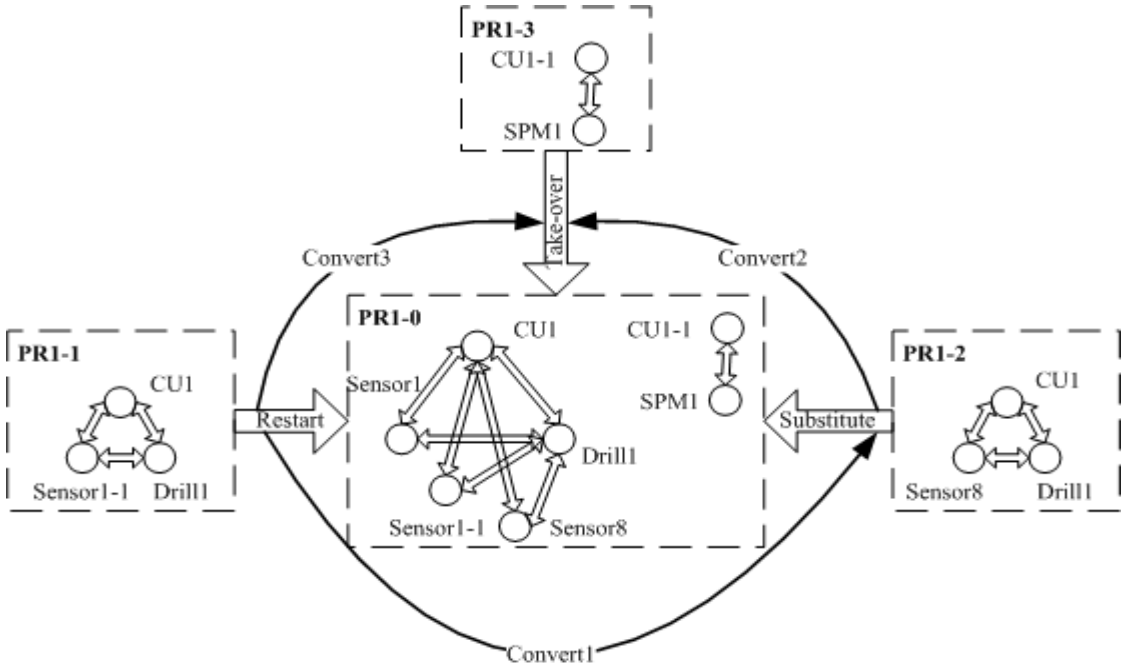


Figure 186: Evolution for Self-Healing in Production Robot

Moreover, the conversions between the plans *Restart*, *Substitute*, and *Take-over* may be interpreted as a *Convert* natural transformations. For example, *Convert1* is used to specify the mapping from *Restart* to *Substitute* when **PR1** cannot be recovered after restarting *Sensor1* due to its defects. Thus, a functor category consisting of those functors

as objects with their natural transformations as morphisms models all possible evolutions and their relations.

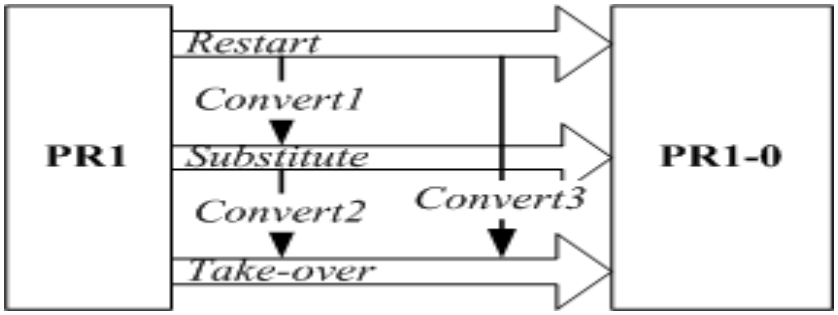


Figure 187: Natural Transformation for Self-Healing in Production Robot1

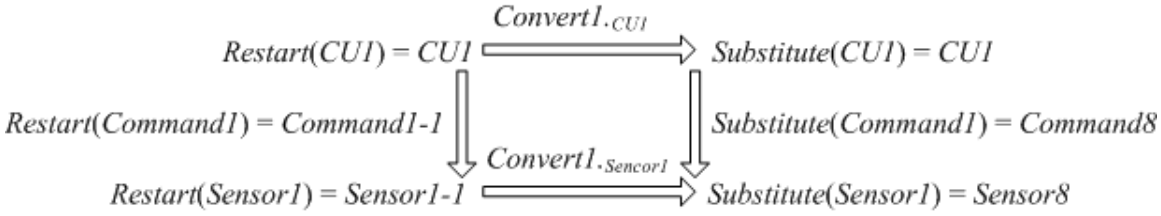


Figure 188: Natural Transformation from *Restart* to *Substitute* in PR1

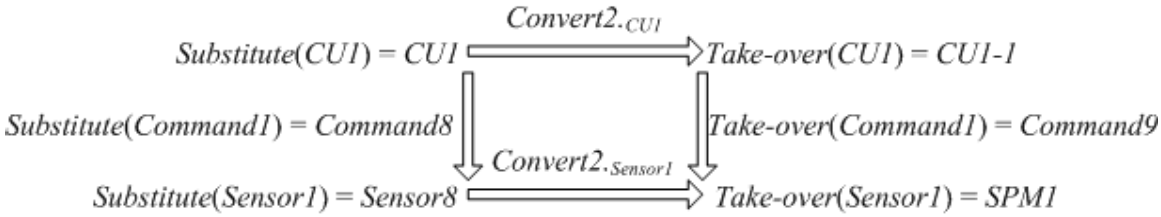


Figure 189: Natural Transformation from *Substitute* to *Take-over* in PR1

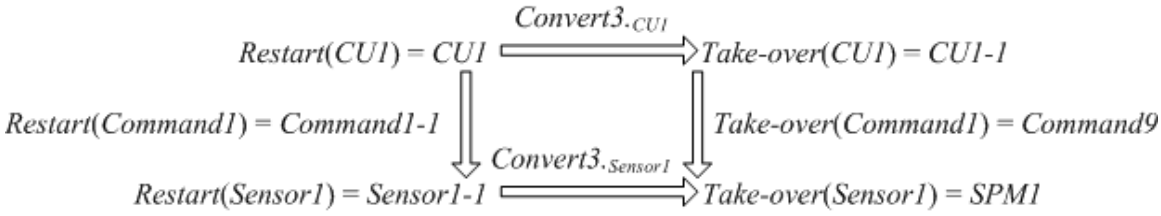


Figure 190: Natural Transformation from *Restart* to *Take-over* in PR1

The following figures illustrate the representation of the categorical model for the self-healing property described above in a XML format (Phase 2 in Section 5.5).

```
<FUNCTOR name = "Production-Robot-Self-Healing-Restart"
                source-category = "PR1-1"
                target-category = "PR1-0">
    <OBJECT-MAPPING>
        <OBJECT-MAPPING source-object = "Sensor1-1"
                                target-object = "Sensor1"/>
        <OBJECT-MAPPING source-object = "Drill1"
                                target-object = "Drill1"/>
        <OBJECT-MAPPING source-object = "Control-Unit1"
                                target-object = "Control-Unit1"/>
    </OBJECT-MAPPING>
    <MORPHISM-MAPPING>
        <MORPHSIM-MAPPING source-morphism = "Command1-1"
                                target-morphism = "Command1"/>
        <MORPHSIM-MAPPING source-morphism = "Command2"
                                target-morphism = "Command2"/>
        <MORPHSIM-MAPPING source-morphism = "Report1-1"
                                target-morphism = "Report1"/>
        <MORPHSIM-MAPPING source-morphism = "Report2"
                                target-morphism = "Report2"/>
        <MORPHSIM-MAPPING source-morphism = "Cooperate1-1"
                                target-morphism = "Cooperate1"/>
        <MORPHSIM-MAPPING source-morphism = "Cooperate2"
                                target-morphism = "Cooperate2"/>
    </MORPHISM-MAPPING>
</FUNCTOR>
```

Figure 191: XML Specification of Functor *PR1-Self-Healing-Restart*

```
<FUNCTOR name = "Production-Robot-Self-Healing-Substitute"
                source-category = "PR1-2"
                target-category = "PR1-0">
    <OBJECT-MAPPING>
        <OBJECT-MAPPING source-object = "Sensor8"
                                target-object = "Sensor1"/>
        <OBJECT-MAPPING source-object = "Drill1"
                                target-object = "Drill1"/>
        <OBJECT-MAPPING source-object = "Control-Unit1"
                                target-object = "Control-Unit1"/>
    </OBJECT-MAPPING>
    <MORPHISM-MAPPING>
        <MORPHSIM-MAPPING source-morphism = "Command8"
```

```
                                    target-morphism = "Command1"/>
        <MORPHSIM-MAPPING source-morphism = "Command2"
                                    target-morphism = "Command2"/>
        <MORPHSIM-MAPPING source-morphism = "Report8"
                                    target-morphism = "Report1"/>
        <MORPHSIM-MAPPING source-morphism = "Report2"
                                    target-morphism = "Report2"/>
        <MORPHSIM-MAPPING source-morphism = "Cooperate8"
                                    target-morphism = "Cooperate1"/>
        <MORPHSIM-MAPPING source-morphism = "Cooperate2"
                                    target-morphism = "Cooperate2"/>
    </MORPHISM-MAPPING>
</FUNCTOR>
```

Figure 192: XML Specification of Functor *PR1-Self-Healing-Substitute*

```
<FUNCTOR name = "Production-Robot-Self-Healing-Take-Over"
            source-category = "PR1-3"
            target-category = "PR1-0">
    <OBJECT-MAPPING>
        <OBJECT-MAPPING source-object = "SPM1"
                                target-object = "Sensor1"/>
        <OBJECT-MAPPING source-object = "SPM1"
                                target-object = "Drill1"/>
        <OBJECT-MAPPING source-object = "Control-Unit1-1"
                                target-object = "Control-Unit1"/>
    </OBJECT-MAPPING>
    <MORPHISM-MAPPING>
        <MORPHSIM-MAPPING source-morphism = "Command6"
                                    target-morphism = "Command1"/>
        <MORPHSIM-MAPPING source-morphism = "Command7"
                                    target-morphism = "Command2"/>
        <MORPHSIM-MAPPING source-morphism = "Report6"
                                    target-morphism = "Report1"/>
        <MORPHSIM-MAPPING source-morphism = "Report7"
                                    target-morphism = "Report2"/>
        <MORPHSIM-MAPPING source-morphism = "Cooperate6"
                                    target-morphism = "Cooperate1"/>
        <MORPHSIM-MAPPING source-morphism = "Cooperate7"
                                    target-morphism = "Cooperate2"/>
    </MORPHISM-MAPPING>
```

```
</FUNCTOR>
```

Figure 193: XML Specification of Functor *PR1-Self-Healing-Take-Over*

```
<NATURAL-TRANSFORMATION name = "Relation-of-PR-Evolution">
    <ARROW>
        <ARROW name = "Convert1"/>
            <FROM-FUNCTOR name = "PR-Self-Healing-Restart"
                                type = "PR-Evolution-Self-Healing"/>
            <TO-FUNCTOR name = "PR-Self-Healing-Substitute"
                                type = "PR-Evolution-Self-Healing"/>
        </ARROW>
        <ARROW name = "Convert3"/>
            <FROM-FUNCTOR name = "PR-Self-Healing-Restart"
                                type = "PR-Evolution-Self-Healing"/>
            <TO-FUNCTOR name = "PR-Self-Healing-Take-Over"
                                type = "PR-Evolution-Self-Healing"/>
        </ARROW>
        <ARROW name = "Convert2"/>
            <FROM-FUNCTOR name = "PR-Self-Healing-Substitute"
                                type = "PR-Evolution-Self-Healing"/>
            <TO-FUNCTOR name = "PR-Self-Healing-Take-Over"
                                type = "PR-Evolution-Self-Healing"/>
        </ARROW>
    </ARROW>
</NATURAL-TRANSFORMATION>
```

Figure 194: XML Specification of Natural Transformation *PR-Evolution-Relation*

```
<CATEGORY name = "Relation-Set-of-PR-Evolution-Self-Healing">
    <OBJECT>
        <OBJECT name = "PR-Self-Healing-Restart"
                    type = "PR-Evolution-Self-Healing" />
        <OBJECT name = "PR-Self-Healing-Substitute"
                    type = "PR-Evolution-Self-Healing"/>
        <OBJECT name = "PR-Self-Healing-Take-Over"
                    type = "PR-Evolution-Self-Healing"/>
    </OBJECT>
    <MORPHISM>
        <MORPHISM name = "Convert1"
                        type ="PR-Evolution-Relation-Self-Healing"/>
            <FROM-OBJECT name = "PR-Self-Healing-Restart"
```

```
                                type = "PR-Evolution-Self-Healing" />
            <TO-OBJECT name = "PR-Self-Healing-Substitute"
                                type = "PR-Evolution-Self-Healing"/>
        </MORPHISM>
        <MORPHISM name = "Convert3"
                            type ="PR-Evolution-Relation-Self-Healing"/>
            <FROM-OBJECT name = "PR-Self-Healing-Restart"
                                type = "PR-Evolution-Self-Healing" />
            <TO-OBJECT name = "PR-Self-Healing-Take-Over"
                                type = "PR-Evolution-Self-Healing"/>
        </MORPHISM>
        <MORPHISM name = "Convert2"
                            type ="PR-Evolution-Relation-Self-Healing"/>
            <FROM-OBJECT name = "PR-Self-Healing-Substitute"
                                type = "PR-Evolution-Self-Healing" />
            <TO-OBJECT name = "PR-Self-Healing-Take-Over"
                                type = "PR-Evolution-Self-Healing"/>
        </MORPHISM>
    </MORPHISM>
</CATEGORY>
```

Figure 195: XML Specification of Functor Category **PR-Evolution-Relation-Set**

10.1.7 Categorical Model of Self-Configuration in Mars-World

The actions in the formation work flow, self-configuration work flow, substitution work flow and take-over work flow of Mars-world can be specified as the categories in which objects are those actions and morphisms are their preorder relationship *before*. Each object (action) in those categories is a quadruple (see Property 6.2.28); the sequences of those actions can be specified as the categories where objects are those sequences and morphisms are *equivalence* relationship between those sequences (see Property 6.2.31).

The transitions in intelligent control loop of *manager robot1* for self-configuration can be specified as a category where objects are those transitions and the morphisms are

their preorder relations *before*. Each object (transition) in that category is a triple (see Property 6.2.23); the sequences of the transitions can be specified as a category in which objects are those sequences and morphisms are the *equivalence* relations between those sequences (see Property 6.2.26).

Let **Mars-world1** be a subcategory of **Mars-world1-0** (a category consisting of all the potential robots for self-configuration in Mars-world1). If **Mars-world1** is conformed to the index category **Mars-world-Formation** by restarting violated *supervisor robot1* or *supervisor robot3*, it will evolve to **Mars-world1-1**, which has the same configuration and categorical structure as **Mars-world1** except for different initial status of *supervisor robot1* or *supervisor robot3*. This evolution can be specified by a *Restart* functor from **Mars-world1-1** to **Mars-world1-0**. If **Mars-world1** is conformed to the **Mars-world-Formation** by substituting *supervisor robot1* or *supervisor robot3* with their isomorphic objects *supervisor7* or *supervisor robot9* (see Definition 3.1.3), it will evolve to **Mars-world1-2** that has the same configuration and categorical structure as **Mars-world1** but replacing *supervisor robot1* or *supervisor robot3* with *supervisor robot7* or *supervisor robot9*. The above is specified by the *Substitute* functor. If **Mars-world1** is conformed to the **Mars-world-Formation** by asking *supervisor robot2* to take over the responsibilities of *supervisor robot1* or *supervisor robot3*, it will evolve to **Mars-world1-3**, which has different categorical structure, but both of them have the equivalent configuration (see Property 8.7.5 and the figure below).

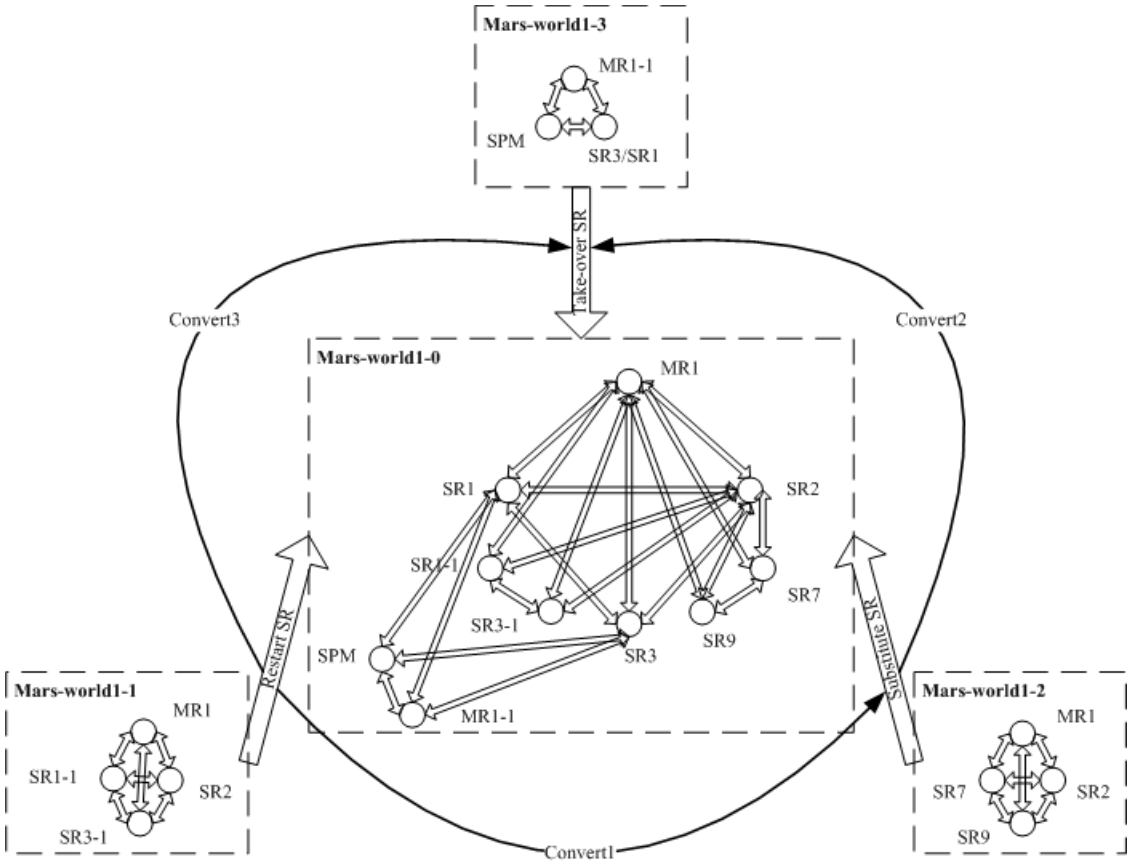


Figure 196: Evolution for Self-Configuration in Mars-world1

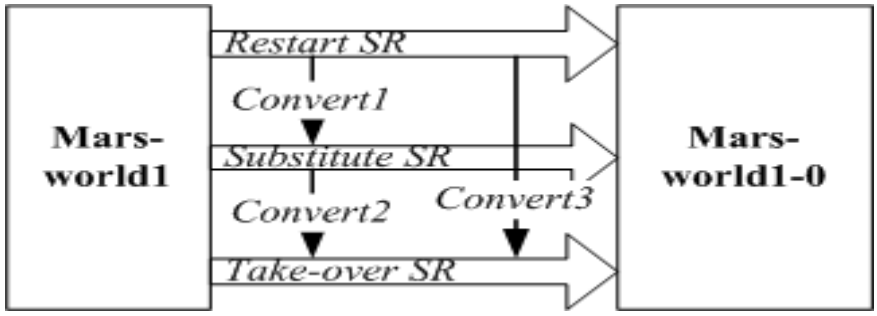


Figure 197: Natural Transformation for Self-Configuration in Mars-world1

The mapping among those evolutions *RestartSR*, *SubstituteSR* and *Take-over-SR* of the **Mars-world1** can be interpreted as natural transformations (see Property 6.1.4). The functor category having those functors as objects and their natural transformations as morphisms illustrate all possible evolutions with their relations. In addition, *Convert3* is a

composition of *Convert1* and *Convert2*, which may be interpreted as the following: the result of the evolution *RestartSR -> SubstituteSR -> Take-over-SR* is equivalent to the evolution *RestartSR -> Take-over-SR*. Figure 198, 199 and 200 illustrate those natural transformations and their composition respectively.

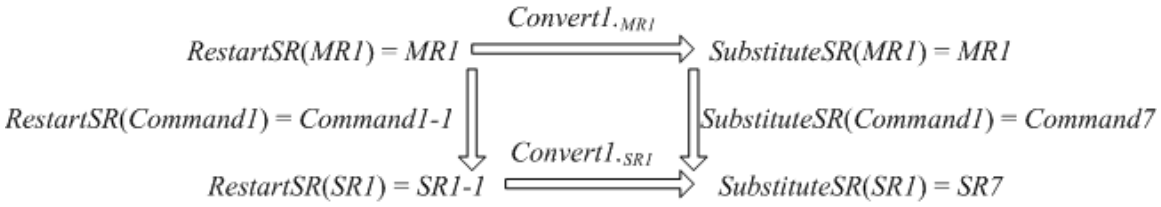


Figure 198: Natural Transformation *RestartSR -> SubstituteSR* in Mars-world1

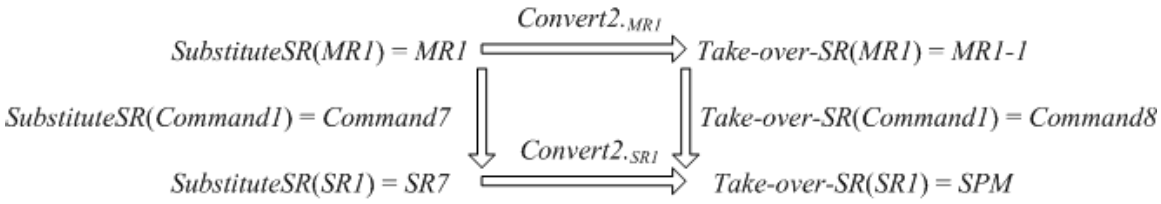


Figure 199: Natural Transformation *SubstituteSR -> Take-over-SR* in Mars-world1

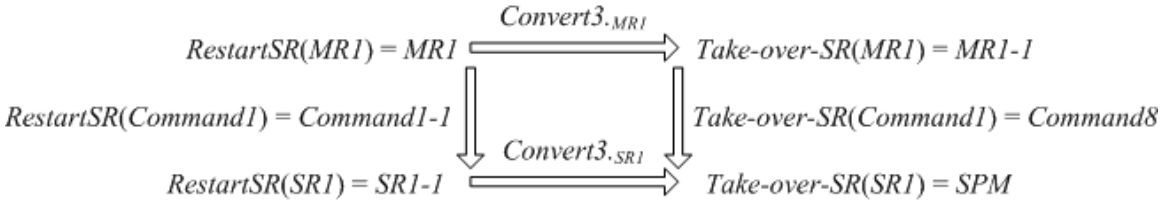


Figure 200: Natural Transformation *RestartSR -> Take-over-SR* in Mars-world1

The following figures illustrate the representation of the categorical model for the self-configuration property described above in a XML format (Phase 2 in Section 5.5).

```
<CATEGORY name = "Formation-Work-Flow-in-Mars-world">
    <OBJECT>
        <OBJECT name = "InitializeManagerRobot" type = "Work-Flow-Action"/>
        <OBJECT name = "InitializeSupervisorRobot"
                    type = "Work-Flow-Action"/>
        <OBJECT name = "Heartbeat" type = "Work-Flow-Action"/>
    </OBJECT>
```

```
    <MORPHISM>
        <MORPHISM name = "Before" type ="Preorder"/>
            <FROM-OBJECT name = "InitializeManagerRobot"
                                type = "Work-Flow-Action"/>
            <TO-OBJECT name = "Heartbeat" type = "Work-Flow-Action"/>
        <MORPHISM>
        <MORPHISM name = "Before" type ="Preorder"/>
            <FROM-OBJECT name = "InitializeManagerRobot"
                                type = "Work-Flow-Action"/>
            <TO-OBJECT name = "InitializeSupervisorRobot"
                              type = "Work-Flow-Action"/>
        <MORPHISM>
        <MORPHISM name = "Before" type ="Preorder"/>
            <FROM-OBJECT name = "InitializeSupervisorRobot"
                                type = "Work-Flow-Action"/>
            <TO-OBJECT name = "Heartbeat" type = "Work-Flow-Action"/>
        <MORPHISM>
    </MORPHISM>
</CATEGORY>
```

Figure 201: XML Specification of Category **Formation-Work-Flow-in-Mars-world**

```
<NATURAL-TRANSFORMATION name = "SR-Evolution-Self-Configuration">
    <ARROW>
        <ARROW name = "Relation1"/>
            <FROM-FUNCTOR name = "SR-Self-Configuration-Restart"
                                type = "SR-Evolution-Self-Configuration"/>
            <TO-FUNCTOR name = "SR-Self-Configuration-Substitute"
                              type = "SR-Evolution-Self-Configuration"/>
        </ARROW>
        <ARROW name = "Relation2"/>
            <FROM-FUNCTOR name = "SR-Self-Configuration-Restart"
                                type = "SR-Evolution-Self-Configuration"/>
            <TO-FUNCTOR name = "SR-Self-Configuration-Take-Over"
                              type = "SR-Evolution-Self-Configuration"/>
        </ARROW>
        <ARROW name = "Relation3"/>
            <FROM-FUNCTOR name = "SR-Self-Configuration-Substitute"
                                type = "SR-Evolution-Self-Configuration"/>
            <TO-FUNCTOR name = "SR-Self-Configuration-Take-Over"
                              type = "SR-Evolution-Self-Configuration"/>
        </ARROW>
```

```
    </ARROW>
</NATURAL-TRANSFORMATION>
```

Figure 202: XML Specification of Natural Transformation *SR-Self-Configuration*

```
<CATEGORY name = "Relation-Set-of-SR-Evolution-Self-Configuration">
    <OBJECT>
        <OBJECT name = "SR-Self-Configuration-Restart"
                    type = "SR-Evolution-Self-Configuration" />
        <OBJECT name = "SR-Self-Configuration-Substitute"
                    type = "SR-Evolution-Self-Configuration"/>
        <OBJECT name = "SR-Self-Configuration-Take-Over"
                    type = "SR-Evolution-Self-Configuration"/>
    </OBJECT>
    <MORPHISM>
        <MORPHISM name = "Relation1"
                        type ="SR-Evolution-Relation-Self-Configuration"/>
            <FROM-OBJECT name = "SR-Self-Configuration-Restart"
                            type = "SR-Evolution-Self-Configuration" />
            <TO-OBJECT name = "SR-Self-Configuration-Substitute"
                        type = "SR-Evolution-Self-Configuration"/>
        </MORPHISM>
        <MORPHISM name = "Relation2"
                        type ="SR-Evolution-Relation-Self-Configuration"/>
            <FROM-OBJECT name = "SR-Self-Configuration-Restart"
                            type = "SR-Evolution-Self-Configuration" />
            <TO-OBJECT name = "SR-Self-Configuration-Take-Over"
                        type = "SR-Evolution-Self-Configuration"/>
        </MORPHISM>
        <MORPHISM name = "Relation3"
                        type ="SR-Evolution-Relation-Self-Configuration"/>
            <FROM-OBJECT name = "SR-Self-Configuration-Substitute"
                            type = "SR-Evolution-Self-Configuration" />
            <TO-OBJECT name = "SR-Self-Configuration-Take-Over"
                        type = "SR-Evolution-Self-Configuration"/>
        </MORPHISM>
    </MORPHISM>
</CATEGORY>
```

Figure 203: XML Specification of Functor Category **SR-Self-Configuration**

```
<CATEGORY name = "Robot-Configuration">
    <OBJECT>
        <OBJECT name = "Sensor1" type = "Sensor"/>
        <OBJECT name = "Control-Unit1" type = "Control-Unit"/>
    </OBJECT>
    <MORPHISM>
        <MORPHISM name = "Command1" type ="Command"/>
            <FROM-OBJECT name = "Control-Unit1" type = "Control-Unit"/>
            <TO-OBJECT name = "Sensor1" type = "Sensor"/>
        <MORPHISM>
        <MORPHISM name = "Report1" type ="Report"/>
            <FROM-OBJECT name = "Sensor1" type = "Sensor"/>
            <TO-OBJECT name = "Control-Unit1" type = "Control-Unit"/>
        <MORPHISM>
    </MORPHISM>
</CATEGORY>
```

Figure 204: XML Specification of Category **Robot-Configuration**

10.1.8 Transform RAS Model of Mars-world to MAS Model

In this section, we will describe the transformation from the RAS model of Mars-world to its MAS model (Phase 3 in Section 5.5). We will focus on the substitutability of RAC (*robots*), since RACG (*exploration group*) is the minimum RAE that can independently fulfill a complete work in RAS (*Mars-world*). More details about this transformation can be found in [148].

In Mars-world, there are five types of agents (*robots*): *Manager*, *Supervisor, Sentry*, *Producer* and *Carry* agents. A *Manager* agent can create and manage *Supervisor* agents; each S*upervisor* is in charge of an exploration group to exploit ore mines. It initiates a number of *Sentry* agents to search and analyze ore targets in the area assigned by the *Manager*. If a target is found, the *Supervisor* assigns the task of analyzing the target to an

available *Sentry* agent and after that forms a group of *Producer* as well as *Carry* agents to perform exploiting tasks. After finishing the analyzing process, the *Sentry* requests some available *Producer* agents to exploit the target mine. After finishing the production work, a *Producer* calls some available *Carry* agents to carry the produced ore to the *home base*. Each *Carry* has a limited capacity of ore so that it travels between the target mine and home base [148].

To simulate the malfunctioning of one agent, users can click on the agent in GUI that is considered as a signal to disable it (see Figure 205). This is done inside the mouse click event listener of the environment panel in the *Mars-world GUI* plan of a *Manager* agent. If the *x* and *y* coordinates of the clicked point falls inside any agent area, a message event that tells the agent to shutdown itself is created. For each agent, there is a *shutdown* plan, which consists of taking a snapshot of the agent execution status, pushing the snapshot into a queue in terms of retrieving it later by a *Supervisor* for recovery, and shutting down the agent. The snapshot has agent snapshot, goal snapshot and message snapshot.

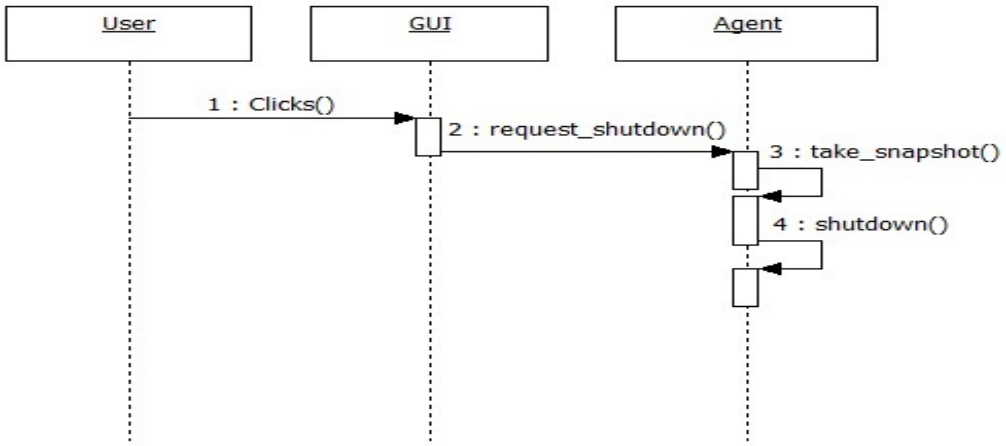


Figure 205: Sequence Diagram of Shutdown [148]

Each *Supervisor* agent has a perform goal (see Figure 206) that checks continuously the status of the agents in the environment belonging to the *Supervisor*'s group. In the plan triggered by this goal, if the *Supervisor* detects any inactive agent whose type is checked, it selects an appropriate recovery plan for that type of agent and creates a top level goal for the agent's recovery [148].

The *Supervisor* agent has a recovery plan for each of the group agents including the *Sentry* agents, *Producer* agents and *Carry* agents. For example, if a *Carry* agent is getting damaged, the *Supervisor* selects *RecoverCarryPlan* to recover the *Carry* agent. This plan consists of four steps: 1) it creates a new *Carry* agent from scratch; 2) it recovers the miscellaneous agent information, such as the location of the agent; 3) it deals with the goal recovery; and 4) it recovers the message event queue of the *Carry* agent [148].

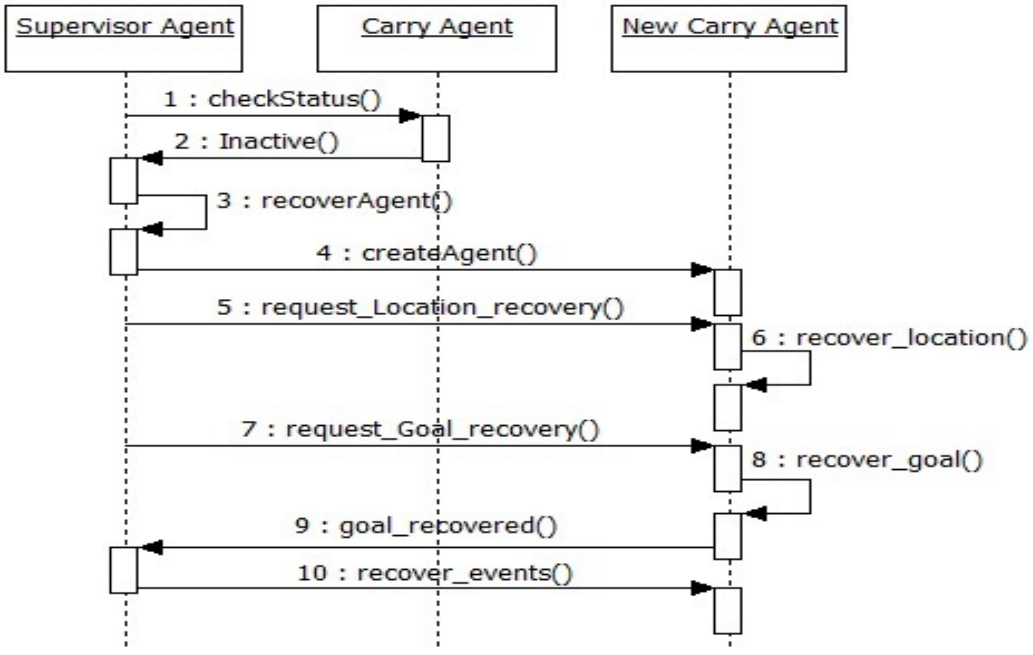


Figure 206: Sequence Diagram of Carry Recovery [148]

10.1.9 Transform MAS Model of Mars-world to Categorical Model

In this section, the following agents from the Mars-world will be used to represent the fault-tolerance property (Phase 4 in Section 5.5). They are repository agent ***Repository***, repository type ***Repository Type***, supervisor agent ***Supervisor***, carry agents ***Carry$_1$*** and ***Carry$_2$***. More details can be found in [62].

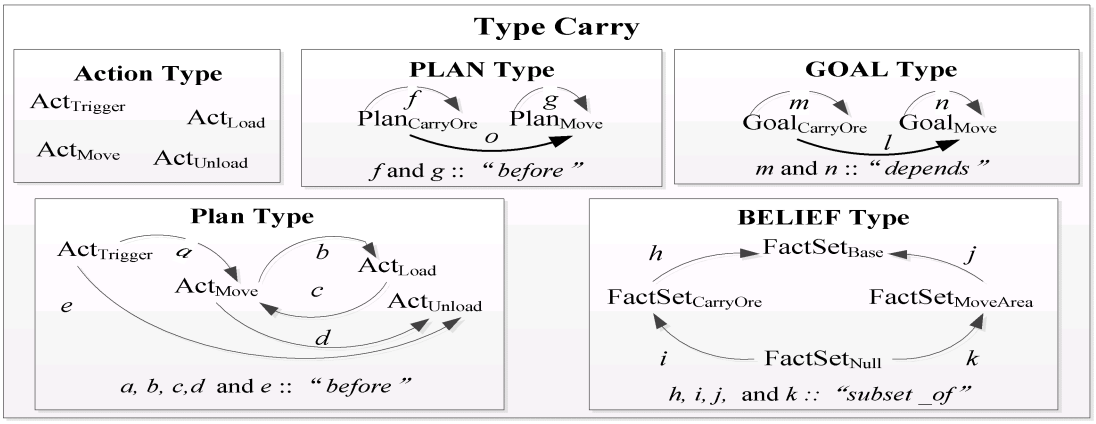


Figure 207: Carry Agent Type [62]

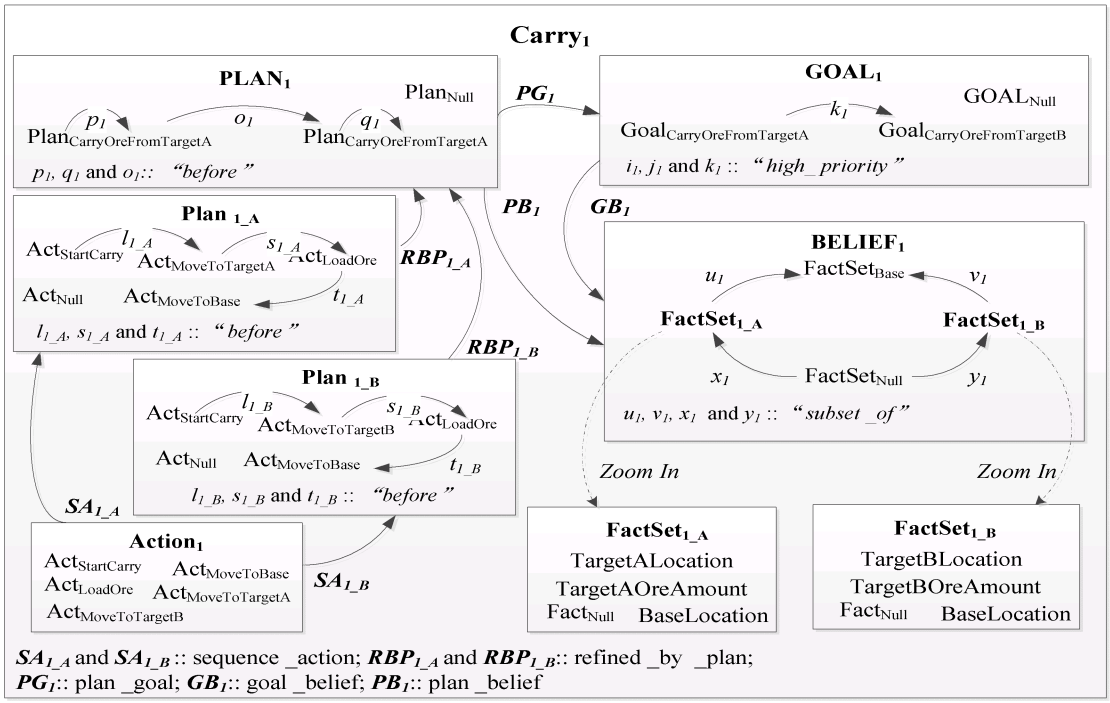


Figure 208: Carry$_1$ Agent [62]

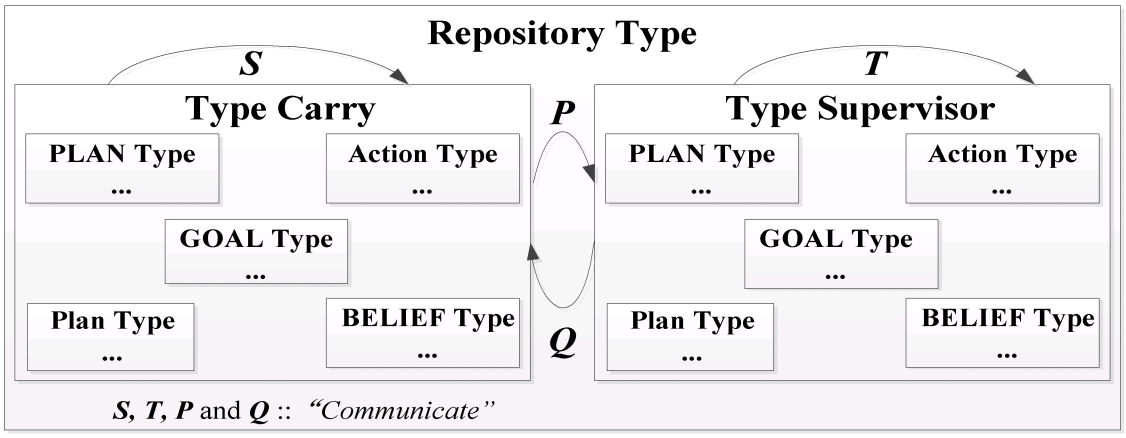


Figure 209: Category **Repository Type** in Mars-world [62]

Property 10.1.1 ***Restart***: *An agent can be restarted, if and only if this agent's categories* ***Action***, ***Plan***, ***PLAN***, ***GOAL***, ***FactSet*** and ***BELIEF*** are *isomorphic to* ***repository agent****'s categories. These categories within repository exist as default before the agent is created, and can be updated during system runtime.* If this agent is restart-able, its supervisor agent will recreate the agent, otherwise, the agent's stored categories will be removed from the repository by the supervisor agent. We write *isomorphism (A, B) == TRUE* to indicate that category *A* is isomorphic to category *B*, otherwise we use *isomorphism (A, B) == FALSE* [62].

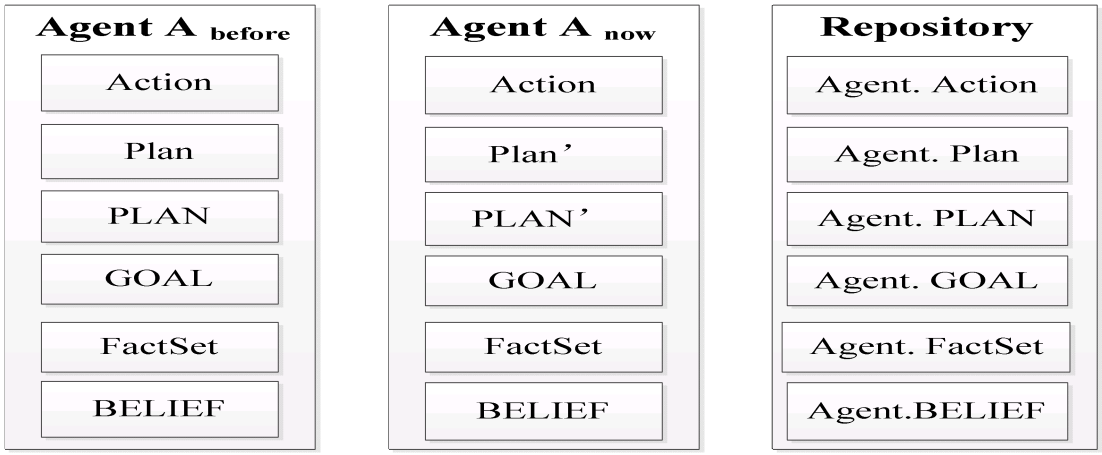


Figure 210: Fault-tolerance Property – Restart in Agent A [62]

Property 10.1.2 ***Inclusion Agent:*** *Let A and B be two agents. If all the following categories:* ***Action, Plan, PLAN, GOAL, FactSet,*** *and* ***BELIEF*** *defined in A* ***include*** *B's* ***Action, Plan, PLAN, GOAL, FactSet,*** *and* ***BELIEF****, we say agent A is an* ***Inclusion Agent*** *of agent B* [62].

Property 10.1.3 ***Takeover:*** *An agent A can take over (i.e. replace) an agent B if and only if IncAgent (A, B) == TRUE* [62].

Figure 211: Include in Action of Mars-world [62]

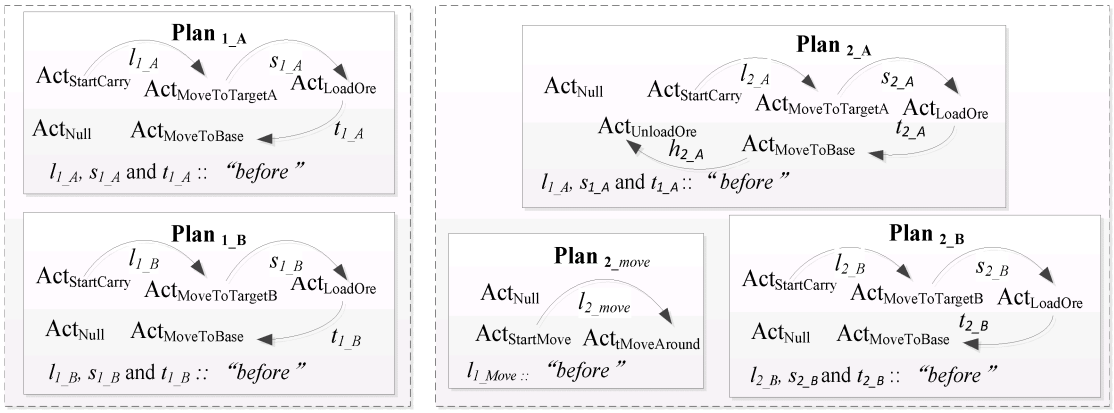


Figure 212: Include in Plan of Mars-world [62]

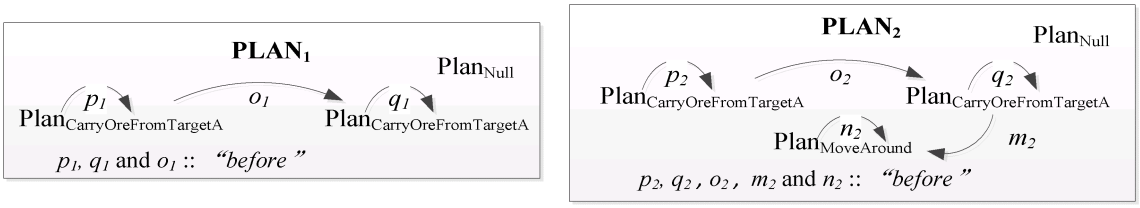


Figure 213: Include in PLAN of Mars-world [62]

Figure 214: Include in GOAL of Mars-world

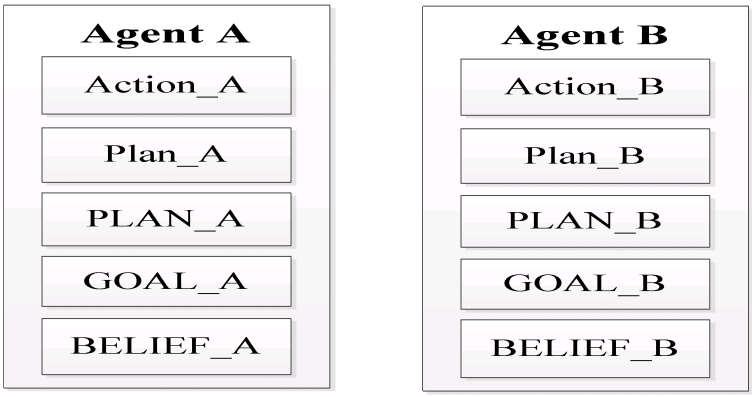


Figure 215: Fault-tolerance Property – Take-over by Inclusion Agent [62]

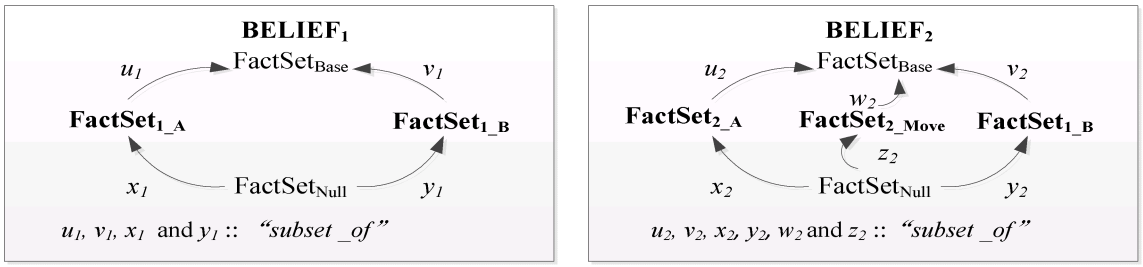


Figure 216: Include in BELIEF of Mars-world

10.1.10 Transform MAS Model of Mars-world to Implementation

In this section, we will introduce the transformation from the MAS model of Mars-world to its implementation (Phase 9 in Section 5.5). More details about this transformation can be found in [148].

To create a new agent, the *ams_create_agent* goal is used. When an agent of a certain type is created, the static initial state of that agent type is recovered automatically. This kind of information can be considered as static initial status of an agent and is recovered

in agent creation phase. There is another type of status information that is dynamic and is changing in time. The recovery of dynamic status of agents is based on the status snapshots taken at the shutdown moment of the agents. In order to simulate the ongoing access of a *Supervisor* agent to the information of its group, there must be a way to inform this *Supervisor* of the current status of its agents. The *Supervisor* polls the current agent snapshot from the agent snapshot queue and creates a message event *request_ location_recovery* and sends this message to the new created *Carry* agent without waiting for the reply from its side [148].

When a message event is received by the new *Carry* agent, it can trigger its *Recover_ Location_Plan*, where the agent is waiting for the *request_location_recovery* message and once received, it restores the location of the agent from agent snapshot and sets the current location of the agent to this value and then creates the *walk_around* goal to start the walking of the agent from this location. The *walk_around* goal is a perform goal that is followed by the agent when there is nothing to do. When an agent is moving around, it can find new sources of ore and inform the *Supervisor* of their existence. This *walk_ around* goal is inhibited if in the next recovery step the *carry_ore* goal is recovered as the latter has a higher priority [148].

The current plan that the agent is pursuing must be recovered. The only way to get a snapshot of the plan execution is to store useful variables from different steps of the plan (commit and rollback). To recover the current goal, the *Supervisor* takes advantage of the *GoalSnapshot* stack in *AgentSnapshot* class. The *Supervisor* agent creates a *request_*

goal_recovery message event and puts the *GoalSnapshot* as its content and sends the request to the new *Carry* agent. If there is no goal to recover, the value of *null* is set as the goal to recover. After sending the request, the *Supervisor* agent waits for the reply from the *Carry* agent to check if it has finished its recovery process. This is done by using the *sendMessageAndWait* method to establish a conversation between those two agents. The reason is that the new *Carry* agent has to finish the unfinished goal of the damaged agent before moving to its message event queue to pick an event message to start a new *carry_ore* goal. By establishing a conversation between those two agents and waiting for the reply, the recovery plan in the *Supervisor* side is suspended until a response comes back from the *Carry* agent [148].

On receiving the request for the goal recovery, the *Carry* agent triggers its plan to recover a goal using the goal snapshot information, by which the *Carry* agent identifies the step that the damaged agent was executing when a problem happened. In *Recover_Goal_Plan*, the *Carry* agent restores the target location as well as ore load from the goal snapshot. The *Recover_Goal_Plan* is a special copy of *CarryOrePlan* with a facility of the conditional entrance points according to the variable checkpoints. After finishing a goal, the *Carry* agent pops the goal from the stack. After finishing this task, the *Carry* agent creates its *carry* goal which listens to the *request_carry* message events. These message events can be from *Supervisor* agent that is recovering the message event queue of the damaged agent or from *Producer* agents as expected in the normal behavior of the system. In this point that the *Carry* agent is listening to *request_carry* message events,

the *Supervisor* can start the recovery of the message events. Thus, the *Carry* agent creates a reply message event named *reply_goal_recovery* in response to the message *request_goal_recovery* of the *Supervisor* agent [148].

10.2 Prospecting Asteroid Mission

In order to support the versatility and flexibility of applying our research outcome, we also select the Prospecting Asteroid Mission (PAM) case study as another application modeled by the RASF. Moreover, we use PAM to illustrate how to visualize categorical RAS model, MAS model as well as implementation (Phase 5, 6 & 11 in Section 5.5) for practical usages. However, we keep a simplified and concise illustration for this case study comparing to the Mars-world case study to avoid redundancy.

The PAM spacecraft study a selected target by particular classes of measurers called virtual teams. For example, an *experiment* team consists of the specialist classes to solve particular scientific problems, such as Petrologist team. The system elements include *generals*, *rulers*, *workers*, and *messengers* (see Figure 19 in Section 4.2). More details about the PAM scenarios can be found in Section 4.2. The specification of fault-tolerance in a Petrologist team is discussed as follows.

A Petrologist team (**RACG1**) has a *ruler* (**RACS1**), an *imaging worker* (**RAC1**), an *X-ray worker* (**RAC2**) and a *messenger* (**RAC3**). *Control Unit* (*CU*, such as *RAOL1*, *RAOL2* & *RAOL3*) and *sail* (*RAO2*, *RAO4*, *RAO5*, *RAO6* as well as *RAO7* are two common devices of each spacecraft. Moreover, different types of *workers* have particular equipments. For instance, *imaging workers* has *Imaging Devices* (*ID*, such as *RAO1*);

X-ray workers has *X-ray Devices* (*XD*, such as *RAO3*). After a Petrologist team is formed and then is sent to explore an asteroid by its *general* (**RACGM1**) of a swarm (**RAS1**), the *general* starts to monitor the *ruler* that monitors its *imaging worker*, *X-ray worker* and *messenger*; moreover, the *CU* in each spacecraft monitors its devices (see Figure 217).

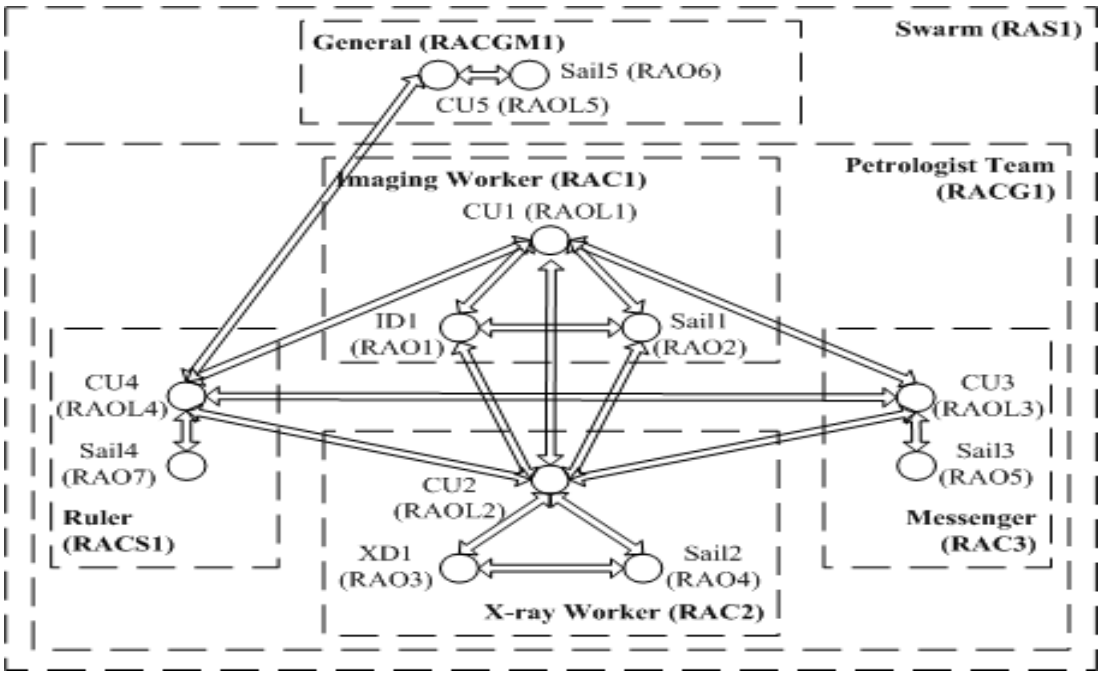


Figure 217: Example of PAM Modeled using RASF

If the *CU* of an *imaging worker* doesn't receive the *heart beat* messages from its *ID* in a required interval, it assumes *ID* is crashed and sends a *restart* message to that *ID*. If the *ID* is restarted successfully and works normally, *CU* continues to monitor its devices as usual; otherwise, *CU* asks its *ruler* to substitute the crashed *ID* with isomorphic one to ensure equivalent behavior. If *ruler* cannot find required *ID* for substitution, the *take-over* plan proposed in section 4 is not applicable either, since the product device of *ID* and *sail* doesn't exist in PAM. Therefore, *ruler* has to replace the whole *imaging worker* with the isomorphic one, which is the case of crashed RAC (see the figure below).

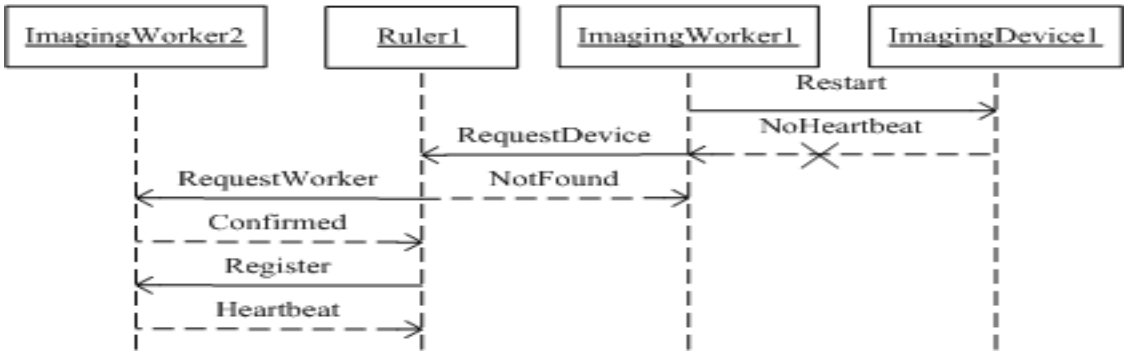


Figure 218: Substitution Work Flow of Imaging Worker

Similarly, if *ruler* doesn't receive *heartbeat* from the *CU* of its *X-ray worker* within a required interval (scenario 2), it sends a *restart* to that *CU*. If the *CU* cannot be restarted, *ruler* substitutes the crashed *CU* with isomorphic one to ensure their equivalent behavior. If *ruler* cannot find required *CU* for substitution, it asks another *worker*, such as *imaging worker* to take over the control of *XD* and *sail* on the crashed *X-ray worker*. If all other workers are not available, *ruler* takes over the *CU* by itself. However, if *ruler* cannot take over *CU* either, it has to replace the whole *X-ray worker* with the isomorphic one [87].

10.2.1 CML Model for Sub-swarm Organization in PAM

Figure 219 & 220 include the CML specification (see Section 6.4.1) and its visual model respectively for a typical sub-swarm organization. More details can be found in [81].

TYPED-CATEGORY	***Type of Morphisms***
PAM Sub-swarm (S_1)	Morphism_Type: Management (m):
Types of Objects	L → TM, L → SM, L → W
Object_Type: Ruler (R)	Morphism_Type: Cooperation (c):
Object_Type_Instances: Leader (L)	W → W, TM → W, W → TM
Object_Type: Messenger (M)	Morphism_Type: Communication (cu):
Object_Type_Instances: Team Messenger (TM),	TM → SM, TM → TM , W → SM, TM → L
Sub-Swarm Messenger (SM)	***Morphisms: Mor(S_1)***
Object_Type: Worker (W)	m_1 (L_1) = TM_2, m_2 (L_1) = SM_1,
Object_Type_Instances: X-Ray (W_{XR}),	m_3 (L_1) = W_{XR1}, m_4 (L_1) = W_{GR1},

Gamma Ray(W_{GR}), Infra-Red(W_{IR}), Altimeter(W_{AL})	m_5 (L_1) = W_{IR1}, m_7 (L_1) = TM_1, c_1 (W_{XR1}) = W_{GR1}, c_2 (TM_2) = W_{XR1}, c_3 (W_{IR1}) = W_{GR1}, c_4 (TM_2) = W_{IR1}, c_5 (TM_2) = W_{GR1}, c_6 (W_{AL1}) = TM_1, c_7 (W_{XR3}) = TM_1, c_8 (W_{XR3}) = W_{AL1}, cu_1 (TM_2) = TM_1, cu_2 (TM_1) = SM_1, cu_3 (TM_2) = SM_1, cu_4 (W_{XR3}) = SM_1, cu_5 (W_{AL1}) = SM_1, cu_8 (TM_2) = L_1
Objects: Obj(S_1) R: L_1 TM: TM_1, TM_2 SM: SM_1 W_{XR}: W_{XR1}, W_{XR3} W_{GR} : W_{GR1} W_{IR} : W_{IR1} W_{IR} : W_{AL1}	***Identity: Identity(S_1)*** *Id*(L_1): $L_1 \rightarrow L_1$, *Id*(SM_1): $SM_1 \rightarrow SM_1$, *Id*(TM_1): $TM_1 \rightarrow TM_1$, *Id*(TM_2): $TM_2 \rightarrow TM_2$, *Id*(W_{XR1}): $W_{XR1} \rightarrow W_{XR1}$, *Id*($W_{XR3}$): $W_{XR3} \rightarrow W_{XR3}$, *Id*($W_{AL1}$): $W_{AL1} \rightarrow W_{AL1}$, *Id*($W_{GR1}$): $W_{GR1} \rightarrow W_{GR1}$, *Id*($W_{IR1}$): $W_{IR1} \rightarrow W_{IR1}$
Composition (c_3 o m_5) = m_4, (c_1 o m_3) = m_4, (c_4 o m_1) = m_5, (c_2 o m_1) = m_3, (c_1 o c_2) = c_5, (cu_1 o m_1) = m_7, (cu_2 o m_7) = m_2, (m_2 o cu_8) = cu_3, (m_3 o cu_8) = c_2, (m_4 o cu_8) = c_5, (m_5 o cu_8) = c_4, (cu_2 o c_7) = cu_4, (cu_5 o c_8) = cu_4, (c_6 o c_8) = c_7, (cu_2 o c_6) = cu_5, (c_3 o c_4) = c_5, (cu_2 o cu_1) = cu_3, (cu_3 o m_1) = m_2, (m_7 o cu_8) = cu_1	***Axioms*** Identity: $\forall x \in Identity(S_1)$, $y \in Mor(S_1)$, $x \circ y = y = y \circ x$ Associativity: $c_1 \circ (c_2 \circ m_1) = (c_1 \circ c_2) \circ m_1$ $c_3 \circ (c_4 \circ m_1) = (c_3 \circ c_4) \circ m_1$ $c_1 \circ (m_3 \circ cu_8) = (c_1 \circ m_3) \circ cu_8$ $c_3 \circ (m_5 \circ cu_8) = (c_3 \circ m_5) \circ cu_8$ $cu_2 \circ (cu_1 \circ m_1) = (cu_2 \circ cu_1) \circ m_1$ $cu_2 \circ (m_7 \circ cu_8) = (cu_2 \circ m_7) \circ cu_8$ $cu_2 \circ (c_6 \circ c_8) = (cu_2 \circ c_6) \circ c_8$

Figure 219: CML Specification Model of a PAM Swarm Scenario [81]

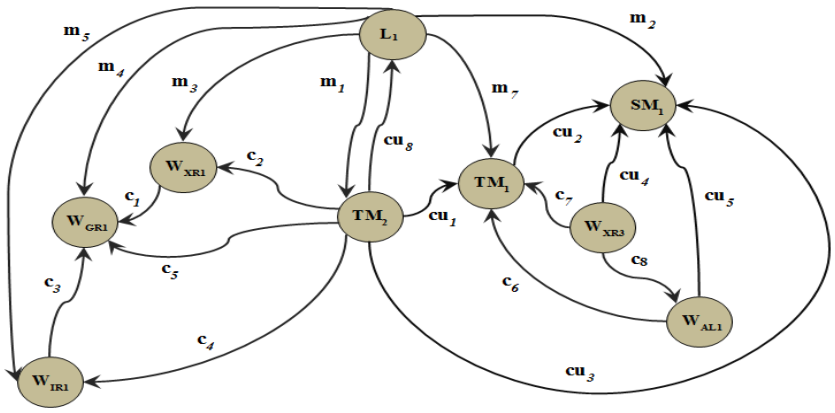


Figure 220: CML Graphical Model of a PAM Swarm Scenario [81]

10.2.2 Self-Configuration in PAM

The virtual teams of spacecraft are configured to carry out optimal science operations on

the target asteroids. When the operations are complete, the team breaks up for possible reconfiguration at another asteroid site. This reconfiguring continues throughout the life of the swarm. Reconfiguring may also be required as the result of a failure or anomaly of some sort. The specification in Figure 221 captures the behavior of a team relocating to a new position in the sub-swarm and its graphical model is given in Figure 222 [81].

Categories:	***Morphisms: Mor(S_2)***
TYPED-CATEGORY	m_7 (L_1) = SM_1, c_{11} (W_{H1}) = W_{IR3},
Petrologist Team (PT_1) …	c_{12} (W_{RS1}) = W_{IR3}, c_9 (W_{H1}) = W_{RS1},
TYPED-CATEGORY	c_{10} (W_{H1}) = W_{IM3}, c_8 (W_{RS1}) = W_{IM3},
PAM Sub-swarm (S_2)	cu_{10} (W_{RS1}) = SM_1, cu_{11} (W_{RS1}) = SM_3,
Types of Objects	cu_{12} (SM_3) = SM_1, cu_{13} (W_{H1}) = SM_1
Object_Type: Ruler (R)	***Identity: Identity(S_2)***
Object_Type_Instances: Leader (L)	*Id*(L_1): $L_1 \rightarrow L_1$, *Id*(SM_1): $SM_1 \rightarrow SM_1$,
Object_Type: Messenger (M)	*Id*(SM_3): $SM_3 \rightarrow SM_3$, *Id*(W_{RS1}): $W_{RS1} \rightarrow W_{RS1}$, *Id*($W_{H1}$): $W_{H1} \rightarrow W_{H1}$, *Id*($W_{IM3}$): $W_{IM3} \rightarrow W_{IM3}$, *Id*($W_{IR3}$): $W_{IR3} \rightarrow W_{IR3}$
Object_Type_Instances: Team Messenger (TM), Sub-Swarm Messenger (SM)	***Composition***
Object_Type: Worker (W)	(c_8 o c_9) = c_{10}, (c_{12} o c_9) = c_{11},
Object_Type_Instances: Radio Sound (W_{RS}), Imager(W_{IM}), Infra-Red(W_{IR}), Helper(W_H)	(m_7 o du_1) = cu_{13}, (cu_{10} o c_9) = cu_{13},
	(cu_{11} o c_9) = cu_{14}, (cu_{12} o cu_{11}) = cu_{10},
	(cu_{12} o cu_{14}) = cu_{13}, (c_{15} o cu_{11}) = cu_{13},
	(cu_{13} o cu_9) = cu_{14}, (c_{13} o c_9) = c_{16},
Objects: Obj(S_2)	(cu_{15} o cu_{14}) = cu_{16}
R: L_1	***Axioms***
SM: SM_1, SM_3	Identity: $\forall$ x $\in$ *Identity*(S_2) , y $\in$ *Mor*(S_2), x o y = y = y o x
W_{RS}: W_{RS1}	
W_{IM} : W_{IM3}	
W_{IR} : W_{IR3}	Associativity:
W_H : W_{H1}	cu_{12} o (cu_{11} o c_9) = (cu_{12} o cu_{11}) o c_9
Type of Morphisms	cu_{15} o (cu_{11} o c_9) = (cu_{15} o cu_{11}) o c_9
Morphism_Type: Data Update (du):	***Category Source***: PT_1
W → L	***Category Target*** : S_2
Morphism_Type: Management (m):	**FUNCTOR** (Team Relocation, R, PT_1, S_2)
L → SM	R(c_3): PT_1 (W_{IR1}, W_{GR1}, c_3) →
Morphism_Type: Cooperation (c):	S_2(R(W_{IR1}): W_{IR3}, R(W_{GR1}) : W_{IR3}, R(c_3): *Id*(W_{IR3})),
W → W	R(c_4): PT_1 (TM_2, W_{IR1}, c_4) →
Morphism_Type: Communication (cu):	S_2(R(TM_2): W_{H1} , R(W_{IR1}): W_{IR3}, R(c_4): c_{11}),

SM → SM, W → SM , SM → L	$R(c_5)$: PT_1 (TM_2, W_{GR1}, c_5) →
Functor Objects	$S_2(R(TM_2)$: W_{H1}, $R(W_{GR1})$: W_{IR3}, $R(c_5)$: $c_{11})$
Messenger:	***Functor Composition***
$R(TM_2)$: PT_1 (TM_2) → S_2 ($R(TM_2)$:W_{H1})	$R(c_3 \circ c_4) = R(c_3) \circ R(c_4) = R(c_5)$
X-Ray:	$R(c_1 \circ c_2) = R(c_1) \circ R(c_2) = R(c_5)$
$R(W_{XR1})$: PT_1 (W_{XR1}) → S_2 ($R(W_{XR1})$:W_{RS1})	***Functor Axioms***
Infra-red:	Identity: $R(Id(TM_2)) = Id(R(TM_2))$
$R(W_{IR1})$: PT_1 (W_{IR1}) → S_2 ($R(W_{IR1})$:W_{IR3})	$R(Id(W_{XR1})) = Id(R(W_{XR1}))$
Gamma Ray:	$R(Id(W_{IR1})) = Id(R(W_{IR1}))$
$R(W_{GR1})$: PT_1 (W_{GR1}) → S_2 ($R(W_{GR1})$:W_{IR3})	$R(Id(W_{GR1})) = Id(R(W_{GR1}))$
Functor Morphisms	
Cooperation:	
$R(c_1)$: PT_1 (W_{XR1}, W_{GR1}, c_1) →	
$S_2(R(W_{XR1})$: W_{RS1}, $R(W_{GR1})$: W_{IR3}, $R(c_1)$: $c_{12})$,	
$R(c_2)$: PT_1 (TM_2, W_{XR1}, c_2) →	
$S_2(R(TM_2)$: W_{H1}, R (W_{XR1}) :W_{RS1}, $R(c_2)$: $c_9)$,	

Figure 221: CML Model for Team Relocation Scenario [81]

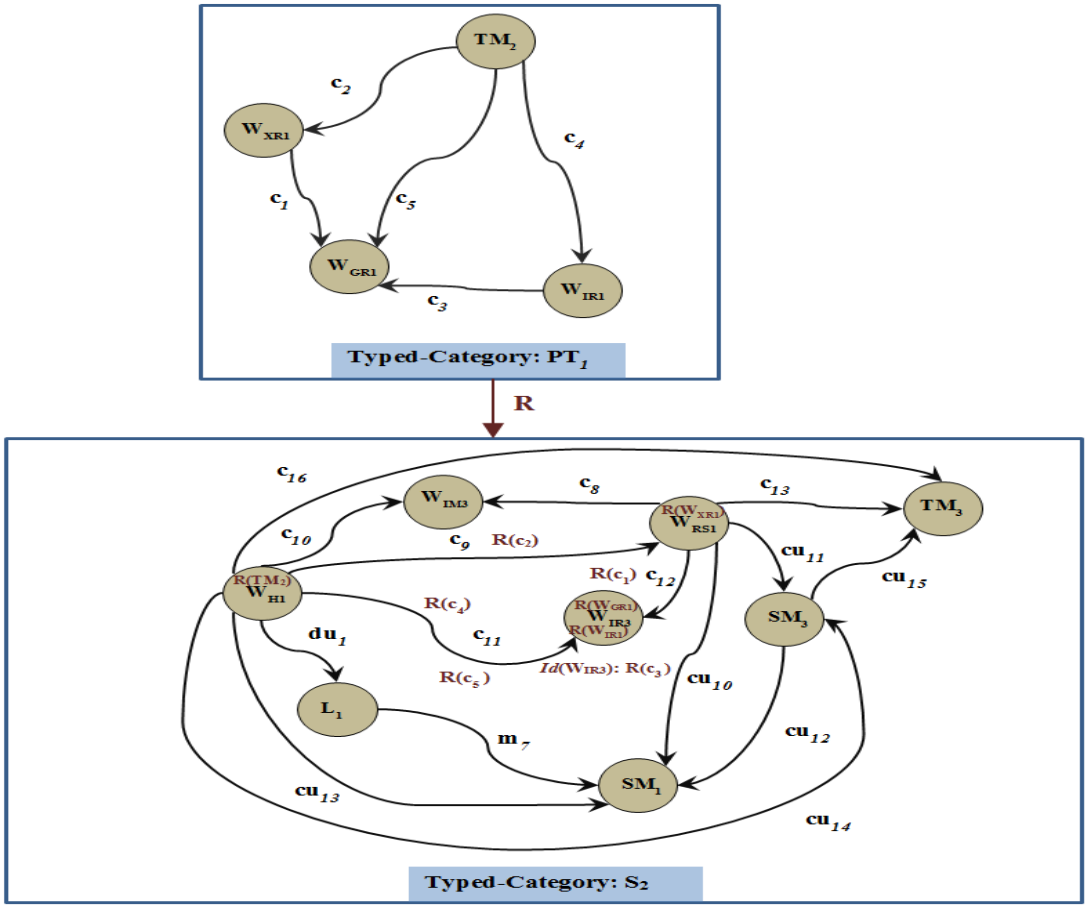


Figure 222: CML Graphical Model for Team Relocation Scenario [81]

10.2.3 Self-Protection in PAM

After receiving the confirmation from the sub-swarm leader regarding a solar storm, a sub-swarm messenger communicates information to the other sub-swarm messengers. All sub-swarm messengers inform their team messengers that in turn inform all the workers in the team. Each spacecraft after receiving a warning message and performing necessary communication puts itself to a "stand by" mode. Figure 223 includes a CML model for this scenario constructed using the limit construct grammar and its graphical model is given in Figure 224 [81].

Diagram:	Communication: D(α): D(k) → D(l)
DIAGRAM (D, IC, S_1)…	D(β): D(j) → D(k)
Cones:	***Category Id:*** S_1
CONE (Object: TM_2)…	***Diagram Id:*** D
CONE (Object: L_1)	***Cone Ids:*** TM_2, L_1
Co-Cone Objects	**LIMIT**
D(k), D(j), D(i),	Terminal Object: TM_2
Co-Cone Morphisms	Unique Morphism(u): m_1: $L_1 \rightarrow TM_2$
Management: $L_1(k)$: $L_1 \rightarrow D(k)$,	***Limit Objects***
$L_1(j)$: $L_1 \rightarrow D(j)$,	D(i), D(j), D(k)
$L_1(i)$: $L_1 \rightarrow D(i)$	***Limit Axioms***
Limit Morphisms	m_1 o $L_1(i)$ = $TM_2(i)$
Management: $L_1(k)$: $L_1 \rightarrow D(k)$,	m_1 o $L_1(j)$ = $TM_2(j)$
$L_1(j)$: $L_1 \rightarrow D(j)$,	m_1 o $L_1(k)$ = $TM_2(k)$
$L_1(i)$: $L_1 \rightarrow D(i)$	
Cooperation: $TM_2(k)$: $TM_2 \rightarrow D(k)$,	
$TM_2(j)$: $TM_2 \rightarrow D(j)$,	
$TM_2(i)$: $TM_2 \rightarrow D(i)$,	
D(α): D(i) → D(j),	
D(β): D(j) → D(k)	

Figure 223: CML Model for PAM Self-Protection [81]

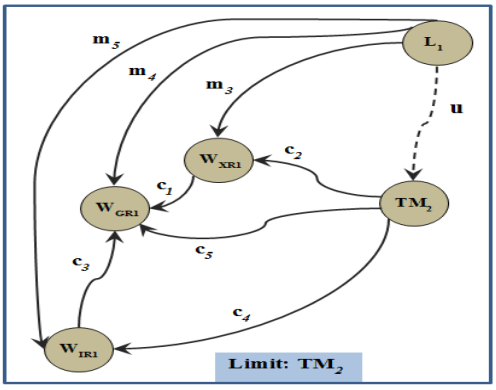


Figure 224: CML Graphical Model for PAM Self-Protection [81]

10.3 End-to-End iFix Tool

In order to support the feasibility of applying our research outcome on industrial projects, I select the End-to-End iFix Tool (E2E) case study as an industrial application modeled by the RASF during my research internship at IBM Canada. We keep a simplified and concise illustration for this case study comparing to the Mars-world case study to avoid redundancy.

The E2E is a web-based application to process official fix creation requests from the IBM support teams. The tool implements an automated and autonomic process to build and test iFixes with minimal user input as well as intervention based on source code for a fix being available and identifiable in a source code repository system. The E2E interacts with a repository tool to store source code and a build tool to compile the source code into object code and a packaging tool to package the object code into an installable fix (see Figure 20 in Section 4.3).

Figure 225 depicts the architecture model of E2E according to the RAS architecture model (see Figure 21 in Section 5.1). This model is a four-layer architecture that consists

of build objects (RAO, such as CMVC session, Aphid request, submit moonstone test and ICT input), build components (RAC, such as CMVC engine, Aphid engine, ICT engine and Moonstone engine), build component groups (RACG, such as back-end support group and front-end support group) and the whole tool itself (RAS). The autonomic features can be implemented by component manager (RAOL, such as CMVC manager, Aphid manager, ICT manager and Moonstone manager), the component group manager (RACS, such as Build manager and GUI manager) as well as tool manager (RACGM, E2E adapter/manager) at the RAC, RACG and RAS layer respectively. In this layered architecture model, each tier communicates with the tier immediately above or below it.

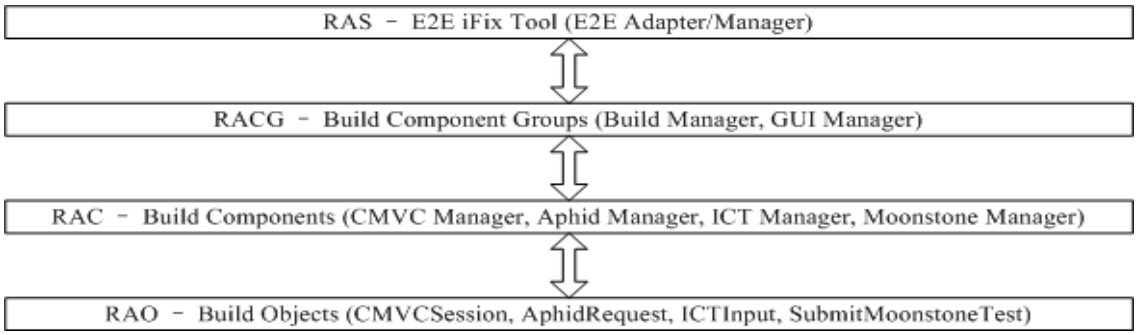


Figure 225: RAS Architecture Model for E2E

Figure 226 depicts an example of the architecture model of E2E for a simplified scenario presented above (see Phase 1 in Section 5.5), where every circle represents an implemented class of the build component and each arrow specifies the procedure call between those classes. In this example, the E2E (RAS1) consists of a director package (RACG3) for tool management, a front-end support package (RACG2) for the business logic related to UI and a back-end support package (RACG1) for the process of building an iFix. The director package includes a component iFix Engine (RACGM1) which has

two classes: E2E Session (RAO7) and E2E Manager (RAOL5). The front-end support package consists of two components: Front-end Engine (RACS2) with two classes GUI Adapter (RAOL7), Build Servlet (RAO9) and GUI Engine (RAC4) with two classes GUI Manager (RAOL6) as well as GUI Data (RAO8). In back-end support package, there are four components Back-end Engine (RACS1), CMVC Engine (RAC1), Aphid Engine (RAC2) and ICT Engine (RAC3) with their classes, such as Build Manager (RAOL4), CMVC Manager (RAOL1), Aphid Manager (RAOL2) and ICT Manager (RAOL3).

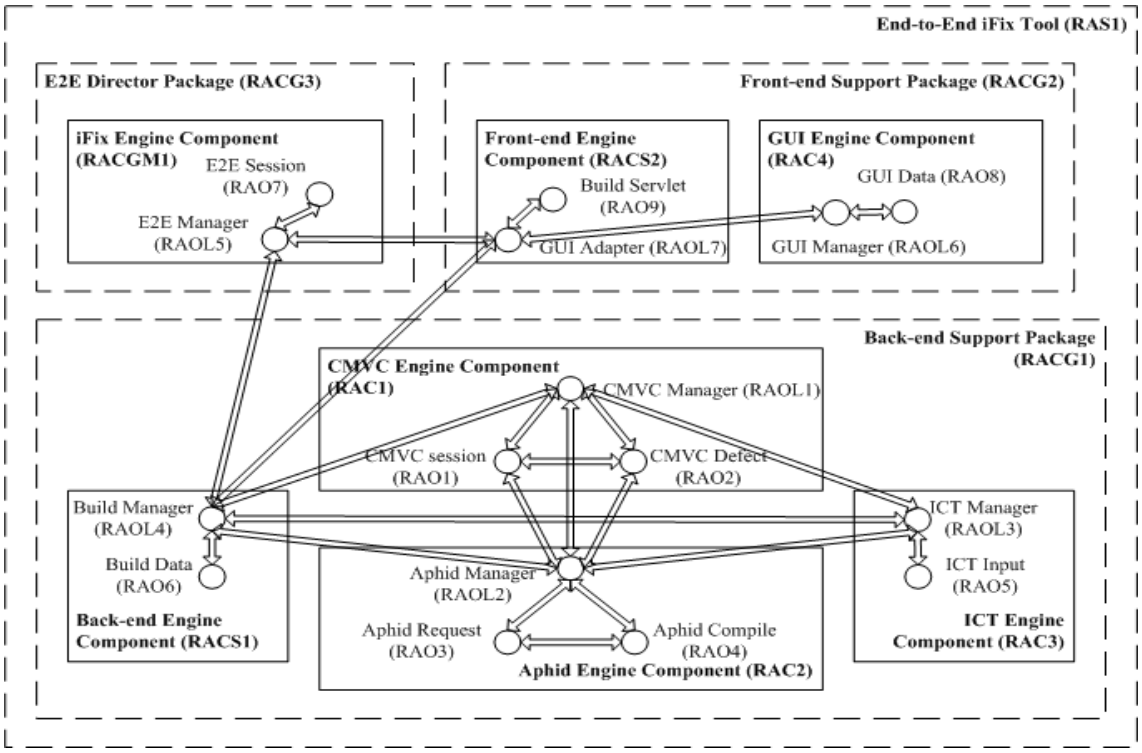


Figure 226: Example of E2E Modeled using RASF

10.3.1 Self-Healing in E2E

After E2E is deployed to the application server, E2E Manager starts to monitor the heart messages sent from GUI Adapter and Build Manager. If there is any exceptions thrown from them that might cause termination of their services, E2E Manager can catch them

and retrieve the self-healing process as we described in Chapter 7, either restarting the services from those two classes by recalling corresponding methods, or substituting them by creating the new instances of those classes.

Another scenario of self-healing is the fault-tolerance when submitting source code compiling requests to Aphid servers. For example, if Aphid Manger receives bad request messages from Aphid servers through Aphid Request due to the unavailability of those servers in the middle of compiling, it can automatically redirect the original requests to another available Aphid server without the intervention of end users.

10.3.2 Self-Configuration in E2E

Build Manager is responsible for monitoring the configuration, communication as well as behavior of the components CMVC Engine, Aphid Engine and ICT Engine by checking the status of CMVC Manager, Aphid Manager and ICT Manager against their meta-model respectively. If there is any incorrect configuration or unexpected behavior from those components, the Build Manager will retrieve the self-configuration process as we stated in Chapter 8.

Another scenario of self-configuration is that E2E is deployed on four nodes, and the iFix building requests from end users can be automatically redirected to an optimal node. The necessary resource and configuration on that node to build an iFix can be acquired automatically based on the meta-model of E2E without the intervention of administrators.

10.4 Summary

In this chapter, we illustrated how to apply RASF approach to three different case studies.

For the Mars-world, we presented a complete process of using RASF approach based on the process model in Section 5.5, which include the architecture model, self-healing and self-configuration properties, categorical models of structure, behavior as well as those properties in Mars-world. For the self-healing property, we showed all possible scenarios at each tier in the architecture model, such as crashed sensors (RAO), crashed control units (RAOL), crashed robots (RAC) and crashed supervisor robots (RACS). For the self-configuration property, we also illustrated all possible scenarios according to the architecture model, such as forming Mars-world, forming exploration groups and forming robots.

For the PAM, we focused on the visualization of the categorical models generated from the RAS model and MAS model using our graphical illustration tool CATCanvas. That visualization can help us to achieve the validation between those categorical models besides the verification between them textually. Moreover, it can help IT professionals with minimum category theory knowledge to better understand those categorical models. It can also help either IT professionals or category theory experts to manipulate the categorical diagrams in terms of reasoning.

Finally, we introduced how to apply the RAS model on the E2E, an industrial project from the IBM Toronto Lab, to support the flexibility and feasibility to use our RASF approach.

Chapter 11: Conclusion and Future Work

In this chapter, we conclude this thesis work by presenting the significance of the RASF approach, contributions, concluding remarks and future work.

11.1 Significance of RASF Approach

We conducted a comprehensive literature review (see Chapter 2) on autonomic systems, multi-agent systems, real-time reactive systems and formal methods. To the best of our knowledge, there is no similar integrated formal framework targeting the whole life cycle of developing reactive autonomic systems, from requirement specification, architecture model (structure and behavior), meta-model (constructed from the architecture model), instance model (generated from the meta-model) and implementation model (the MAS model and implemented from the instance model) to formal specification (using category theory), visualization of the categorical model (using CATCanvas), model transformation between those models, model checking (verification & validation) and tooling support (RASFIT).

Furthermore, to the best of our knowledge, our RASF approach is the first attempt to formally specify the autonomic systems with self-* properties, multi-agent systems with goals, plans as well as beliefs and real-time reactive systems with time constraints using category theory (a relatively young branch of the mathematics has been successfully extended to the fields of computer science and software engineering). Category theory as a formal method can address the characteristics of reactive autonomic systems as well as multi-agent systems very well because: i) it offers a specification structure that can isolate

analysis of changes in a small number of components and analyze impacts of a change on inter- connected components; ii) it has a rich body of theory to reason objects and their relations; iii) it adopts a *correct by construction* approach by which components can be specified, proved and composed in the way of preserving their properties.

Domain theory is introduced as a study of special kinds of partially ordered sets (or posets) in mathematics, those sets are called domains. A partially ordered set (poset) can formalize and generalize the intuitive concept of an ordering, sequencing or arrangement of the elements for a set. "Partial order" means that not every pair of elements needs to be related: for some pairs, it may be that neither element precedes the other in the same poset. In comparison to category theory, it has a limitation of capturing all kinds of the relations between posets, such as "*depends on*", since it is not a ordering, sequencing or arrangement relation. Domain theory cannot be used to model self-relation of elements in a poset, which is well defined as the identity morphism in category theory.

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

Comparing to other formal approaches which either lack of emergent and self-management behavior specification, verification as well as validation (CSP, Temporal Logic, Unity Logic, ASSL and PTN), or lack of visualization (CSP, WSCCS, Temporal Logic, Unity Logic and ASSL) or tooling support (WSCCS, Temporal Logic, X-Machine, Unity Logic and PTN), our RASF approach supports each aspect of specifying, verifying

and validating reactive autonomic systems, such as formal basis, visual formalism, adaptability to programming, tooling support, modularity and self-management behavior.

11.2 Contributions

This thesis work proposed a formal framework (RASF) which can leverage modeling, formal specification as well as development of the RAS. The main contributions of this thesis work are summarized below:

- Reactive Autonomic Systems Framework that includes the architecture model with structure (Section 5.1) and behavior (Section 5.2) as well as process model (Section 5.5). I had one publication [90] for this contribution.
- Categorical RAS model in RASF that includes the categorical model of structure (Section 6.1), behavior (Section 6.2) and their XML representations (Section 6.3). I had one publication [90] for this contribution.
- XML representation of the categorical MAS model (plans, goals, beliefs and agents) in RASF (Section 6.6). I had one publication in preparation for this contribution.
- Specification of the self-healing in RASF which includes three scenarios (Section 7.1, 7.3 & 7.5) with their categorical illustration (Section 7.2, 7.4 & 7.6), categorical specifications of self-healing properties (Section 7.7) and their XML representations (Section 7.8). I had two publications [177 & 87] for this contribution.
- Specification of the self-configuration in RASF which includes three scenarios

(Section 8.1, 8.3 & 8.5) with their categorical illustration (Section 8.2, 8.4 & 8.6), categorical specifications of self-configuration properties (Section 8.7) as well as their XML representations (Section 8.8). I had one publication in preparation for this contribution.

- RASF integration tool (RASFIT) as a plug-in of Eclipse to support the RASF approach and apply the RASF process model (Chapter 9). I had one publication [85] and one in preparation during this contribution.
- Modeling, specification and design of case studies using RASF (Section 10.1.1) with self-healing (Section 10.1.2), self-configuration (10.1.3) and their categorical models (Section 10.1.4 – 10.1.7). I had one publication [86] and one in preparation for this contribution.
- Applied our RASF approach to three industrial projects: End-to-End iFix Tool, Integrated Data Access Tool as well as Rational Team Concert Validation Tool. Moreover, we are preparing the application of NSERC's Engage Grant with IBM Centers for Advanced Studies at IBM Toronto Lab based on one of those three projects.
- Joint work with two master students for mapping the RAS model to the MAS model (Section 5.3) and model transformation from RAS to MAS implementation (Section 5.4).
- Joint work with two master students for the graphical illustration (CATCanvas) of categorical models (Section 6.4) and categorical specifications of the MAS model

(Section 6.5).

- Joint work with two master students for transforming the RAS models of case studies to their MAS models (Section 10.1.8), transforming the MAS models to their categorical models (Section 10.1.9) and transforming the MAS models to their implementations (Section 10.1.10).

11.3 Challenges of RASF Approach

RASF approach still faces some challenges at current stage which might be overcome through our future work. For example, 1) how RASF approach can seamlessly fill the knowledge and technical gaps between IT professionals and category theory experts, so that both of them can use RASF freely from their own perspective without knowing too much details about the other side; 2) how RASF approach can be widely accepted and easily applied to the industrial projects, since category theory is relatively abstract and there are very few industrial research projects on using it right now.

Another challenge is how RASF implements a formal reasoning mechanism to reason about new properties from existing ones, either from the XML representations of the categorical models or from the visualization of those models. Moreover, we can discover flaws for the various models in RASF by that formal reasoning mechanism.

11.4 Future Work

Some of the future extensions to this thesis work include the following aspects:

- Our approach focuses on the self-healing and self-configuration properties in RAS. However, self-* properties of autonomic systems include self-optimization, self-

protection, etc., a future direction would be the extension of our approach to address those properties.

- We need to enhance RASFIT in terms of verifying if the categorical MAS model generated in Phase 4 (see Phase 7 in Section 5.5) conforms to the categorical RAS model in Phase 2.
- We need to enhance RASFIT in terms of validating if the graphical illustration of categorical MAS model in Phase 6 (see Phase 8 in Section 5.5) conforms to the graphical illustration of categorical RAS model in Phase 5.
- We need to support the transformation from the MAS implementation in Phase 9 (see Phase 10 in Section 5.5) to its categorical model using category theory.
- We need to support the visualization of categorical MAS implementation in Phase 10 (see Phase 11 in Section 5.5) to its graphical representation by importing its XML representation in Phase 10 to our graphical illustration tool CATCanvas.
- We need to enhance RASFIT for verifying if the categorical MAS implementation in Phase 10 (see Phase 12 in Section 5.5) conforms to the categorical MAS model in Phase 4.
- We need to enhance RASFIT for validating if the graphical illustration of the categorical MAS implementation in Phase 11 (see the Phase 13 in Section 5.5) conforms to the graphical illustration of the categorical MAS model in Phase 6.

References

[1] S. Abdelwahed and N. Kandasamy, "A Control-Based Approach to Autonomic Performance Management in Computing Systems", *Autonomic Computing: Concepts, Infrastructure, and Applications*, CRC Press, December 2006, Page 149 – 167.

[2] M. Agarwal, V. Bhat, H. Liu, V. Matossian, V. Putty, C. Schmidt, G. Zhang, L. Zhen, M. Parashar, B. Khargharia, S. Hariri, "AutoMate: Enabling Autonomic Applications on the Grid", Autonomic Computing Workshop, June 2003, Page 48 – 57.

[3] M. Agarwal, M. Parashar, "Enabling Autonomic Compositions in Grid Environment", Proceedings of the 4th International Workshop on Grid Computing, November 2003, Page 34 – 41.

[4] J. Appavoo, K. Hui, C. A. N. Soules, R. W. Wisniewski, D. M. Da Silva, O. Krieger, M. A. Auslander, D. J. Edelsohn, B. Gamsa, G. R. Gagner, P. McKennedy, M. Ostrowski, B. Rosenburg, M. Stumm, and J. Xenidis, "Enabling autonomic behavior in systems software with hot swaping", IBM Systems Journal, Volume 42, No.1, January 2003, Page 60 – 76.

[5] M. A. Arbib and E. G. Manes, "Machines in a Category: An Expository Introduction", SIAM Review, Volume 16, No. 2, April 1974, Page 163 – 192.

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

[7] R. Barrett, P. P. Maglio, E. Kandogan, and J. Bailey, "Usable autonomic Computing Systems: the Administrator's Perspective", Proceedings of the 1st International Conference on Autonomic Computing, May 2004, Page 18 – 26.

[8] F. Bellifemine, G. Caire, and D. Greenwood, *Developing Multi-Agent Systems with JADE*, John Wiley & Sons, April 2007.

[9] J. Bentahar, Z. Maamar, D. Benslimane, and P. Thiran, "An Argumentation Framework for Communities of Web Services", IEEE Intelligent Systems, Volume 22, Issue 6, November 2007, Page 75 – 83.

[10] J. Bentahar, "A Pragmatic and Semantic Unified Framework for Agent Communication", PhD Thesis, Department of Computer Science and Software Engineering, Laval University, Laval, Quebec, Canada, May 2005.

[11] P. Besnard and A. Hunter, "A Logic-Based Theory of Deductive Arguments", Artificial Intelligence, Volume 128, Issue 1-2, May 2001, Page 203 – 235.

[12] A. Beygelmizer, G. Grinstein, R. Linsker, I. Rish, "Improving Network Robustness", Proceedings of the 1st International Conference on Autonomic Computing, May 2004, Page 322 – 323.

[13] V. Bhat, M. Parashar, and N. Kandasamy, "Autonomic Data Streaming for High-Performance Scientific Applications", *Autonomic Computing: Concepts, Infrastructure, and Applications*, CRC Press, December 2006, Page 413 – 433.

[14] J. P. Bigus, D. A. Schlosnagle, J. R. Pilgrim, W. N. Mills III, and Y. Diao, "ABLE: A Toolkit for Building Multi Agent Autonomic System", IBM Systems Journal, Volume 41, No.3, September 2002, Page 350 – 371.

[15] D. Bonino, A. Bosca, and F. Corno, "An Agent Based Autonomic Semantic Platform", Proceedings of the 1st International Conference on Autonomic Computing,

May 2004, Page 189 – 196.

[16] C. Boutilier, R. Das, G. Tesauro, J. O. Kephart, and W. E. Walsh, "Cooperative Negotiation in Autonomic Systems using Incremental Utility Elicitation", Proceedings of the 19th Conference in Uncertainty in Artificial Intelligence, August 2003, Page 89 – 97.

[17] M. Bratman, *Intention, Plans, and Practical Reason*, Harvard University Press, November 1987.

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

[19] A. J. Chakravarti, G. Baumgartner, M. Lauria, "The Organic Grid: Self-Organizing Computation on a Peer-to-Peer Network", Proceedings of the 1st International Conference on Autonomic Computing, May 2004, Page 96 – 103.

[20] M. E. Bratman, "Planning and the Stability of Intentions", Journal of Minds and Machines, Volume 2, No. 1, February 1992, Page 1 – 16.

[21] F. Brazier, B. D. Keplicz, N. R. Jennings, and J. Treur, "Formal Specification of Multi-Agent Systems: a Real-World Case", Proceedings of the 1st International Conference on Multi-Agent Systems, June 1995, Page 25 – 32.

[22] R. A. Brooks, "A Robust Layered Control System for a Mobile Robot", IEEE Journal of Robotics and Automation, Volume 2, Issue 1, March 1986, Page 14 – 23.

[23] L. Chen, G. Agrawal, "Self-Adaptation in a Middleware for Processing Data Stream", Proceedings of the 1st International Conference on Autonomic Computing, May 2004,

Page 292 – 293.

[24] D. M. Chess, C. C. Palmer, and S. R. White, “Security in an Autonomic Computing Environment”, IBM Systems Journal, Volume 42, No.1, January, 2003, Page 107 – 118.

[25] P. E. Clark, M. L. Rilee, W. Truszkowski, G. Marr, S. A. Curtis, C. Y. Cheung, and M. Rudisill, “PAM: Biologically Inspired Engineering and Exploration Mission Concept, Components, and Requirements for Asteroid Population Survey”, Proceedings of the 55th International Astronautical Congress, October 2004, IAC-04-Q5.07.

[26] I. Croitoru, “Autonomic Systems Modeling Development: A Survey”, Master Major Report, Department of Computer Science & Software Engineering, Concordia University, Montreal, Quebec, Canada, April 2006.

[27] S. Curtis, J. Mica, J. Nuth, G. Marr, M. Rilee, and M. Bhat, “ANTS (Autonomous Nano Technology Swarm): an Artificial Intelligence Approach to Asteroid Belt Resource Exploration”, Proceedings of the 51st International Astronautical Congress, October 2000, IAA-00-IAA.Q.5.08.

[28] K. Decker and V. Lesser, “Designing a Family of Coordination Algorithms”, Proceedings of the 1st International Conference on Multi-Agent Systems, June 1995, Page 73 – 80.

[29] R. Depke, R. Heckel, and J. M. Kuster, “Formal Agent-Oriented Modeling with UML and Graph Transformation”, Science of Computer Programming, Volume 44, Issue 2, August 2002, Page 229 – 252.

[30] Y. Diao, J. L. Hellerstein, S. Parekh, and J. P. Bigus, “Managing Web Server

Performance with AutoTune Agents", IBM Systems Journal, Volume 42, No.1, January 2003, Page 136 – 149.

[31] E. H. Durfee, "Coordination of Distributed Problem Solvers", Kluwer Academic Publishers, August 1988.

[32] G. B. Enguix and M. D. J. Lopez, "Agent-Environment Interaction in a Multi-Agent System: a Formal Model", Proceedings of the 2007 GECCO Conference on Genetic and Evolutionary Computation, July 2007, Page 2607 – 2612.

[33] T. Eymann, M. Reinickke, O. Ardaiz, P. Artigas, F. Freitag, L. Navarro, "Self-Organizing Resource Allocation for Autonomic Network", Proceedings of the 14th International Workshop on Database and Expert Systems Applications, September 2003, Page 656 – 660.

[34] M. Fasli, "Social Interactions in Multi-Agent Systems: A Formal Approach", Proceedings of the 2003 IEEE/WIC International Conference on Intelligent Agent Technology, October 2003, Page 240 – 246.

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

[36] J. Fiadeiro and T. Maibaum, "A Mathematical Toolbox for the Software Architect", Proceedings of the 8th International Workshop on Software Specification and Design, March 1996, Page 46 – 55.

[37] M. Fisher, "Representing and Executing Agent-Based Systems", Proceedings of the

Workshop on Agent Theories, Architectures, and Languages on Intelligent Agents, August 1995, Page 307 – 323.

[38] A. G. Ganek, T. A. Corbi, "The Dawning of the Autonomic Computing Era", IBM Systems Journal, Volume 42, No. 1, January 2003, Page 5 – 18.

[39] M. P. Georgeff, "Communication and Interaction in Multi-Agent Planning", Proceedings of the 3rd National Conference on Artificial Intelligence, August 1983, Page 125 – 129.

[40] R. Ginis, R. Strom, "An Automatic Messaging Middleware with Stateful Stream Transformation", Proceedings of the 1st International Conference on Autonomic Computing, May 2004, Page 316 – 317.

[41] J. A. Goguen, J. W. Thatcher, and E. G. Wagner, "An Initial Algebra Approach to the Specification, Correctness and Implementation of Abstract Data types", *Current Trends in Programming Methodology: Data Structuring*, Prentice-Hall, September 1978, Page 80 – 149.

[42] J. A. Goguen and R. M. Burstall, "Some Fundamental Algebraic Tools for the Semantics of Computation – Part 1: Comma Categories, Colimits, Signatures and Theories", Theoretical Computer Science, Volume 31, No. 2, July 1984, Page 175 – 209.

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

[44] B. Grosz and C. Sidner, "Collaborative Plans for Complex Group Actions", Artificial

Intelligence, Volume 86, No. 2, October 1996, Page 269 – 357.

[45] H. Guo, J, Gao, P. Zhu, and F. Zhang, "A Self-Organized Model of Agent-Enabling Autonomic Computing for Grid Environment", Proceedings of the 6th World Congress on Intelligent Control and Automation, June 2006, Page 2623 – 2627.

[46] J. Guo, "Using Category Theory to Model Software Component Dependencies", Proceedings of the 9th Annual IEEE International Conference and Workshop on the Engineering of Computer-Based Systems, April 2002, Page 185 – 192.

[47] R. Haas, P. Droz, and B. Stiller, "Autonomic service deployment in networks", IBM Systems Journal, Volume 42, No.1, January 2003, Page 150 – 164.

[48] T. Heinis, C. Pautasso, and G. Alonso, "A Self-Configuring Service Composition Engine", *Autonomic Computing: Concepts, Infrastructure, and Applications*, CRC Press, December 2006, Page 237 – 251.

[49] V. Hilaire, A. Koukam, P. Gruer, and J. P. Muller, "Formal Specification and Prototyping of Multi-Agent Systems", Proceedings of the 1st International Workshop on Engineering Societies in the Agent World, August 2000, Page 114 – 127.

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

[51] M. G. Hinchey, C. A. Rouff, J. L. Rash, and W. F. Truszkowski, "Requirements of an Integrated Formal Method for Intelligent Swarms", Proceedings of the 10th International Workshop on Formal Methods for Industrial Critical Systems, September 2005, Page 125

– 133.

[52] M. Hinchey, Y. Dai, J. L. Rash, W. Truszkowski, and M. Madhusoodan, “Bionic Autonomic Nervous System and Self-Healing for NASA ANTS-Like Missions”, Proceedings of the 2007 ACM Symposium on Applied Computing, March 2007, Page 90 – 96.

[53] M. Hinchey, Y. Dai, C. A. Rouff, J. L. Rash, and M. Qi, “Modeling for NASA Autonomous Nano-Technology Swarm Missions and Model-Driven Autonomic Computing”, Proceedings of the 21st International Conference on Advanced Networking and Applications, May 2007, Page 250 – 257.

[54] K.V. Hindriks, F. S. De Boer, W. V. Der Hoek, and J. Ch. Meyer, “Agent Programming in 3APL”, International Journal of Autonomous Agents and Multi-Agent Systems, Volume 2, Issue 4, November 1999, Page 357 – 401.

[55] W. M. L. Holcombe, “Towards a Formal Description of Intracellular Biochemical Organization”, Technical Report CS-86-1, Department of Computer Science, Sheffield University, Sheffield, South Yorkshire, United Kingdom, January 1986.

[56] W. M. L. Holcombe, “Mathematical Models of Cell Biochemistry”, Technical Report CS-86-4, Department of Computer Science, Sheffield University, Sheffield, South Yorkshire, United Kingdom, April 1986.

[57] W. M. L. Holcombe, “X-Machines as a Basis for Dynamic System Specification”, Software Engineering Journal, Volume 3, Issue 2, March1988, Page 69 – 76.

[58] P. Horn, “Autonomic Computing: IBM Perspective on the State of Information

Technology", Presented at AGENDA 2001, IBM T. J. Watson Labs, October 2001.

[59] J. Hu, J. Gao, B. Liao, and J. Chen, "Multi-Agent System Based Autonomic Computing Environment", Proceedings of the 3rd International Conference on Machine Learning and Cybernetics, August 2004, Page 105 – 110.

[60] J. Hou, J. Wan, and S. Wang, "Formalization of Architecture-Centric Model Mapping Using Category", Proceedings of the 8th ACIS International Conference on Software Engineering, Artificial Intelligence, Networking, and Parallel/Distributed Computing, July 2007, Page 670 – 675.

[61] J. Hu and M. P. Wellman, "Self-Fulfilling Bias in Multi Agent Learning", Proceedings of the 2nd International Conference on Multi-Agent Systems, December 1996, Page 118 – 125.

[62] J. Huang, "Modeling Multi-Agent Systems with Category Theory", Master's Thesis, Department of Computer Science and Software Engineering, Concordia University, Montreal, Canada, 2011.

[63] IBM Corporation, "An Architectural Blueprint for Autonomic Computing," White Paper, 4th Edition, June 2006.

[64] IBM Corporation, "Automating problem determination, a first step towards self-healing computing systems", White Paper, October 2003.

[65] IBM Corporation, "An architectural blueprint for autonomic computing", White Paper, 1st Edition, April 2003.

[66] IBM Corporation, "An architectural blueprint for autonomic computing", White

Paper, 2nd Edition, October 2004.

[67] IBM Corporation, "An architectural blueprint for autonomic computing", White Paper, 3rd Edition, June 2005.

[68] IEEE Standard 1061-1998, "IEEE Standard for a Software Quality Metrics Methodology", IEEE Computer Society, March 1998.

[69] I. Ilyas, V. Markl, P. Haas, P. G. Brown, A. Aboulnaga, "Automatic Relationship Discovery in Self-Managing Database Systems", Proceedings of the 1st International Conference on Autonomic Computing, May 2004, Page 340 – 341.

[70] ISO/IEC 14977:1996(E), "Information technology - Syntactic metalanguage - Extended BNF", ISO/IEC, 1996.

[71] V. Janarthanan and P. Sinha, "Modular Composition and Verification of Transaction Processing Protocols", Proceedings of the 23rd International Conference on Distributed Computing Systems, May 2003, Page 450 – 457.

[72] J. Jann, L. M. Browning, and R. S. Burgula, "Dynamic Reconfiguration: Basic building blocks for autonomic computing on IBM pSeries servers", IBM Systems Journal, Volume 42, No.1, January 2003, Page 29 – 37.

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

[74] N. R. Jennings, "Controlling Cooperative Problem Solving in Industrial Multi-Agent Systems Using Joint Intentions", Artificial Intelligence, Volume 75, Issue 2, June 1995,

Page 195 – 240.

[75] N. R. Jennings, “Specification and Implementation of a Belief Desire Joint-Intention Architecture for Collaborative Problem Solving”, Journal of Intelligent and Cooperative Information Systems, Volume 2, No. 3, June 1993, Page 289 – 318.

[76] N. R. Jennings, P. Faratin, M. J. Johnson, T. J. Norman, P. O’Brien, and M. E. Wiegand, “Agent-Based Business Process Management”, International Journal of Cooperative Information systems, Volume 5, No. 2 & 3, June & September 1996, Page 105 – 130.

[77] W. Jiao, M. Zhou, and Q. Wang, “Formal Framework for Adaptive Multi-Agent Systems”, Proceedings of the 2003 IEEE/WIC International Conference on Intelligent Agent Technology, October 2003, Page 442 – 445.

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

[79] S. E. Johnston, R. Sterritt, E. Hanna, and P. O’Hagan, “Reflex Autonomicity in an Agent-Based Security System: The Autonomic Access Control System”, Proceedings of the 4th IEEE International Workshop on Engineering of Autonomic and Autonomous Systems, March 2007, Page 68 – 78.

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

[81] N. Khurshid, “Towards Specifying Swarm-Based Systems using Categorical

Modeling Language: A Case Study", Master's Thesis, Department of Computer Science and Software Engineering, Concordia University, Montreal, Canada, 2011.

[82] J. R. Kiniry, "The Specification of Dynamic Distributed Component Systems", Master Thesis, Department of Computer Science, California Institute of Technology, Pasadena, California, United State of America, May 1998.

[83] S. Kraus, J. Wilkenfeld, and G. Zlotkin, "Multi Agent Negotiation under Time Constraints", Artificial Intelligence, Volume 75, No. 2, June 1995, Page 297 – 345.

[84] H. Kuang, "Architecture for Reactive Autonomic Systems: AS-TRM Approach", Master Thesis, Department of Computer Science & Software Engineering, Concordia University, Montreal, Quebec, Canada, April 2006.

[85] H.Kuang, "Architecture for Reactive Autonomic Systems: AS-TRM Approach", LAP LAMBERT Academic Publishing, 2010, ISBN: 978- 3838364124.

[86] H. Kuang, J. Bentahar, O. Ormandjieva, N. Shafieidizaji, S. Klasa, "Formal Specification of Substitutability Property for Fault-Tolerance in Reactive Autonomic Systems", *Frontiers in Artificial Intelligence and Applications*, Volume 217, pp. 357 – 380, IOS Press, 2010, DOI: 10.3233/978-1-60750-629-4-357.

[87] H. Kuang, O. Ormandjieva, S. Klasa, J. Bentahar, "A Formal Specification of Fault-Tolerance in Prospecting Asteroid Mission with Reactive Autonomic Systems Framework", *Proceedings of the Twenty-First IEEE International Conference on Application-specific Systems, Architectures and Processors*, pp. 99 – 106, Rennes, France, July 2010.

[88] H. Kuang, O. Ormandjieva, "Self-Monitoring of Non-Functional Requirements in Reactive Autonomic Systems Framework: A Multi-Agent Systems Approach", *Proceedings of the Third International Multi-Conference on Computing in the Global Information Technology*, pp. 186-192, Athens, Greece, July 2008.

[89] H. Kuang, "Reactive Autonomic Systems Framework: Multi-Agent Approach", Ph.D. Thesis Proposal, Department of Computer Science & Software Engineering, Concordia University, Montreal, Quebec, Canada, February 2009.

[90] H. Kuang, O. Ormandjieva, S. Klasa, N. Khurshid, and J. Bentahar, "Towards specifying reactive autonomic systems with a categorical approach: a case study", Studies in Computational Intelligence, Volume 253/2009, Springer Berlin/Heidelberg, November 2009, 119-134.

[91] A. Labella and A. Pettorossi, "Universal Models in Categories for Process Synchronization", Proceedings on Mathematical Models for the Semantics of Parallelism, September 1986, Page 183 – 198.

[92] G. Lafranchi, P. D. Peruta, A. Perrone, and D. Calvanese, "Toward a new landscape of systems management in an autonomic computing environment", IBM Systems Journal, Volume 42, No. 1, January 2003, Page 119 – 128.

[93] S. M. Lane, "Categories for the Working Mathematician", Springer–Verlag: New York, Heidelberg, Berlin, 1971.

[94] W. M. Lee, "Modeling and Specification of Autonomous Systems Using Category Theory", PhD Thesis, Department of Computer Science, University College London,

London, Greater London, United Kingdom, October 1989.

[95] S. Leriche and J. P. Arcangeli, "Flexible Architectures and Agents for Adaptive Autonomic Systems", Proceedings of the 4th IEEE International Workshop on Engineering of Autonomic and Autonomous Systems, March 2007, Page 99 – 106.

[96] V. R. Lesser, "A Retrospective View of FA/C Distributed Problem Solving", IEEE Transactions on Systems, Man, and Cybernetics, Volume 21, No. 6, December 1991, Page 1347 – 1362.

[97] H. J. Levesque, P. R. Cohen, and J. H. T. Nunes, "On Acting Together", Proceedings of the 8th National Conference on Artificial Intelligence, July 1990, Page 94 – 99.

[98] Z. Li and M. Parashar, "A Decentralized Agent Framework for Dynamic Composition and Coordination for Autonomic Applications", Proceedings of the 16th International Workshop on Database and Expert Systems Applications, August 2005, Page 165 – 169.

[99] Z. Li and M. Parashar, "Rudder: A Rule-Based Multi-Agent Infrastructure for Supporting Autonomic Grid Applications", Proceedings of the 1st International Conference on Autonomic Computing, May 2004, Page 278 – 279.

[100] S. Lightstone, B. Schiefer, D. Zilio, J. Kleewein, "Autonomic Computing for Relational Databases: the Ten-Year Vision", Proceedings of the IEEE International Conference on Industrial Informatics, August 2003, Page 419 – 424.

[101] P. Lin, A. MacArthur, J. Leaney, "Defining Autonomic Computing: A Software Engineering Perspective", Proceedings of the 2005 Australian Conference on Software

Engineering, March 2005, Page 88 – 97.

[102] H. Liu and M. Parashar, "A Programming System for Autonomic Self-Managing Applications", *Autonomic Computing: Concepts, Infrastructure, and Applications*, CRC Press, December 2006, Page 211 – 235.

[103] C. Loeser, M. Ditze, P. AltenBernd, F. Rammig, "GRUSEL – a Self-Optimizing Bandwidth Aware Video on Demand P2P Application", Proceedings of the 1st International Conference on Autonomic Computing, May 2004, Page 330 – 331.

[104] M. Luck and M. D'Inverno, "A Conceptual Framework for Agent Definition and Development", the Computer Journal, Volume 44, No. 1, November 2001, Page 1 – 20.

[105] M. Luck, P. McBurney, O. Shehory, S. Willmott, and AgentLink Community, *Agent Technology: Computing as Interaction, a Roadmap for Agent-Based Computing*, AgentLink III, September 2005.

[106] A. K. Mackworth and Y. Zhang, "A Formal Approach to Agent Design: an Overview of Constraint-Based Agents", Constraints, Volume 8, Issue 3, July 2003, Page 229 – 242.

[107] A. Madureira, J. Santos, and I. Pereira, "Self-Managing Agents for Dynamic Scheduling in Manufacturing", Proceedings of the 2008 GECCO Conference on Genetic and Evolutionary Computation, July 2008, Page 2187 – 2192.

[108] P. Maes, "Situated Agents Can Have Goals", *Designing Autonomous Agents: Theory and Practice from Biology to Engineering and Back*, MIT Press, February 1991, Page 49 – 70.

[109] E. Mainsah, “Autonomic Computing: the Next Era of Computing”, IEE Electronics Communication Engineering Journal, Volume 14, No. 1, February 2002, Page 2 – 3.

[110] V. Markl, G. M. Lohman, and V. Rahman, “LEO: An Autonomic Query Optimizer for DB2”, IBM Systems Journal, Volume 42, No.1, January 2003, Page 98 – 106.

[111] T. Marshall and Y. S. Dai, “Reliability Improvement and Models in Autonomic Computing”, Proceedings of the 11th International Conference on Parallel and Distributed Systems, July 2005, Page 468 – 472.

[112] P. McBurney, S. Parsons, and M. Wooldridge, “Desiderata for Agent Argumentation Protocols”, Proceedings of the 1st International Joint Conference on Autonomous Agents and Multi Agent Systems, July 2002, Page 402 – 409.

[113] P. McBurney and S. Parsons, “Games that Agents Play: a Formal Framework for Dialogues between Autonomous Agents”, Journal of Logic, Language, and Information, Volume 11, Issue 3, July 2002, Page 315 – 334.

[114] Microsoft Corporation, “Microsoft Dynamic Systems Initiative”, White Paper, October 2003.

[115] R. Milner, “A Theory of Type Polymorphism in Programming”, Journal of Computer and System Sciences, Volume 17, No. 3, December 1978, Page 348 – 375.

[116] P. Moraitis and N. Spanoudakis, “Argumentation-Based Agent Interaction in an Ambient-Intelligence Context”, IEEE Intelligent Systems, Volume 22, Issue 6, November 2007, Page 84 – 93.

[117] A. F. Moreira, R. Vieira, and R. H. Bordini, “Extending the Operational Semantics

of a BDI Agent-Oriented Programming Language for Introducing Speech-Act Based Communication", Proceedings of the 1st International Workshop on Declarative Agent Languages and Technologies, July 2003, Page 135 – 154.

[118] R. Murch, *Autonomic Computing*, IBM Press, April 2004.

[119] J. V. Neumann, *Theory of Self-Reproducing Automata*, University of Illinois Press, July 1966.

[120] A. Newell, *Unified Theories of Cognition*, Harvard University Press, April 1994.

[121] M. Parashar, "Autonomic Grid Computing: Concepts, Requirements, and Infrastructure", *Autonomic Computing: Concepts, Infrastructure, and Applications*, CRC Press, December 2006, Page 49 – 70.

[122] Y. A. Obaisat and R. Braun, "A Multi-Agent Flexible Architecture for Autonomic Services and Network Management", Proceedings of the 5th ACS/IEEE International Conference on Computer Systems and Applications, May 2007, Page 132 – 138.

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

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

[125] O. Ormandjieva, I. Hussain, “Towards Automatic Generation of Formal Scenarios Specifications from Real-Time Reactive Systems Requirements Written in NL”, Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications, Las Vegas, USA, June 2006, pp. 991 – 999.

[126] O. Ormandjieva, H. Kuang, S. Klasa, "Reactive Autonomic System Performance Modeling and Self-Monitoring with Category Theory", *Proceedings of the Fourth International Conference on Software and Data Technologies*, pp. 325-330, Sofia, Bulgaria, July 2009.

[127] W. H. Oyenan and S. A. DeLoach, “Design and Evaluation of a Multi Agent Autonomic Information System”, Proceedings of the 2007 IEEE/WIC/ACM International Conference on Intelligent Agent Technology, November 2007, Page 182 – 188.

[128] P. A. Patsouris, “Coordinated Flows in a Formal Multi-Agent System Based on a Modal Algebra”, Proceedings of the 1999 International Conference on Parallel Processing, September 1999, Page 480 – 487.

[129] L. Paulson, “Computer System Heal Thyself”, IEEE Computer, Volume 35, No. 8, August 2002, Page 20 – 22.

[130] J. Pfalzgraf, T. Soboll, “On a General Notion of Transformationfor Multiagent Systems”, Integrated Design and Process Technology, IDPT-2007 printed in the United States of America, June 2007.

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

[132] G. Pour, “Prospects for Expanding Telehealth: Multi-Agent Autonomic Architecture”, Proceedings of the International Conference on Computational Intelligence for Modeling Control and Automation, and International Conference on Intelligent Agent, Web Technologies and Internet Commerce, November 2006, Page 130 – 135.

[133] G. Qu, S. Hariri, S. Jangiti, J. Rudraraju, S. Oh, S. Fayssal, G. Zhang, M. Parashar, “Online Monitoring and Analysis for Self-Protection Against Network Attacks”, Proceedings of the 1st International Conference on Autonomic Computing, May 2004, Page 324 – 325.

[134] J. Quiroz, “Methodology for Automatic Generation of Behavioral Specification in Reactive Autonomic Systems", Master’s Thesis, Department of Computer Science and Software Engineering, Concordia University, Montreal, Canada, 2007.

[135] I. Rahwan, P. Moraitis, and C. Reed, *Argumentation in Multi-Agent Systems*, Proceedings of the 1st Workshop on Argumentation in Multi-Agent Systems, July 2004.

[136] A. Rao and M. Georgeff, “An Abstract Architecture for Rational Agents”, Proceedings of the 3rd International Conference on Principles of Knowledge Representation and Reasoning, October 1992, Page 439 – 449.

[137] A. Rao and M. Georgeff, “An Abstract Architecture for Rational Agents”, Proceedings of the 3rd International Conference on Principles of Knowledge Representation and Reasoning, October 1992, Page 439 – 449.

[138] A. Rao, “AgentSpeak(L): BDI Agents Speak Out in a Logical Computable

Language", Proceedings of the 7th European Workshop on Modelling Autonomous Agents in a Multi-Agent World, January 1996, Page 42 – 55.

[139] A. Regayeg, A. H. Kacem, and M. Jmaiel, "Towards a Formal Methodology for Developing Multi-Agent Applications Using Temporal Z", Proceedings of the 3rd ACS/IEEE International Conference on Computer Systems and Applications, January 2005, Page 123 – 130.

[140] C. Rich and C. Sidner, "COLLAGEN: When Agents Collaborate with People", Proceedings of the 1st International Conference on Autonomous Agents, February 1997, Page 284 – 291.

[141] J. S. Rosenschein, "Rational Interaction: Cooperation among Intelligent Agents", PhD Thesis, Department of Computer Science, Stanford University, Stanford, California, United States of America, January 1986.

[142] J. S. Rosenschein and G. Zlotkin, *Rules of Encounter: Designing Conventions for Automated Negotiation among Computers*, MIT Press, July 1994.

[143] L. W. Russel, S. P. Morgan, and E. G. Chron, "Clockwork: A new movement in autonomic systems", IBM System Journal, Volume 42, No.1, January 2003, Page 77 – 84.

[144] S. M. Sadjadi, P. K. McKinley, R. E. K. Stirewalt, B. H. C. Cheng, "Generation of Self-Optimizing Wireless Network Application", Proceedings of the 1st International Conference on Autonomic Computing, May 2004, Page 310 – 311.

[145] T. Sandholm and V. Lesser, "Issues in Automated Negotiation and Electronic Commerce: Extending the Contract Net Protocol", Proceedings of the 1st International

Conference on Multi Agent Systems, June 1995, Page 328 – 335.

[146] A. Sernadas, C. Sernadas, and C. Caleiro, "Synchronization of Logics", Journal of Studia Logica, Volume 59, No. 2, September 1997, Page 217 – 247.

[147] S. A. Seshia, "Autonomic Reactive Systems via Online Learning", Proceedings of the 4th International Conference on Autonomic Computing, June 2007, Page 30 – 39.

[148] N. Shafiei-Dizaji, "Multi-Agent Approach to Modeling and Implementing Fault-Tolerance in Reactive Autonomic Systems", Master's Thesis, Department of Computer Science and Software Engineering, Concordia University, Montreal, Canada, 2011.

[149] M. P. Singh and M. N. Huhns, *Service-Oriented Computing: Semantics, Processes, and Agents*, John Wiley & Sons, January 2005.

[150] G. Smith and J. Derrick, "Refinement and Verification of Concurrent Systems Specified in Object-Z and CSP", Proceedings of the 1st IEEE International Conference on Formal Engineering Methods, November 1997, Page 293 – 302.

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

[152] B. Solomon, D. Ionescu, M. Litoiu, and M. Mihaescu, "A Real-Time Adaptive Control of Autonomic Computing Environments", Proceedings of the Conference of the Centre for Advanced Studies on Collaborative Research, October 2007, Page 124 – 136.

[153] W. M. Spears and D. F. Gordon, “Using Artificial Physics to Control Agents”, Proceedings of the 1999 International Conference on Information Intelligence and Systems, October 1999, Page 281 – 288.

[154] P. Steenkiste and A. Huang, “Recipe-Based Service Configuration and Adaptation”, *Autonomic Computing: Concepts, Infrastructure, and Applications*, CRC Press, December 2006, Page 189 – 207.

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

[156] D. J. T. Sumpter, G. B. Blanchard, and D. S. Broomhead, “Ants and Agents: a Process Algebra Approach to Modeling Ant Colony Behaviour”, Bulletin of Mathematical Biology, Volume 63, No. 5, September 2001, Page 951 – 980.

[157] Sun Microsystems, “Sun Cluster Grid Architecture”, White Paper, May 2002.

[158] Sun Microsystems, “ARCO, N1 Grid Engine 6 Accounting and Reporting Console”, White Paper, May 2005.

[159] K. P. Sycara, “Multi Agent Systems”, AI Magazine, Volume 19, No. 2, July 1998, Page 79 – 92.

[160] K. P. Sycara, “Resolving Goal Conflicts via Negotiation”, Proceedings of the 7th National Conference on Artificial Intelligence, August 1988, Page 245 – 250.

[161] K. P. Sycara, “Persuasive Argumentation in Negotiation”, Journal of Theory and Decision, Volume 28, No. 3, May 1990, Page 203 – 242.

[162] K. P. Sycara, “Negotiation Planning: an AI Approach”, European Journal of Operational Research, Volume 46, No. 2, May 1990, Page 216 – 234.

[163] Y. Takahara and T. Takai, “Category Theoretical Framework of General Systems”, International Journal of General Systems, Volume 11, No. 1, February 1985, Page 1 – 33.

[164] Y. Takahara and T. Takai, “The Category Theory of Time System”, International Journal of General Systems, Volume 12, No. 1, February 1986, Page 71 – 105.

[165] M. Tambe, “Towards Flexible Teamwork”, Journal of Artificial Intelligence Research, Volume 7, July 1997, Page 83 – 124.

[166] R. Telford, R. Horman, S. Lightstone, N. Markov, S. O. Connell, and G. Lohman, “Usability and design consideration for an autonomic relational database management system”, IBM Systems Journal, Volume 42, No.4, 2003, Page 568 – 581.

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

[168] S. N. Thorsen and M. E. Oxley, “Describing Data Fusion Using Category”, Proceedings of the 6th International Conference on Information Fusion, July 2003, Page 1202 – 1206.

[169] H. Tianfield, “Multi-Agent Autonomic Architecture and Its Application in E-Medicine”, Proceedings of the IEEE/WIC International Conference on Intelligent Agent Technology, October 2003, Page 601 – 604.

[170] C. Tofts, “Describing Social Insect Behaviour Using Process Algebra”, Transaction on Social Computing Simulation, Volume 10, No. 1, December 1992, Page 227 – 283.

[171] I. Trencansky, R. Cervenka, and D. Greenwood, “Applying a UML-Based Agent Modeling Language to the Autonomic Computing Domain”, Proceedings of the ACM SIGPLAN International Conference on Object-Oriented Programming Systems, Language, and Applications, October 2006, Page 521 – 529.

[172] W. Trumler, J. Petzold, F. Bagci, T. Ungerer, “AMUN – Autonomic Middleware for Ubiquitous Environments Applied to the Smart Doorplate Project”, Proceedings of the 1st International Conference on Autonomic Computing, May 2004, Page 274 – 275.

[173] W. Truszkowski, J. Rash, C. Rouff, and M. Hinchey, “Some Autonomic Properties of Two Legacy Multi-Agent Systems – LOGOS and ACT”, Proceedings of the 11th IEEE International Conference and Workshop on the Engineering of Computer-Based Systems, May 2004, Page 490 – 498.

[174] W. F. Truszkowski, M. G. Hinchey, J. L. Rash, and C. A. Rouff, “Autonomous and Autonomic Systems: a Paradigm for Future Space Exploration Missions”, IEEE Transaction on Systems, Man, and Cybernetics, Part C: Applications and Reviews, Volume 36, Issue 3, May 2006, Page 279 – 291.

[175] E. Vassev, M. Hinchey, and J. Paquet, “A Self-Scheduling Model for NASA Swarm-Based Exploration Missions Using ASSL”, Proceedings of the 5th IEEE Workshop on Engineering of Autonomic and Autonomous Systems, March 2008, Page 54 – 64.

[176] E. Vassev, H. Kuang, O. Ormandjieva, J. Paquet, "Reactive, Distributed and Autonomic Computing Aspects of AS-TRM", *Proceedings of the First International Conference on Software and Data Technologies*, pp. 196-202, Setubal, Portugal, September 2006.

[177] E. Vassev, Q. T. D. Nguyen, H. Kuang, "Fault-Tolerance through Message-logging and Check-pointing: Disaster Recovery for CORBA –based Distributed Bank Servers", CoRR abs/0911.3092, 2009.

[178] D. C. Verma, S. Sahu, S. Calo, A. Shaikh, I. Chang, and A. Acharya, "SRIRAM: A scalable resilient autonomic mesh", IBM Systems Journal, Volume 42, No.1, January 2003, Page 19 – 28.

[179] C. Vermeulen and B. Bauwens, "Software Agents Using XML for Telecom Service Modeling: a Practical Experience", Proceedings of the SGML/XML Europe'98, May 1998, Page 253 – 262.

[180] D. N. Walton and E. C. W. Krabble, *Commitment in Dialogue: Basic Concepts of Interpersonal Reasoning*, SUNY Press, July 1995.

[181] W. Wan, "Specifying and Verifying Communities of Web Services Using Argumentative Agents", Master Thesis, Concordia Institute for Information Systems Engineering, Concordia University, Montreal, Quebec, Canada, August 2008.

[182] D. B. Weaver, M. J. Healy, and T. P. Caudell, "An Application of Category-Theoretic Design Methods to the Control of a Simulated Robot", Proceedings of the International Joint Conference on Neural Networks, August 2007, Page 2058 –

2063.

[183] M. Weiser, “Creating the Invisible Interface”, Proceedings of the 7th Annual ACM Symposium on User Interface Software and Technology, 1994, Page 1 – 2.

[184] K. Whinsnant, Z. T. Kalbarczyk, and R. K. Tyer, “A System Model for Dynamically Reconfigurable Software”, IBM Systems Journal, Volume 42, No. 1, January 2003, Page 45 – 59.

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

[186] G. Winskel, “Categories of Models for Concurrency”, Lecture Notes in Computer Science, Volume 197, July 1984, Page 246 – 267.

[187] G. Winskel, “Petri Nets, Algebras and Morphisms”, Technical Report, No. 79, Computer Laboratory, University of Cambridge, Cambridge, Cambridgeshire, United Kingdom, September 1985.

[188] G. Winskel, “Category Theory and Models for Parallel Computation”, Technical Report, No. 85, Computer Laboratory, University of Cambridge, Cambridge, Cambridgeshire, United Kingdom, April 1986.

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

[190] T. D. Wolf and T. Holvoet, “Towards Autonomic Computing: Agent-Based Modeling, Dynamical Systems Analysis, and Decentralised Control”, Proceedings of the

1st International Workshop on Autonomic Computing Principles and Architectures, August 2003, Page 470 – 479.

[191] M. Wooldridge, *An Introduction to Multi Agent Systems*, John Wiley & Sons, June 2002.

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

[193] M. Wooldridge, *Reasoning about Rational Agents*, MIT Press, July 2000.

[194] M. Wooldridge and N. Jennings, "The Cooperative Problem-Solving Process", Journal of Logic and Computation, Volume 9, No. 4, August 1999, Page 563 – 592.

[195] D. Xu, R. Volz, and T. Ioerger, "Modeling and Verifying Multi-Agent Behaviors Using Predicate/Transition Nets", Proceedings of the 14th International Conference on Software Engineering and Knowledge Engineering, July 2002, Page 193 – 200.

[196] H. Xu and S. M. Shatz, "ADK: an Agent Development Kit Based on a Formal Design Model for Multi-Agent Systems", Automated Software Engineering, Volume 10, Issue 4, October 2003, Page 337 – 365.

[197] D. M. Yellin, "Competitive Algorithms for the Dynamic Selection of Component Implementation", IBM Systems Journal, Volume 42, No.1, January 2003, Page 85 – 97.

[198] D. Zeng and K. P. Sycara, "Benefits of Learning in Negotiation", Proceedings of the 14th National Conference on Artificial Intelligence, July 1997, Page 36 – 41.

[199] D. Zeng and K. P. Sycara, "Bayesian Learning in Negotiation", International

Journal of Human-Computer Studies, Volume 48, Issue 1, January 1998, Page 125 – 141.

[200] H. Zhu, "SLABS: a Formal Specification Language for Agent-Based Systems", International Journal of Software Engineering and Knowledge Engineering, Volume 11, No.5, November 2001, Page 529 – 558.

[201] http://jadex-agents.informatik.uni-hamburg.de/xwiki/bin/view/Usages/Examples, Last viewed on 2013.01.15.

[202] http://www.eclipse.org/.

[203] http://www.sparxsystems.com/products/ea/index.html.

[204] http://jadex.informatik.uni-hamburg.de/xwiki/bin/view/About/Overview.

Appendix A: Representation for Categorical Model of Constructors

```
<CATEGORY name = "RAE-Type-Instance">
    <OBJECT>
        <OBJECT name = "RAE_i" type = "RAE-Type_i" />
        <OBJECT name = "RAE_j" type = "RAE-Type_j"/>
    </OBJECT>
    <MORPHISM>
        <MORPHISM name = "Communication_n" type ="Communication-Type_n"/>
            <FROM-OBJECT name = "RAE_i" type = "RAE-Type_i"/>
            <TO-OBJECT name = "RAE_j" type = "RAE-Type_j"/>
        </MORPHISM>
    </MORPHISM>
</CATEGORY>
```

Figure 227: XML Specification of Category **RAE-Type-Instance**

```
<CATEGORY name = "RAC">
    <OBJECT>
        <OBJECT name = "RAO_i" type = "RAO-Type_i"/>
        <OBJECT name = "RAO_j" type = "RAO-Type_j"/>
    </OBJECT>
    <MORPHISM>
        <MORPHISM name = "Communication_n" type ="Communication-Type_n"/>
            <FROM-OBJECT name = "RAO_i" type = "RAO-Type_i"/>
            <TO-OBJECT name = "RAO_j" type = "RAO-Type_j"/>
        </MORPHISM>
    </MORPHISM>
</CATEGORY>
```

Figure 228: XML Specification of Category **RAC**

```
<FUNCTOR name = "RAC-Evolution" source-category = "RAC'"
                target-category = "RAC">
    <OBJECT-MAPPING>
        <OBJECT-MAPPING source-object = "RAO_j" target-object = "RAO_i"/>
        <OBJECT-MAPPING source-object = "RAO_k" target-object = "RAO_i"/>
    </OBJECT-MAPPING>
    <MORPHISM-MAPPING>
        <MORPHSIM-MAPPING source-morphism = "Communication_j"
                                target-morphism = "Communication_i"/>
        <MORPHSIM-MAPPING source-morphism = "Communication_k"
```

```
                                        target-morphism = "Communication_i"/>
    </MORPHISM-MAPPING>
</FUNCTOR>
```

Figure 229: XML Specification of Functor *RAC-Evolution*

```
<NATURAL-TRANSFORMATION name = "Relation-of-RAE-Evolution">
    <ARROW>
        <ARROW name = "Relation_n"/>
            <FROM-FUNCTOR name = "RAE-Evolution_i"
                                    type = "RAE-Evolution-Type_i"/>
            <TO-FUNCTOR name = "RAE-Evolution_j"
                                type = "RAE-Evolution-Type_i"/>
        </ARROW>
    </ARROW>
</NATURAL-TRANSFORMATION>
```

Figure 230: XML Specification of Natural Transformation *Relation*

```
<CATEGORY name = "Relation-Set-of-RAE-Evolution">
    <OBJECT>
        <OBJECT name = "RAE-Evolution_i" type = "RAE-Evolution-Type_i" />
        <OBJECT name = "RAE-Evolution_j" type = "RAE-Evolution-Type_j"/>
    </OBJECT>
    <MORPHISM>
        <MORPHISM name = "Relation_n" type ="Relation-Type_n"/>
            <FROM-OBJECT name = "RAE-Evolution_i"
                                type = "RAE-Evolution-Type_i" />
            <TO-OBJECT name = "RAE-Evolution_j"
                            type = "RAE-Evolution-Type_j"/>
        </MORPHISM>
    </MORPHISM>
</CATEGORY>
```

Figure 231: XML Specification of Functor Category **Relation-Set**

```
<CATEGORY name = "RACG">
    <OBJECT>
        <OBJECT name = "RAC_i" type = "RAC-Type_i" />
        <OBJECT name = "RAC_j" type = "RAC-Type_j"/>
    </OBJECT>
    <MORPHISM>
        <MORPHISM name = "Communication_n" type ="Communication-Type_n"/>
```

```
            <FROM-OBJECT name = "RAC_i" type = "RAC-Type_i"/>
            <TO-OBJECT name = "RAC_j" type = "RAC-Type_j"/>
        </MORPHISM>
    </MORPHISM>
</CATEGORY>
```

Figure 232: XML Specification of Category **RACG**

```
<FUNCTOR name = "RACG-Evolution" source-category = "RACG'"
            target-category = "RACG">
    <OBJECT-MAPPING>
        <OBJECT-MAPPING source-object = "RAC_j" target-object = "RAC_i"/>
        <OBJECT-MAPPING source-object = "RAC_k" target-object = "RAC_i"/>
    </OBJECT-MAPPING>
    <MORPHISM-MAPPING>
        <MORPHSIM-MAPPING source-morphism = "Communication_j"
                                target-morphism = "Communication_i"/>
        <MORPHSIM-MAPPING source-morphism = "Communication_k"
                                target-morphism = "Communication_i"/>
    </MORPHISM-MAPPING>
</FUNCTOR>
```

Figure 233: XML Specification of Functor *RACG-Evolution*

```
<CATEGORY name = "RAS">
    <OBJECT>
        <OBJECT name = "RACG_i" type = "RACG-Type_i" />
        <OBJECT name = "RACG_j" type = "RACG-Type_j"/>
    </OBJECT>
    <MORPHISM>
        <MORPHISM name = "Communication_n" type ="Communication-Type_n"/>
            <FROM-OBJECT name = "RACG_i" type = "RACG-Type_i"/>
            <TO-OBJECT name = "RACG_j" type = "RACG-Type_j"/>
        </MORPHISM>
    </MORPHISM>
</CATEGORY>
```

Figure 234: XML Specification of Category **RAS**

```
<FUNCTOR name = "RAS-Evolution" source-category = "RAS'"
            target-category = "RAS">
    <OBJECT-MAPPING>
        <OBJECT-MAPPING source-object = "RACG_j" target-object = "RACG_i"/>
```

```
        <OBJECT-MAPPING source-object = "RACG_k" target-object = "RACG_i"/>
    </OBJECT-MAPPING>
    <MORPHISM-MAPPING>
        <MORPHSIM-MAPPING source-morphism = "Communication_j"
                          target-morphism = "Communication_i"/>
        <MORPHSIM-MAPPING source-morphism = "Communication_k"
                          target-morphism = "Communication_i"/>
    </MORPHISM-MAPPING>
</FUNCTOR>
```

Figure 235: XML Specification of Functor *RAS-Evolution*

Appendix B: Representation for Categorical Model of Behavior

```
<CATEGORY name = "RAE-Behavior">
    <OBJECT>
        <OBJECT name = "Function-Pair_i" type = "Function-Pair-Type_i"/>
        <OBJECT name = "Function-Pair_j" type = "Function-Pair-Type_j"/>
    </OBJECT>
    <MORPHISM>
        <MORPHISM name = "Interaction_n" type ="Interaction-Type_n"/>
            <FROM-OBJECT name = "Function-Pair_i"
                                type = "Function-Pair-Type_i"/>
            <TO-OBJECT name = "Function-Pair_j" type = "Function-Pair-Type_j"/>
        <MORPHISM>
    </MORPHISM>
</CATEGORY>
```

Figure 236: XML Specification of Category **RAE-Behavior**

```
<CATEGORY name = "Discrete-Time">
    <OBJECT>
        <OBJECT name = "Abstract-Time-Unit_i" type = "Integer"/>
        <OBJECT name = "Abstract-Time-Unit_j" type = "Integer"/>
    </OBJECT>
    <MORPHISM>
        <MORPHISM name = "Before" type ="Preorder"/>
            <FROM-OBJECT name = "Abstract-Time-Unit_i" type = "Integer"/>
            <TO-OBJECT name = "Abstract-Time-Unit_j" type = "Integer"/>
        <MORPHISM>
    </MORPHISM>
</CATEGORY>
```

Figure 237: XML Specification of Category **Discrete-Time**

```
<CATEGORY name = "State-Type">
    <OBJECT>
        <OBJECT name = "State-Type_i"/>
        <OBJECT name = "State-Type_j"/>
    </OBJECT>
    <MORPHISM>
        <MORPHISM name = "Transition-Type_n"/>
            <FROM-OBJECT name = "State-Type_i"/>
            <TO-OBJECT name = "State-Type_j"/>
```

```
        <MORPHISM>
    </MORPHISM>
</CATEGORY>
```

Figure 238: XML Specification of Index Category **State-Type**

```
<CATEGORY name = "STATE">
    <OBJECT>
        <OBJECT name = "State_i" type = "State-Type_i"/>
        <OBJECT name = "State_j" type = "State-Type_j"/>
    </OBJECT>
    <MORPHISM>
        <MORPHISM name = "Transition_n" type ="Transition-Type_n"/>
            <FROM-OBJECT name = "State_i" type = "State-Type_i"/>
            <TO-OBJECT name = "State_j" type = "State-Type_j"/>
        <MORPHISM>
    </MORPHISM>
</CATEGORY>
```

Figure 239: XML Specification of Category **STATE**

```
<FUNCTOR name = "Time-Constraint" source-category = "STATE"
            target-category = "Discrete-Time'">
    <OBJECT-MAPPING>
        <OBJECT-MAPPING source-object = "State_i"
                                target-object = "Abstract-Time-Unit_i"/>
        <OBJECT-MAPPING source-object = "State_j"
                                target-object = "Abstract-Time-Unit_j"/>
    </OBJECT-MAPPING>
    <MORPHISM-MAPPING>
        <MORPHSIM-MAPPING source-morphism = "Transition_n"
                                    target-morphism = "Before"/>
        <MORPHSIM-MAPPING source-morphism = "Transition_n"
                                    target-morphism = "Before"/>
    </MORPHISM-MAPPING>
</FUNCTOR>
```

Figure 240: XML Specification of Functor *Time-Constraint-for-State*

```
<PRODUCT name = "Synchronous-Communication-between-RAE">
    <PRODUCT-OBJECT name = "Synchronous-Communication_n"
                            type = "Synchronous-Communication-Type_n"/>
    <BETWEEN-OBJECT name = "RAE_i" type = "RAE-Type_i"/>
```

```
    <BETWEEN-OBJECT name = "RAE_j" type = "RAE-Type_j"/>
</PRODUCT>
```

Figure 241: XML Specification of Product *Synchronous Communication of RAE*

```
<COPRODUCT name = "Asynchronous-Communication-between-RAE">
    <COPRODUCT-OBJECT name = "Asynchronous-Communication_n"
                                type = "Asynchronous-Communication-Type_n"/>
    <BETWEEN-OBJECT name = "RAE_i" type = "RAE-Type_i"/>
    <BETWEEN-OBJECT name = "RAE_j" type = "RAE-Type_j"/>
</COPRODUCT>
```

Figure 242: XML Specification of Coproduct *Asynchronous Communication of RAE*

```
<PUSHOUT name = "Next-Communication-Relay-of-RAE">
    <SOURCE-OBJECT name = "RAE_n" type = "RAE-Type_n"/>
    <RELAY-OBJECT name = "RAE_i" type = "RAE-Type_i"/>
    <RELAY-OBJECT name = "RAE_j" type = "RAE-Type_j"/>
    <DESTINATION-OBJECT name = "RAE_pushout" type = "RAE-Type_pushout"/>
</PUSHOUT>
```

Figure 243: XML Specification of Pushout *Next Communication Relay of RAE*

```
<PULLBACK name = "Previous-Communication-Relay-of-RAE">
    <SOURCE-OBJECT name = "RAE_pullback" type = "RAE-Type_pullback"/>
    <RELAY-OBJECT name = "RAE_i" type = "RAE-Type_i"/>
    <RELAY-OBJECT name = "RAE_j" type = "RAE-Type_j"/>
    <DESTINATION-OBJECT name = "RAE_n" type = "RAE-Type_n"/>
</PULLBACK>
```

Figure 244: XML Specification of Pullback *Previous Communication Relay of RAE*

```
<CATEGORY name = "RAE-Behavior-Designated">
    <OBJECT>
        <OBJECT name = "Cone-to-Diagram_i" type = "Cone-to-Diagram-Type_i"/>
        <OBJECT name = "Cone-to-Diagram_j" type = "Cone-to-Diagram-Type_j"/>
    </OBJECT>
     <MORPHISM>
        <MORPHISM name = "Incoming-Communication_n"
                        type ="Incoming-Communication-Type_n"/>
            <FROM-OBJECT name = "Cone-to-Diagram_i"
                                type = "Cone-to-Diagram-Type_i"/>
            <TO-OBJECT name = "Cone-to-Diagram_j"
                            type = "Cone-to-Diagram-Type_j"/>
```

```
        <MORPHISM>
    </MORPHISM>
</CATEGORY>
```

Figure 245: XML Specification of Category **RAE-Behavior-Designated**

```
<LIMIT name = "RAC-Behavior-Designated-Limit">
    <DIAGRAM name = "Construct_n" source-category = "RAO-Type"
                destination-category = "RAC_n"/>
    <BEHAVIOR-CATEGORY name = "RAC_n-Behavior-Designated"/>
    <TERMINAL-OBJECT name = "RAOL_n" type = "RAOL-Type_n"/>
</LIMIT>
```

Figure 246: XML Specification of Limit *RAC-Behavior-Designated-Limit*

```
<CATEGORY name = "RAE-Behavior-Achieved">
    <OBJECT>
        <OBJECT name = "Cocone-to-Diagram_i"
                type = "Cocone-to-Diagram-Type_i"/>
        <OBJECT name = "Cocone-to-Diagram_j"
                type = "Cocone-to-Diagram-Type_j"/>
    </OBJECT>
    <MORPHISM>
        <MORPHISM name = "Outgoing-Communication_n"
                type ="Outgoing-Communication-Type_n"/>
            <FROM-OBJECT name = "Cocone-to-Diagram_i"
                        type = "Cocone-to-Diagram-Type_i"/>
            <TO-OBJECT name = "Cocone-to-Diagram_j"
                        type = "Cocone-to-Diagram-Type_j"/>
        <MORPHISM>
    </MORPHISM>
</CATEGORY>
```

Figure 247: XML Specification of Category **RAE-Behavior-Achieved**

```
<COLIMIT name = "RAC-Behavior-Achieved-Colimit">
    <DIAGRAM name = "Construct_n" source-category = "RAO-Type"
                destination-category = "RAC_n"/>
    <BEHAVIOR-CATEGORY name = "RAC_n-Behavior-Achieved"/>
    <INITIAL-OBJECT name = "RAOL_n" type = "RAOL-Type_n"/>
</COLIMIT>
```

Figure 248: XML Specification of Colimit *RAC-Behavior-Achieved-Colimit*

```
<SLICE-CATEGORY name = "RAC/RAOL">
    <OBJECT>
        <OBJECT name = "Outgoing-Communication_i"
                    type = "Outgoing-Communication-Type_i"/>
        <OBJECT name = "Outgoing-Communication_j"
                    type = "Outgoing-Communication-Type_j"/>
    </OBJECT>
     <MORPHISM>
        <MORPHISM name = "Connection_n" type ="Connection-Type_n"/>
            <FROM-OBJECT name = "Outgoing-Communication_i"
                                type = "Outgoing-Communication-Type_i"/>
            <TO-OBJECT name = "Outgoing-Communication_j"
                            type = "Outgoing-Communication-Type_j"/>
        <MORPHISM>
     </MORPHISM>
</SLICE-CATEGORY>
```

Figure 249: XML Specification of Slice Category **RAC**/*RAOL*

```
<COSLICE-CATEGORY name = "RAOL/RAC">
    <OBJECT>
        <OBJECT name = "Incoming-Communication_i"
                    type = "Incoming-Communication-Type_i"/>
        <OBJECT name = "Incoming-Communication_j"
                    type = "Incoming-Communication-Type_j"/>
    </OBJECT>
     <MORPHISM>
        <MORPHISM name = "Connection_n" type ="Connection-Type_n"/>
            <FROM-OBJECT name = "Incoming-Communication_i"
                                type = "Incoming-Communication-Type_i"/>
            <TO-OBJECT name = "Incoming-Communication_j"
                            type = "Incoming-Communication-Type_j"/>
        <MORPHISM>
     </MORPHISM>
</CATEGORY>
```

Figure 250: XML Specification of Coslice Category *RAOL*/**RAC**

```
<LIMIT name = "RACG-Behavior-Designated-Limit">
    <DIAGRAM name = "Construct_n" source-category = "RAC-Type"
                destination-category = "RACG_n"/>
    <BEHAVIOR-CATEGORY name = "RACG_n-Behavior-Designated"/>
```

```
    <TERMINAL-OBJECT name = "RACS_n" type = "RACS-Type_n"/>
</LIMIT>
```

Figure 251: XML Specification of Limit *RACG-Behavior-Designated-Limit*

```
<COLIMIT name = "RACG-Behavior-Achieved-Colimit">
    <DIAGRAM name = "Construct_n" source-category = "RAC-Type"
                destination-category = "RACG_n"/>
    <BEHAVIOR-CATEGORY name = "RACG_n-Behavior-Achieved"/>
    <INITIAL-OBJECT name = "RACS_n" type = "RACS-Type_n"/>
</COLIMIT>
```

Figure 252: XML Specification of Colimit *RACG-Behavior-Achieved-Colimit*

```
<LIMIT name = "RAS-Behavior-Designated-Limit">
    <DIAGRAM name = "Construct_n" source-category = "RACG-Type"
                destination-category = "RAS_n"/>
    <BEHAVIOR-CATEGORY name = "RAS_n-Behavior-Designated"/>
    <TERMINAL-OBJECT name = "RACGM_n" type = "RACGM-Type_n"/>
</LIMIT>
```

Figure 253: XML Specification of Limit *RAS-Behavior-Designated-Limit*

```
<COLIMIT name = "RAS-Behavior-Achieved-Colimit">
    <DIAGRAM name = "Construct_n" source-category = "RACG-Type"
                destination-category = "RAS_n"/>
    <BEHAVIOR-CATEGORY name = "RAS_n-Behavior-Achieved"/>
    <INITIAL-OBJECT name = "RACGM_n" type = "RACGM-Type_n"/>
</COLIMIT>
```

Figure 254: XML Specification of Colimit *RAS-Behavior-Achieved-Colimit*

```
<CATEGORY name = "Transition-Type">
    <OBJECT>
        <OBJECT name = "Transition-Type_i"/>
        <OBJECT name = "Transition-Type_j"/>
    </OBJECT>
     <MORPHISM>
        <MORPHISM name = "Before"/>
            <FROM-OBJECT name = "Transition-Type_i"/>
            <TO-OBJECT name = "Transition-Type_j"/>
        <MORPHISM>
     </MORPHISM>
```

```
</CATEGORY>
```

Figure 255: XML Specification of Index Category **Transition-Type**

```
<CATEGORY name = "Transition">
    <OBJECT>
        <OBJECT name = "Transition_i" type = "Transition-Type_i"/>
        <OBJECT name = "Transition_j" type = "Transition-Type_j"/>
    </OBJECT>
    <MORPHISM>
        <MORPHISM name = "Before" type ="Preorder"/>
            <FROM-OBJECT name = "Transition_i" type = "Transition-Type_i"/>
            <TO-OBJECT name = "Transition_j" type = "Transition-Type_j"/>
        <MORPHISM>
    </MORPHISM>
</CATEGORY>
```

Figure 256: XML Specification of Category **Transition**

```
<FUNCTOR name = "Time-Constraint" source-category = "Transition"
            target-category = "Discrete-Time'">
    <OBJECT-MAPPING>
        <OBJECT-MAPPING source-object = "Transition_i"
                                target-object = "Abstract-Time-Unit_i"/>
        <OBJECT-MAPPING source-object = "Transition_j"
                                target-object = "Abstract-Time-Unit_j"/>
    </OBJECT-MAPPING>
    <MORPHISM-MAPPING>
        <MORPHSIM-MAPPING source-morphism = "Before"
                                    target-morphism = "Before"/>
        <MORPHSIM-MAPPING source-morphism = "Before"
                                    target-morphism = "Before"/>
    </MORPHISM-MAPPING>
</FUNCTOR>
```

Figure 257: XML Specification of Functor *Time-Constraint-for-Transition*

```
<CATEGORY name = "TRANSITION">
    <OBJECT>
        <OBJECT name = "Sequence_i" type = "Transition-Sequence"/>
        <OBJECT name = "Sequence_j" type = "Transition-Sequence"/>
    </OBJECT>
    <MORPHISM>
```

```
        <MORPHISM name = "Equivalent" type ="Preorder"/>
            <FROM-OBJECT name = "Sequence_i" type = "Transition-Sequence"/>
            <TO-OBJECT name = "Sequence_j" type = "Transition-Sequence"/>
        <MORPHISM>
    </MORPHISM>
</CATEGORY>
```

Figure 258: XML Specification of Category **TRANSITION**

```
<CATEGORY name = "Action-Type">
    <OBJECT>
        <OBJECT name = "Action-Type_i"/>
        <OBJECT name = "Action-Type_j"/>
    </OBJECT>
    <MORPHISM>
        <MORPHISM name = "Before"/>
            <FROM-OBJECT name = "Action-Type_i"/>
            <TO-OBJECT name = "Action-Type_j"/>
        <MORPHISM>
    </MORPHISM>
</CATEGORY>
```

Figure 259: XML Specification of Index Category **Action-Type**

```
<CATEGORY name = "Action">
    <OBJECT>
        <OBJECT name = "Action_i" type = "Action-Type_i"/>
        <OBJECT name = "Action_j" type = "Action-Type_j"/>
    </OBJECT>
    <MORPHISM>
        <MORPHISM name = "Before" type ="Preorder"/>
            <FROM-OBJECT name = "Action_i" type = "Action-Type_i"/>
            <TO-OBJECT name = "Action_j" type = "Action-Type_j"/>
        <MORPHISM>
    </MORPHISM>
</CATEGORY>
```

Figure 260: XML Specification of Category **Action**

```
<FUNCTOR name = "Time-Constraint" source-category = "Action"
            target-category = "Discrete-Time'">
    <OBJECT-MAPPING>
        <OBJECT-MAPPING source-object = "Action_i"
```

```
                                target-object = "Abstract-Time-Unit_i"/>
        <OBJECT-MAPPING source-object = "Action_j"
                                target-object = "Abstract-Time-Unit_j"/>
    </OBJECT-MAPPING>
    <MORPHISM-MAPPING>
        <MORPHSIM-MAPPING source-morphism = "Before"
                                target-morphism = "Before"/>
        <MORPHSIM-MAPPING source-morphism = "Before"
                                target-morphism = "Before"/>
    </MORPHISM-MAPPING>
</FUNCTOR>
```

Figure 261: XML Specification of Functor *Time-Constraint-for-Action*

```
<CATEGORY name = "INTERACTION">
    <OBJECT>
        <OBJECT name = "Sequence_i" type = "Action-Sequence"/>
        <OBJECT name = "Sequence_j" type = "Action-Sequence"/>
    </OBJECT>
    <MORPHISM>
        <MORPHISM name = "Equivalent" type ="Preorder"/>
            <FROM-OBJECT name = "Sequence_i" type = "Action-Sequence"/>
            <TO-OBJECT name = "Sequence_j" type = "Action-Sequence"/>
        <MORPHISM>
    </MORPHISM>
</CATEGORY>
```

Figure 262: XML Specification of Category **INTERACTION**

```
<CATEGORY name = "RAE-Social-Life">
    <OBJECT>
        <OBJECT name = "RAE_i" type = "RAE-Type_i"/>
        <OBJECT name = "RAE_j" type = "RAE-Type_j"/>
    </OBJECT>
    <MORPHISM>
        <MORPHISM name = "Social Connection_m"
                        type ="Social-Connection-Type_m"/>
            <FROM-OBJECT name = "RAE_i" type = "RAE-Type_i"/>
            <TO-OBJECT name = "RAE_j" type = "RAE-Type_j"/>
        <MORPHISM>
    </MORPHISM>
```

```
</CATEGORY>
```

Figure 263: XML Specification of Category **RAE-Social-Life**

```
<CATEGORY name = "Evolution-Type">
    <OBJECT>
        <OBJECT name = "Evolution-Type_i"/>
        <OBJECT name = "Evolution-Type_j"/>
    </OBJECT>
    <MORPHISM>
        <MORPHISM name = "Before"/>
            <FROM-OBJECT name = "Evolution-Type_i"/>
            <TO-OBJECT name = "Evolution-Type_j"/>
        <MORPHISM>
    </MORPHISM>
</CATEGORY>
```

Figure 264: XML Specification of Index Category **Evolution-Type**

```
<CATEGORY name = "Evolution">
    <OBJECT>
        <OBJECT name = "Evolution_i" type = "Evolution-Type_i"/>
        <OBJECT name = "Evolution_j" type = "Evolution-Type_j"/>
    </OBJECT>
    <MORPHISM>
        <MORPHISM name = "Before" type ="Preorder"/>
            <FROM-OBJECT name = "Evolution_i" type = "Evolution-Type_i"/>
            <TO-OBJECT name = "Evolution_j" type = "Evolution-Type_j"/>
        <MORPHISM>
    </MORPHISM>
</CATEGORY>
```

Figure 265: XML Specification of Category **Evolution**

```
<FUNCTOR name = "Time-Constraint" source-category = "Evolution"
            target-category = "Discrete-Time'">
    <OBJECT-MAPPING>
        <OBJECT-MAPPING source-object = "Evolution_i"
                        target-object = "Abstract-Time-Unit_i"/>
        <OBJECT-MAPPING source-object = "Evolution_j"
                        target-object = "Abstract-Time-Unit_j"/>
    </OBJECT-MAPPING>
    <MORPHISM-MAPPING>
```

```
        <MORPHSIM-MAPPING source-morphism = "Before"
                                  target-morphism = "Before"/>
        <MORPHSIM-MAPPING source-morphism = "Before"
                                  target-morphism = "Before"/>
    </MORPHISM-MAPPING>
</FUNCTOR>
```

Figure 266: XML Specification of Functor *Time-Constraint-for-Evolution*

```
<CATEGORY name = "EVOLUTION">
    <OBJECT>
        <OBJECT name = "Sequence_i" type = "Evolution-Sequence"/>
        <OBJECT name = "Sequence_j" type = "Evolution-Sequence"/>
    </OBJECT>
    <MORPHISM>
        <MORPHISM name = "Equivalent" type ="Preorder"/>
            <FROM-OBJECT name = "Sequence_i" type = "Evolution-Sequence"/>
            <TO-OBJECT name = "Sequence_j" type = "Evolution-Sequence"/>
        <MORPHISM>
    </MORPHISM>
</CATEGORY>
```

Figure 267: XML Specification of Category **EVOLUTION**

Appendix C: Representation of Categorical MAS Models in RASF

```
<CATEGORY name = "Plan">
    <OBJECT>
        <OBJECT name = "Action_i" type = "Action-Type_i"/>
        <OBJECT name = "Action_j" type = "Action-Type_j"/>
        <OBJECT name = "Action_Null" type = "Action-Type_Null"/>
    </OBJECT>
    <MORPHISM>
        <MORPHISM name = "Before" type = "Partial-Order"/>
            <FROM-OBJECT name = "Action_i" type = "Action-Type_i"/>
            <TO-OBJECT name = "Action_j" type = "Action-Type_j"/>
        </MORPHISM>
    </MORPHISM>
</CATEGORY>
```

Figure 268: XML Specification of Category **Plan**

```
<CATEGORY name = "PLAN">
    <OBJECT>
        <OBJECT name = "Plan_i" type = "Category"/>
        <OBJECT name = "Plan_j" type = "Category"/>
        <OBJECT name = "Plan_Null" type = "Category"/>
    </OBJECT>
    <MORPHISM>
        <MORPHISM name = "Before" type = "Functor"/>
            <FROM-OBJECT name = "Plan_i" type = "Category"/>
            <TO-OBJECT name = "Plan_j" type = "Category"/>
        </MORPHISM>
    </MORPHISM>
</CATEGORY>
```

Figure 269: XML Specification of Category **PLAN**

```
<FUNCTOR name = "Refined-by-Plan" source-category = "Plan"
                target-category = "PLAN">
    <OBJECT-MAPPING>
        <OBJECT-MAPPING source-object = "Action_i" target-object = "Plan_i"/>
        <OBJECT-MAPPING source-object = "Action_j" target-object = "Plan_j"/>
    </OBJECT-MAPPING>
    <MORPHISM-MAPPING>
        <MORPHSIM-MAPPING source-morphism = "Before"
```

```
                                    target-morphism = "Identity-Plan_i"/>
        <MORPHSIM-MAPPING source-morphism = "Before"
                                    target-morphism = "Identity-Plan_j"/>
    </MORPHISM-MAPPING>
</FUNCTOR>
```

Figure 270: XML Specification of Functor *Refined-by-Plan*

```
<FUNCTOR name = "Timing-Plan" source-category = "PLAN"
            target-category = "Discrete-Time">
    <OBJECT-MAPPING>
        <OBJECT-MAPPING source-object = "Plan_i"
                                target-object = "Abstract-Time-Unit_i"/>
        <OBJECT-MAPPING source-object = "Plan_j"
                                target-object = "Abstract-Time-Unit_j"/>
    </OBJECT-MAPPING>
    <MORPHISM-MAPPING>
        <MORPHSIM-MAPPING source-morphism = "Before"
                                    target-morphism = "Before"/>
        <MORPHSIM-MAPPING source-morphism = "Before"
                                    target-morphism = "Before"/>
    </MORPHISM-MAPPING>
</FUNCTOR>
```

Figure 271: XML Specification of Functor *Timing-Plan*

```
<CATEGORY name = "Goal-Type">
    <OBJECT>
        <OBJECT name = "Goal-Type_i"/>
        <OBJECT name = "Goal-Type_j"/>
    </OBJECT>
    <MORPHISM>
        <MORPHISM name = "Partial-Order"/>
            <FROM-OBJECT name = "Goal-Type_i"/>
            <TO-OBJECT name = "Goal-Type_j"/>
        </MORPHISM>
    </MORPHISM>
</CATEGORY>
```

Figure 272: XML Specification of Type Category **Goal-Type**

```
<CATEGORY name = "Goal-Type-Instance">
    <OBJECT>
```

```
        <OBJECT name = "Goal_i" type = "Goal-Type_i"/>
        <OBJECT name = "Goal_j" type = "Goal-Type_j"/>
    </OBJECT>
    <MORPHISM>
        <MORPHISM name = "Depend" type ="Partial-Order"/>
            <FROM-OBJECT name = "Goal_i" type = "Goal-Type_i"/>
            <TO-OBJECT name = "Goal_j" type = "Goal-Type_j"/>
        <MORPHISM>
    </MORPHISM>
</CATEGORY>
```

Figure 273: XML Specification of Category **Goal-Type-Instance**

```
<CATEGORY name = "GOAL">
    <OBJECT>
        <OBJECT name = "Goal_i" type = "Goal-Type_i"/>
        <OBJECT name = "Goal_j" type = "Goal-Type_j"/>
        <OBJECT name = "Goal_Null" type = "Goal-Type_Null"/>
    </OBJECT>
    <MORPHISM>
        <MORPHISM name = "Depend" type = "Partial-Order"/>
            <FROM-OBJECT name = "Goal_i" type = "Goal-Type_i"/>
            <TO-OBJECT name = "Goal_j" type = "Goal-Type_j"/>
        </MORPHISM>
    </MORPHISM>
</CATEGORY>
```

Figure 274: XML Specification of Category **GOAL**

```
<CATEGORY name = "Priority-Type">
    <OBJECT>
        <OBJECT name = "Priority-Type_i"/>
        <OBJECT name = "Priority-Type_j"/>
    </OBJECT>
    <MORPHISM>
        <MORPHISM name = "Partial-Order"/>
            <FROM-OBJECT name = "Priority-Type_i"/>
            <TO-OBJECT name = "Priority-Type_j"/>
        </MORPHISM>
    </MORPHISM>
</CATEGORY>
```

Figure 275: XML Specification of Type Category **Priority-Type**

```
<CATEGORY name = "Priority-Type-Instance">
    <OBJECT>
        <OBJECT name = "Priority_i" type = "Priority-Type_i"/>
        <OBJECT name = "Priority_j" type = "Priority-Type_j"/>
    </OBJECT>
    <MORPHISM>
        <MORPHISM name = "Higher-Than" type ="Partial-Order"/>
            <FROM-OBJECT name = "Priority_i" type = "Priority-Type_i"/>
            <TO-OBJECT name = "Priority_j" type = "Priority-Type_j"/>
        <MORPHISM>
    </MORPHISM>
</CATEGORY>
```

Figure 276: XML Specification of Category **Priority-Type-Instance**

```
<CATEGORY name = "Dependency">
    <OBJECT>
        <OBJECT name = "Priority_i" type = "Priority-Type_i"/>
        <OBJECT name = "Priority_j" type = "Priority-Type_j"/>
    </OBJECT>
    <MORPHISM>
        <MORPHISM name = "Higher-Than" type = "Partial-Order"/>
            <FROM-OBJECT name = "Priority_i" type = "Priority-Type_i"/>
            <TO-OBJECT name = "Priority_j" type = "Priority-Type_j"/>
        </MORPHISM>
    </MORPHISM>
</CATEGORY>
```

Figure 277: XML Specification of Category **Dependency**

```
<FUNCTOR name = "Assigned-Dependency" source-category = "GOAL"
                target-category = "Dependency">
    <OBJECT-MAPPING>
        <OBJECT-MAPPING source-object = "Goal_i" target-object = "Priority_i"/>
        <OBJECT-MAPPING source-object = "Goal_j" target-object = "Priority_j"/>
    </OBJECT-MAPPING>
    <MORPHISM-MAPPING>
        <MORPHSIM-MAPPING source-morphism = "Depend"
                                target-morphism = "Higher-Than"/>
        <MORPHSIM-MAPPING source-morphism = "Depend"
                                target-morphism = "Higher-Than"/>
    </MORPHISM-MAPPING>
```

```
</FUNCTOR>
```

Figure 278: XML Specification of Functor *Assigned-Dependency*

```
<CATEGORY name = "Fact-Type">
    <OBJECT>
        <OBJECT name = "Fact-Type_i"/>
        <OBJECT name = "Fact-Type_j"/>
    </OBJECT>
    <MORPHISM>
        <MORPHISM name = "Partial-Order"/>
            <FROM-OBJECT name = "Fact-Type_i"/>
            <TO-OBJECT name = "Fact-Type_j"/>
        </MORPHISM>
    </MORPHISM>
</CATEGORY>
```

Figure 279: XML Specification of Type Category **Fact-Type**

```
<CATEGORY name = "FactSet">
    <OBJECT>
        <OBJECT name = "Fact_i" type = "Fact-Type_i"/>
        <OBJECT name = "Fact_j" type = "Fact-Type_j"/>
        <OBJECT name = "Fact_Null" type = "Fact-Type_Null"/>
    </OBJECT>
    <MORPHISM>
        <MORPHISM name = "Identity-Fact_i" type = "Identity-Morphism"/>
            <FROM-OBJECT name = "Fact_i" type = "Fact-Type_i"/>
            <TO-OBJECT name = "Fact_i" type = "Fact-Type_i"/>
        </MORPHISM>
        <MORPHISM name = "Identity-Fact_j"/>
            <FROM-OBJECT name = "Fact_j" type = "Fact-Type_j"/>
            <TO-OBJECT name = "Fact_j" type = "Fact-Type_j"/>
        </MORPHISM>
        <MORPHISM name = "Identity-Fact_Null"/>
            <FROM-OBJECT name = "Fact_Null" type = "Fact-Type_Null"/>
            <TO-OBJECT name = "Fact_Null" type = "Fact-Type_Null"/>
        </MORPHISM>
    </MORPHISM>
</CATEGORY>
```

Figure 280: XML Specification of Discrete Category **FactSet**

```
<CATEGORY name = "FactSet_Null">
    <OBJECT>
    </OBJECT>
     <MORPHISM>
     </MORPHISM>
</CATEGORY>
```

Figure 281: XML Specification of Discrete Category **$FactSet_{Null}$**

```
<CATEGORY name = "FactSet_Base">
    <OBJECT>
        <OBJECT name = "Base-Fact_i" type = "Base-Fact-Type_i"/>
        <OBJECT name = "Base-Fact_j" type = "Base-Fact-Type_j"/>
        <OBJECT name = "Base-Fact_Null" type = "Base-Fact-Type_Null"/>
    </OBJECT>
     <MORPHISM>
        <MORPHISM name = "Identity-Base-Fact_i" type = "Identity-Morphism"/>
            <FROM-OBJECT name = "Base-Fact_i" type = "Base-Fact-Type_i"/>
            <TO-OBJECT name = "Base-Fact_i" type = "Base-Fact-Type_i"/>
        </MORPHISM>
        <MORPHISM name = "Identity-Base-Fact_j" type = "Identity-Morphism"/>
            <FROM-OBJECT name = "Base-Fact_j" type = "Base-Fact-Type_j"/>
            <TO-OBJECT name = "Base-Fact_j" type = "Base-Fact-Type_j"/>
        </MORPHISM>
        <MORPHISM name = "Identity-Base-Fact_Null"
                        type = "Identity-Morphism"/>
            <FROM-OBJECT name = "Base-Fact_Null"
                                type = "Base-Fact-Type_Null"/>
            <TO-OBJECT name = "Base-Fact_Null"
                            type = "Base-Fact-Type_Null"/>
        </MORPHISM>
     </MORPHISM>
</CATEGORY>
```

Figure 282: XML Specification of Discrete Category **$FactSet_{Base}$**

```
<CATEGORY name = "BELIEF">
    <OBJECT>
        <OBJECT name = "FactSet_Base" type = "FactSet"/>
        <OBJECT name = "FactSet_Null" type = "FactSet"/>
        <OBJECT name = "FactSet_i" type = "FactSet"/>
        <OBJECT name = "FactSet_j" type = "FactSet"/>
```

```
    </OBJECT>
    <MORPHISM>
        <MORPHISM name = "Subset-of" type = "Partial-Order"/>
            <FROM-OBJECT name = "FactSet_i" type = "FactSet"/>
            <TO-OBJECT name = "FactSet_j" type = "FactSet"/>
        </MORPHISM>
        <MORPHISM name = "Subset-of" type = "Partial-Order"/>
            <FROM-OBJECT name = "FactSet_Base" type = "FactSet"/>
            <TO-OBJECT name = "FactSet_i" type = "FactSet"/>
        </MORPHISM>
        <MORPHISM name = "Subset-of" type = "Partial-Order"/>
            <FROM-OBJECT name = "FactSet_Null" type = "FactSet"/>
            <TO-OBJECT name = "FactSet_i" type = "FactSet"/>
        </MORPHISM>
        <MORPHISM name = "Subset-of" type = "Partial-Order"/>
            <FROM-OBJECT name = "FactSet_Base" type = "FactSet"/>
            <TO-OBJECT name = "FactSet_j" type = "FactSet"/>
        </MORPHISM>
        <MORPHISM name = "Subset-of" type = "Partial-Order"/>
            <FROM-OBJECT name = "FactSet_Null" type = "FactSet"/>
            <TO-OBJECT name = "FactSet_j" type = "FactSet"/>
        </MORPHISM>
    </MORPHISM>
</CATEGORY>
```

Figure 283: XML Specification of Category **BELIEF**

```
<FUNCTOR name = "Plan-Goal" source-category = "PLAN"
                target-category = "GOAL">
    <OBJECT-MAPPING>
        <OBJECT-MAPPING source-object = "Plan_i" target-object = "Goal_i"/>
        <OBJECT-MAPPING source-object = "Plan_j" target-object = "Goal_j"/>
    </OBJECT-MAPPING>
    <MORPHISM-MAPPING>
        <MORPHSIM-MAPPING source-morphism = "Before"
                                target-morphism = "Depend"/>
        <MORPHSIM-MAPPING source-morphism = "Before"
                                target-morphism = "Depend"/>
    </MORPHISM-MAPPING>
</FUNCTOR>
```

Figure 284: XML Specification of Functor *Plan-Goal*

```
<FUNCTOR name = "Plan-Belief" source-category = "PLAN"
                target-category = "BELIEF">
    <OBJECT-MAPPING>
        <OBJECT-MAPPING source-object = "Plan_i" target-object = "FactSet_i"/>
        <OBJECT-MAPPING source-object = "Plan_j" target-object = "FactSet_j"/>
    </OBJECT-MAPPING>
    <MORPHISM-MAPPING>
        <MORPHSIM-MAPPING source-morphism = "Before"
                                target-morphism = "Identity-FactSet_Null"/>
        <MORPHSIM-MAPPING source-morphism = "Before"
                                target-morphism = "Identity-FactSet_Null"/>
    </MORPHISM-MAPPING>
</FUNCTOR>
```

Figure 285: XML Specification of Functor *Plan-Belief*

```
<FUNCTOR name = "Goal-Belief" source-category = "GOAL"
                target-category = "BELIEF">
    <OBJECT-MAPPING>
        <OBJECT-MAPPING source-object = "Goal_i" target-object = "FactSet_i"/>
        <OBJECT-MAPPING source-object = "Goal_j" target-object = "FactSet_j"/>
    </OBJECT-MAPPING>
    <MORPHISM-MAPPING>
        <MORPHSIM-MAPPING source-morphism = "Depend"
                                target-morphism = "Identity-FactSet_Null"/>
        <MORPHSIM-MAPPING source-morphism = "Depend"
                                target-morphism = "Identity-FactSet_Null"/>
    </MORPHISM-MAPPING>
</FUNCTOR>
```

Figure 286: XML Specification of Functor *Goal-Belief*

```
<CATEGORY name = "AGENT">
    <OBJECT>
        <OBJECT name = "Action" type = "Category"/>
        <OBJECT name = "Plan" type = "Category"/>
        <OBJECT name = "PLAN" type = "Category"/>
        <OBJECT name = "GOAL" type = "Category"/>
        <OBJECT name = "BELIEF" type = "Category"/>
        <OBJECT name = "FactSet" type = "Category"/>
    </OBJECT>
    <MORPHISM>
```

```
        <MORPHISM name = "Plan-Goal" type = "Functor"/>
            <FROM-OBJECT name = "PLAN" type = "Category"/>
            <TO-OBJECT name = "GOAL" type = "Category"/>
        </MORPHISM>
        <MORPHISM name = "Plan-Belief" type = "Functor"/>
            <FROM-OBJECT name = "PLAN" type = "Category"/>
            <TO-OBJECT name = "BELIEF" type = "Category"/>
        </MORPHISM>
        <MORPHISM name = "Goal-Belief" type = "Functor"/>
            <FROM-OBJECT name = "GOAL" type = "Category"/>
            <TO-OBJECT name = "BELIEF" type = "Category"/>
        </MORPHISM>
        <MORPHISM name = "Refined-by-Plan" type = "Functor"/>
            <FROM-OBJECT name = "Plan" type = "Category"/>
            <TO-OBJECT name = "PLAN" type = "Category"/>
        </MORPHISM>
        <MORPHISM name = "Sequence-Action" type = "Functor"/>
            <FROM-OBJECT name = "Action" type = "Category"/>
            <TO-OBJECT name = "Plan" type = "Category"/>
        </MORPHISM>
    </MORPHISM>
</CATEGORY>
```

Figure 287: XML Specification of Category **AGENT**

```
<CATEGORY name = "Agent-Type">
    <OBJECT>
        <OBJECT name = "Agent-Type_i"/>
        <OBJECT name = "Agent-Type_j"/>
    </OBJECT>
    <MORPHISM>
        <MORPHISM name = "Communication-Type_n"/>
            <FROM-OBJECT name = "Agent-Type_i"/>
            <TO-OBJECT name = "Agent-Type_j"/>
        </MORPHISM>
    </MORPHISM>
</CATEGORY>
```

Figure 288: XML Specification of Type Category **Agent-Type**

```
<CATEGORY name = "Agent-Type-Instance">
    <OBJECT>
```

```
        <OBJECT name = "Agent_i" type = "Agent-Type_i"/>
        <OBJECT name = "Agent_j" type = "Agent-Type_j"/>
    </OBJECT>
    <MORPHISM>
        <MORPHISM name = "Communication_n" type ="Communication-Type_n"/>
            <FROM-OBJECT name = "Agent_i" type = "Agent-Type_i"/>
            <TO-OBJECT name = "Agent_j" type = "Agent-Type_j"/>
        <MORPHISM>
    </MORPHISM>
</CATEGORY>
```

Figure 289: XML Specification of Category **Agent-Type-Instance**

```
<CATEGORY name = "MAS">
    <OBJECT>
        <OBJECT name = "Agent_i" type = "Agent-Type_i"/>
        <OBJECT name = "Agent_j" type = "Agent-Type_j"/>
    </OBJECT>
    <MORPHISM>
        <MORPHISM name = "Communication_n"
                    type = "Communication-Type_n"/>
            <FROM-OBJECT name = "Agent_i" type = "Agent-Type_i"/>
            <TO-OBJECT name = "Agent_j" type = "Agent-Type_j"/>
        </MORPHISM>
    </MORPHISM>
</CATEGORY>
```

Figure 290: XML Specification of Category **MAS**

```
<CATEGORY name = "Repository-Type">
    <OBJECT>
        <OBJECT name = "Repository-Type_i"/>
        <OBJECT name = "Repository-Type_j"/>
    </OBJECT>
    <MORPHISM>
        <MORPHISM name = "Connection-Type_n"/>
            <FROM-OBJECT name = "Repository-Type_i"/>
            <TO-OBJECT name = "Repository-Type_j"/>
        </MORPHISM>
    </MORPHISM>
</CATEGORY>
```

Figure 291: XML Specification of Type Category **Repository-Type**

```
<CATEGORY name = "Repository-Type-Instance">
    <OBJECT>
        <OBJECT name = "Repository_i" type = "Repository-Type_i" />
        <OBJECT name = "Repository_j" type = "Repository-Type_j"/>
    </OBJECT>
    <MORPHISM>
        <MORPHISM name = "Connection_n" type ="Connection-Type_n"/>
            <FROM-OBJECT name = "Repository_i" type = "Repository-Type_i"/>
            <TO-OBJECT name = "Repository_j" type = "Repository-Type_j"/>
        </MORPHISM>
    </MORPHISM>
</CATEGORY>
```

Figure 292: XML Specification of Category **Repository-Type-Instance**

```
<FUNCTOR name = "Repository-Access" source-category = "MAS"
            target-category = "Repository-Type-Instance">
    <OBJECT-MAPPING>
        <OBJECT-MAPPING source-object = "Agent_i"
                        target-object = "Repository_i"/>
        <OBJECT-MAPPING source-object = "Agent_j"
                        target-object = "Repository_j"/>
    </OBJECT-MAPPING>
    <MORPHISM-MAPPING>
        <MORPHSIM-MAPPING source-morphism = "Communication_n"
                          target-morphism = "Connection_n"/>
        <MORPHSIM-MAPPING source-morphism = "Communication_m"
                          target-morphism = "Connection_m"/>
    </MORPHISM-MAPPING>
</FUNCTOR>
```

Figure 293: XML Specification of Functor *Repository-Access*

```
<CATEGORY name = "MAS-Type">
    <OBJECT>
        <OBJECT name = "MAS-Type_i"/>
        <OBJECT name = "MAS-Type_j"/>
    </OBJECT>
    <MORPHISM>
        <MORPHISM name = "Communication-Type_n"/>
            <FROM-OBJECT name = "MAS-Type_i"/>
            <TO-OBJECT name = "MAS-Type_j"/>
```

```
        </MORPHISM>
    </MORPHISM>
</CATEGORY>
```

Figure 294: XML Specification of Type Category **MAS-Type**

```
<CATEGORY name = "MAS-Type-Instance">
    <OBJECT>
        <OBJECT name = "MAS_i" type = "MAS-Type_i" />
        <OBJECT name = "MAS_j" type = "MAS-Type_j"/>
    </OBJECT>
    <MORPHISM>
        <MORPHISM name = "Communication_n" type ="Communication-Type_n"/>
            <FROM-OBJECT name = "MAS_i" type = "MAS-Type_i"/>
            <TO-OBJECT name = "MAS_j" type = "MAS-Type_j"/>
        </MORPHISM>
    </MORPHISM>
</CATEGORY>
```

Figure 295: XML Specification of Category **MAS-Type-Instance**

Appendix D: Representation of Categorical Self-Healing

```
<CATEGORY name = "Take-over-Work-Flow-for-Self-Healing">
    <OBJECT>
        <OBJECT name = "Restart" type = "Work-Flow-Action"/>
        <OBJECT name = "NoHeartbeat" type = "Work-Flow-Action"/>
        <OBJECT name = "RequestRAE" type = "Work-Flow-Action"/>
        <OBJECT name = "NotFound" type = "Work-Flow-Action"/>
        <OBJECT name = "Take-over" type = "Work-Flow-Action"/>
        <OBJECT name = "Confirmed" type = "Work-Flow-Action"/>
    </OBJECT>
    <MORPHISM>
        <MORPHISM name = "Before" type ="Preorder"/>
            <FROM-OBJECT name = "Restart" type = "Work-Flow-Action"/>
            <TO-OBJECT name = "NoHeartbeat" type = "Work-Flow-Action"/>
        <MORPHISM>
        <MORPHISM name = "Before" type ="Preorder"/>
            <FROM-OBJECT name = "NoHeartbeat"
                                type = "Work-Flow-Action"/>
            <TO-OBJECT name = "RequestRAE" type = "Work-Flow-Action"/>
        <MORPHISM>
        <MORPHISM name = "Before" type ="Preorder"/>
            <FROM-OBJECT name = "RequestRAE" type = "Work-Flow-Action"/>
            <TO-OBJECT name = "NotFound" type = "Work-Flow-Action"/>
        <MORPHISM>
        <MORPHISM name = "Before" type ="Preorder"/>
            <FROM-OBJECT name = "NotFound" type = "Work-Flow-Action"/>
            <TO-OBJECT name = "Take-over" type = "Work-Flow-Action"/>
        <MORPHISM>
        <MORPHISM name = "Before" type ="Preorder"/>
            <FROM-OBJECT name = "Take-over" type = "Work-Flow-Action"/>
            <TO-OBJECT name = "Confirmed" type = "Work-Flow-Action"/>
        <MORPHISM>
    </MORPHISM>
</CATEGORY>
```

Figure 296: XML Specification of Category **Take-Over-Flow-Self-Healing**

```
<CATEGORY name = "Intelligent-Control-Loop-Time-for-Self-Healing">
    <OBJECT>
        <OBJECT name = "t0" type = "Integer"/>
        <OBJECT name = "t1" type = "Integer"/>
```

```
        <OBJECT name = "t2" type = "Integer"/>
        <OBJECT name = "t3" type = "Integer"/>
    </OBJECT>
    <MORPHISM>
        <MORPHISM name = "Before" type ="Preorder"/>
            <FROM-OBJECT name = "t0" type = "Integer"/>
            <TO-OBJECT name = "t1" type = "Integer"/>
        <MORPHISM>
        <MORPHISM name = "Before" type ="Preorder"/>
            <FROM-OBJECT name = "t0" type = "Integer"/>
            <TO-OBJECT name = "t2" type = "Integer"/>
        <MORPHISM>
        <MORPHISM name = "Before" type ="Preorder"/>
            <FROM-OBJECT name = "t0" type = "Integer"/>
            <TO-OBJECT name = "t3" type = "Integer"/>
        <MORPHISM>
        <MORPHISM name = "Before" type ="Preorder"/>
            <FROM-OBJECT name = "t1" type = "Integer"/>
            <TO-OBJECT name = "t2" type = "Integer"/>
        <MORPHISM>
        <MORPHISM name = "Before" type ="Preorder"/>
            <FROM-OBJECT name = "t1" type = "Integer"/>
            <TO-OBJECT name = "t3" type = "Integer"/>
        <MORPHISM>
        <MORPHISM name = "Before" type ="Preorder"/>
            <FROM-OBJECT name = "t2" type = "Integer"/>
            <TO-OBJECT name = "t3" type = "Integer"/>
        <MORPHISM>
    </MORPHISM>
</CATEGORY>
```

Figure 297: XML Specification of Category **ICL-Time-Self-Healing**

```
<FUNCTOR name = "Intelligent-Control-Loop-Time-Constraint-Self-Healing"
            source-category = "ICL-State-Self-Healing"
            target-category = "ICL-Time-Self-Healing'">
    <OBJECT-MAPPING>
        <OBJECT-MAPPING source-object = "Analyze" target-object = "t0"/>
        <OBJECT-MAPPING source-object = "Analyze" target-object = "t1"/>
        <OBJECT-MAPPING source-object = "Plan" target-object = "t2"/>
        <OBJECT-MAPPING source-object = "Plan" target-object = "t3"/>
    </OBJECT-MAPPING>
```

```
    <MORPHISM-MAPPING>
        <MORPHSIM-MAPPING source-morphism = "Restart-RAE"
                          target-morphism = "Before"/>
        <MORPHSIM-MAPPING source-morphism = "Has-Action"
                          target-morphism = "Before"/>
        <MORPHSIM-MAPPING source-morphism = "Substitute"
                          target-morphism = "Before"/>
        <MORPHSIM-MAPPING source-morphism = "Take-Over"
                          target-morphism = "Before"/>
        <MORPHSIM-MAPPING source-morphism = "Action-Done"
                          target-morphism = "Before"/>
    </MORPHISM-MAPPING>
</FUNCTOR>
```

Figure 298: XML Specification of Category **Time-Constraint-Self-Healing**

```
<FUNCTOR name = "RACG-Self-Healing-Restart"
            source-category = "RACG1-1"
            target-category = "RACG1-0">
    <OBJECT-MAPPING>
        <OBJECT-MAPPING source-object = "RAOL1-1"
                        target-object = "RAOL1"/>
        <OBJECT-MAPPING source-object = "RAOL2"
                        target-object = "RAOL2"/>
        <OBJECT-MAPPING source-object = "RACS1"
                        target-object = "RACS1"/>
    </OBJECT-MAPPING>
    <MORPHISM-MAPPING>
        <MORPHSIM-MAPPING source-morphism = "Command1-1"
                          target-morphism = "Command1"/>
        <MORPHSIM-MAPPING source-morphism = "Command2"
                          target-morphism = "Command2"/>
        <MORPHSIM-MAPPING source-morphism = "Report1-1"
                          target-morphism = "Report1"/>
        <MORPHSIM-MAPPING source-morphism = "Report2"
                          target-morphism = "Report2"/>
        <MORPHSIM-MAPPING source-morphism = "Cooperate1-1"
                          target-morphism = "Cooperate1"/>
        <MORPHSIM-MAPPING source-morphism = "Cooperate2"
                          target-morphism = "Cooperate2"/>
    </MORPHISM-MAPPING>
```

```
</FUNCTOR>
```

Figure 299: XML Specification of Functor *RACG-Self-Healing-Restart*

```
<FUNCTOR name = "RACG-Self-Healing-Substitute"
                source-category = "RACG1-2"
                target-category = "RACG1-0">
    <OBJECT-MAPPING>
        <OBJECT-MAPPING source-object = "RAOL3"
                                target-object = "RAOL1"/>
        <OBJECT-MAPPING source-object = "RAOL2"
                                target-object = "RAOL2"/>
        <OBJECT-MAPPING source-object = "RACS1"
                                target-object = "RACS1"/>
    </OBJECT-MAPPING>
    <MORPHISM-MAPPING>
        <MORPHSIM-MAPPING source-morphism = "Command3"
                                    target-morphism = "Command1"/>
        <MORPHSIM-MAPPING source-morphism = "Command2"
                                    target-morphism = "Command2"/>
        <MORPHSIM-MAPPING source-morphism = "Report3"
                                    target-morphism = "Report1"/>
        <MORPHSIM-MAPPING source-morphism = "Report2"
                                    target-morphism = "Report2"/>
        <MORPHSIM-MAPPING source-morphism = "Cooperate3"
                                    target-morphism = "Cooperate1"/>
        <MORPHSIM-MAPPING source-morphism = "Cooperate2"
                                    target-morphism = "Cooperate2"/>
    </MORPHISM-MAPPING>
</FUNCTOR>
```

Figure 300: XML Specification of Functor *RACG-Self-Healing-Substitute*

```
<FUNCTOR name = "RACG-Self-Healing-Take-Over"
                source-category = "RACG1-3"
                target-category = "RACG1-0">
    <OBJECT-MAPPING>
        <OBJECT-MAPPING source-object = "SPM2" target-object = "RAOL1"/>
        <OBJECT-MAPPING source-object = "SPM2" target-object = "RAOL2"/>
        <OBJECT-MAPPING source-object = "RACS1-1"
                                target-object = "RACS1"/>
    </OBJECT-MAPPING>
```

```
    <MORPHISM-MAPPING>
        <MORPHSIM-MAPPING source-morphism = "Command6"
                                target-morphism = "Command1"/>
        <MORPHSIM-MAPPING source-morphism = "Command7"
                                target-morphism = "Command2"/>
        <MORPHSIM-MAPPING source-morphism = "Report6"
                                target-morphism = "Report1"/>
        <MORPHSIM-MAPPING source-morphism = "Report7"
                                target-morphism = "Report2"/>
        <MORPHSIM-MAPPING source-morphism = "Cooperate6"
                                target-morphism = "Cooperate1"/>
        <MORPHSIM-MAPPING source-morphism = "Cooperate7"
                                target-morphism = "Cooperate2"/>
    </MORPHISM-MAPPING>
</FUNCTOR>
```

Figure 301: XML Specification of Functor *RACG-Self-Healing-Take-Over*

```
<NATURAL-TRANSFORMATION name = "Relation-of-RACG-Evolution">
    <ARROW>
        <ARROW name = "Relation4"/>
            <FROM-FUNCTOR name = "RACG-Self-Healing-Restart"
                                type = "RACG-Evolution-Self-Healing"/>
            <TO-FUNCTOR name = "RACG-Self-Healing-Substitute"
                                type = "RACG-Evolution-Self-Healing"/>
        </ARROW>
        <ARROW name = "Relation5"/>
            <FROM-FUNCTOR name = "RACG-Self-Healing-Restart"
                                type = "RACG-Evolution-Self-Healing"/>
            <TO-FUNCTOR name = "RACG-Self-Healing-Take-Over"
                                type = "RACG-Evolution-Self-Healing"/>
        </ARROW>
        <ARROW name = "Relation6"/>
            <FROM-FUNCTOR name = "RACG-Self-Healing-Substitute"
                                type = "RACG-Evolution-Self-Healing"/>
            <TO-FUNCTOR name = "RACG-Self-Healing-Take-Over"
                                type = "RACG-Evolution-Self-Healing"/>
        </ARROW>
    </ARROW>
</NATURAL-TRANSFORMATION>
```

Figure 302: XML Specification of Natural Transformation *RACG-Evolution-Relation*

```
<CATEGORY name = "Relation-Set-of-RACG-Evolution-Self-Healing">
    <OBJECT>
        <OBJECT name = "RACG-Self-Healing-Restart"
                    type = "RACG-Evolution-Self-Healing" />
        <OBJECT name = "RACG-Self-Healing-Substitute"
                    type = "RACG-Evolution-Self-Healing"/>
        <OBJECT name = "RACG-Self-Healing-Take-Over"
                    type = "RACG-Evolution-Self-Healing"/>
    </OBJECT>
    <MORPHISM>
        <MORPHISM name = "Relation4"
                        type ="RACG-Evolution-Relation-Self-Healing"/>
            <FROM-OBJECT name = "RACG-Self-Healing-Restart"
                            type = "RACG-Evolution-Self-Healing" />
            <TO-OBJECT name = "RACG-Self-Healing-Substitute"
                        type = "RACG-Evolution-Self-Healing"/>
        </MORPHISM>
        <MORPHISM name = "Relation5"
                        type ="RACG-Evolution-Relation-Self-Healing"/>
            <FROM-OBJECT name = "RACG-Self-Healing-Restart"
                            type = "RACG-Evolution-Self-Healing" />
            <TO-OBJECT name = "RACG-Self-Healing-Take-Over"
                        type = "RACG-Evolution-Self-Healing"/>
        </MORPHISM>
        <MORPHISM name = "Relation6"
                        type ="RACG-Evolution-Relation-Self-Healing"/>
            <FROM-OBJECT name = "RACG-Self-Healing-Substitute"
                            type = "RACG-Evolution-Self-Healing" />
            <TO-OBJECT name = "RACG-Self-Healing-Take-Over"
                        type = "RACG-Evolution-Self-Healing"/>
        </MORPHISM>
    </MORPHISM>
</CATEGORY>
```

Figure 303: XML Specification of Functor Category **RACG-Evolution-Relation-Set**

Appendix E: Representation of Categorical Self-Configuration

```
<CATEGORY name = "Self-Configuration-Work-Flow">
    <OBJECT>
        <OBJECT name = "ValidateRACGM" type = "Work-Flow-Action"/>
        <OBJECT name = "ValidateRACS" type = "Work-Flow-Action"/>
        <OBJECT name = "LaunchInvestigation" type = "Work-Flow-Action"/>
        <OBJECT name = "ValidateRACScommunication"
                    type = "Work-Flow-Action"/>
        <OBJECT name = "Conform" type = "Work-Flow-Action"/>
        <OBJECT name = "NotConform" type = "Work-Flow-Action"/>
    </OBJECT>
     <MORPHISM>
        <MORPHISM name = "Before" type ="Preorder"/>
            <FROM-OBJECT name = "ValidateRACGM"
                                type = "Work-Flow-Action"/>
            <TO-OBJECT name = "Conform" type = "Work-Flow-Action"/>
        <MORPHISM>
        <MORPHISM name = "Before" type ="Preorder"/>
            <FROM-OBJECT name = "ValidateRACGM"
                                type = "Work-Flow-Action"/>
            <TO-OBJECT name = "NotConform" type = "Work-Flow-Action"/>
        <MORPHISM>
        <MORPHISM name = "Before" type ="Preorder"/>
            <FROM-OBJECT name = "ValidateRACS"
                                type = "Work-Flow-Action"/>
            <TO-OBJECT name = "Conform" type = "Work-Flow-Action"/>
        <MORPHISM>
        <MORPHISM name = "Before" type ="Preorder"/>
            <FROM-OBJECT name = "ValidateRACS"
                                type = "Work-Flow-Action"/>
            <TO-OBJECT name = "NotConform" type = "Work-Flow-Action"/>
        <MORPHISM>
        <MORPHISM name = "Before" type ="Preorder"/>
            <FROM-OBJECT name = "ValidateRACScommunication"
                                type = "Work-Flow-Action"/>
            <TO-OBJECT name = "Conform" type = "Work-Flow-Action"/>
        <MORPHISM>
        <MORPHISM name = "Before" type ="Preorder"/>
            <FROM-OBJECT name = "ValidateRACScommunication"
                                type = "Work-Flow-Action"/>
```

```
            <TO-OBJECT name = "NotConform" type = "Work-Flow-Action"/>
        <MORPHISM>
        <MORPHISM name = "Before" type ="Preorder"/>
            <FROM-OBJECT name = "NotConform" type = "Work-Flow-Action"/>
            <TO-OBJECT name = "LaunchInvestigation"
                              type = "Work-Flow-Action"/>
        <MORPHISM>
    </MORPHISM>
</CATEGORY>
```

Figure 304: XML Specification of Category **Self-Configuration-Work-Flow**

```
<CATEGORY name = "Substitution-Work-Flow-for-Self-Configuration">
    <OBJECT>
        <OBJECT name = "WrongCommType" type = "Work-Flow-Action"/>
        <OBJECT name = "SelfViolation" type = "Work-Flow-Action"/>
        <OBJECT name = "Request" type = "Work-Flow-Action"/>
        <OBJECT name = "Confirm" type = "Work-Flow-Action"/>
        <OBJECT name = "Register" type = "Work-Flow-Action"/>
        <OBJECT name = "Heartbeat" type = "Work-Flow-Action"/>
        <OBJECT name = "Connect" type = "Work-Flow-Action"/>
    </OBJECT>
    <MORPHISM>
        <MORPHISM name = "Before" type ="Preorder"/>
            <FROM-OBJECT name = "WrongCommType"
                              type = "Work-Flow-Action"/>
            <TO-OBJECT name = "SelfViolation" type = "Work-Flow-Action"/>
        <MORPHISM>
        <MORPHISM name = "Before" type ="Preorder"/>
            <FROM-OBJECT name = "SelfViolation"
                              type = "Work-Flow-Action"/>
            <TO-OBJECT name = "Request" type = "Work-Flow-Action"/>
        <MORPHISM>
        <MORPHISM name = "Before" type ="Preorder"/>
            <FROM-OBJECT name = "Request" type = "Work-Flow-Action"/>
            <TO-OBJECT name = "Confirm" type = "Work-Flow-Action"/>
        <MORPHISM>
        <MORPHISM name = "Before" type ="Preorder"/>
            <FROM-OBJECT name = "Confirm" type = "Work-Flow-Action"/>
            <TO-OBJECT name = "Register" type = "Work-Flow-Action"/>
        <MORPHISM>
        <MORPHISM name = "Before" type ="Preorder"/>
```

```
            <FROM-OBJECT name = "Register" type = "Work-Flow-Action"/>
            <TO-OBJECT name = "Heartbeat" type = "Work-Flow-Action"/>
        <MORPHISM>
        <MORPHISM name = "Before" type ="Preorder"/>
            <FROM-OBJECT name = "Heartbeat" type = "Work-Flow-Action"/>
            <TO-OBJECT name = "Connect" type = "Work-Flow-Action"/>
        <MORPHISM>
    </MORPHISM>
</CATEGORY>
```

Figure 305: XML Specification of Category **Substitution-Flow-Self-Configuration**

```
<CATEGORY name = "Take-over-Work-Flow-for-Self-Configuration">
    <OBJECT>
        <OBJECT name = "WrongCommType" type = "Work-Flow-Action"/>
        <OBJECT name = "SelfViolation" type = "Work-Flow-Action"/>
        <OBJECT name = "Take-over" type = "Work-Flow-Action"/>
        <OBJECT name = "Confirm" type = "Work-Flow-Action"/>
        <OBJECT name = "Connect" type = "Work-Flow-Action"/>
        <OBJECT name = "Heartbeat" type = "Work-Flow-Action"/>
    </OBJECT>
    <MORPHISM>
        <MORPHISM name = "Before" type ="Preorder"/>
            <FROM-OBJECT name = "WrongCommType"
                                type = "Work-Flow-Action"/>
            <TO-OBJECT name = "SelfViolation" type = "Work-Flow-Action"/>
        <MORPHISM>
        <MORPHISM name = "Before" type ="Preorder"/>
            <FROM-OBJECT name = "SelfViolation"
                                type = "Work-Flow-Action"/>
            <TO-OBJECT name = "Take-over" type = "Work-Flow-Action"/>
        <MORPHISM>
        <MORPHISM name = "Before" type ="Preorder"/>
            <FROM-OBJECT name = "Take-over" type = "Work-Flow-Action"/>
            <TO-OBJECT name = "Confirm" type = "Work-Flow-Action"/>
        <MORPHISM>
        <MORPHISM name = "Before" type ="Preorder"/>
            <FROM-OBJECT name = "Confirm" type = "Work-Flow-Action"/>
            <TO-OBJECT name = "Connect" type = "Work-Flow-Action"/>
        <MORPHISM>
        <MORPHISM name = "Before" type ="Preorder"/>
            <FROM-OBJECT name = "Connect" type = "Work-Flow-Action"/>
```

```
            <TO-OBJECT name = "Heartbeat" type = "Work-Flow-Action"/>
        <MORPHISM>
    </MORPHISM>
</CATEGORY>
```

Figure 306: XML Specification of Category **Take-over-Flow-Self-Configuration**

```
<CATEGORY name = "Intelligent-Control-Loop-Time-for-Self-Configuration">
    <OBJECT>
        <OBJECT name = "t0" type = "Integer"/>
        <OBJECT name = "t4" type = "Integer"/>
        <OBJECT name = "t5" type = "Integer"/>
        <OBJECT name = "t6" type = "Integer"/>
    </OBJECT>
    <MORPHISM>
        <MORPHISM name = "Before" type ="Preorder"/>
            <FROM-OBJECT name = "t0" type = "Integer"/>
            <TO-OBJECT name = "t4" type = "Integer"/>
        <MORPHISM>
        <MORPHISM name = "Before" type ="Preorder"/>
            <FROM-OBJECT name = "t0" type = "Integer"/>
            <TO-OBJECT name = "t5" type = "Integer"/>
        <MORPHISM>
        <MORPHISM name = "Before" type ="Preorder"/>
            <FROM-OBJECT name = "t0" type = "Integer"/>
            <TO-OBJECT name = "t6" type = "Integer"/>
        <MORPHISM>
        <MORPHISM name = "Before" type ="Preorder"/>
            <FROM-OBJECT name = "t4" type = "Integer"/>
            <TO-OBJECT name = "t5" type = "Integer"/>
        <MORPHISM>
        <MORPHISM name = "Before" type ="Preorder"/>
            <FROM-OBJECT name = "t4" type = "Integer"/>
            <TO-OBJECT name = "t6" type = "Integer"/>
        <MORPHISM>
        <MORPHISM name = "Before" type ="Preorder"/>
            <FROM-OBJECT name = "t5" type = "Integer"/>
            <TO-OBJECT name = "t6" type = "Integer"/>
        <MORPHISM>
    </MORPHISM>
```

```
</CATEGORY>
```

Figure 307: XML Specification of Category **ICL-Time-Self-Configuration**

```
<FUNCTOR name = "ICL-Time-Constraint-Self-Configuration"
          source-category = "ICL-State-Self-Configuration"
          target-category = "ICL-Time-Self-Configuration'">
    <OBJECT-MAPPING>
        <OBJECT-MAPPING source-object = "Analyze" target-object = "t0"/>
        <OBJECT-MAPPING source-object = "Analyze" target-object = "t4"/>
        <OBJECT-MAPPING source-object = "Plan" target-object = "t5"/>
        <OBJECT-MAPPING source-object = "Plan" target-object = "t6"/>
    </OBJECT-MAPPING>
    <MORPHISM-MAPPING>
        <MORPHSIM-MAPPING source-morphism = "Restart-RAE"
                          target-morphism = "Before"/>
        <MORPHSIM-MAPPING source-morphism = "Launch-Self-Healing"
                          target-morphism = "Before"/>
        <MORPHSIM-MAPPING source-morphism = "Substitute"
                          target-morphism = "Before"/>
        <MORPHSIM-MAPPING source-morphism = "Take-Over"
                          target-morphism = "Before"/>
        <MORPHSIM-MAPPING source-morphism = "Action-Done"
                          target-morphism = "Before"/>
    </MORPHISM-MAPPING>
</FUNCTOR>
```

Figure 308: XML Specification of Category **Time-Constraint-Self-Configuration**

```
<FUNCTOR name = "RAC-Self-Configuration-RestartRAOL"
          source-category = "RAC1-4"
          target-category = "RAC1-0'">
    <OBJECT-MAPPING>
        <OBJECT-MAPPING source-object = "RAO1" target-object = "RAO1"/>
        <OBJECT-MAPPING source-object = "RAO2" target-object = "RAO2"/>
        <OBJECT-MAPPING source-object = "RAO3" target-object = "RAO3"/>
        <OBJECT-MAPPING source-object = "RAOL1-1"
                        target-object = "RAOL1"/>
    </OBJECT-MAPPING>
    <MORPHISM-MAPPING>
        <MORPHSIM-MAPPING source-morphism = "Command1-1"
                          target-morphism = "Command1"/>
```

```
        <MORPHSIM-MAPPING source-morphism = "Command2-1"
                          target-morphism = "Command2"/>
        <MORPHSIM-MAPPING source-morphism = "Command3-1"
                          target-morphism = "Command3"/>
        <MORPHSIM-MAPPING source-morphism = "Report1-1"
                          target-morphism = "Report1"/>
        <MORPHSIM-MAPPING source-morphism = "Report2-1"
                          target-morphism = "Report2"/>
        <MORPHSIM-MAPPING source-morphism = "Report3-1"
                          target-morphism = "Report3"/>
        <MORPHSIM-MAPPING source-morphism = "Cooperate1"
                          target-morphism = "Cooperate1"/>
        <MORPHSIM-MAPPING source-morphism = "Cooperate2"
                          target-morphism = "Cooperate2"/>
        <MORPHSIM-MAPPING source-morphism = "Cooperate3"
                          target-morphism = "Cooperate3"/>
        <MORPHSIM-MAPPING source-morphism = "Cooperate4"
                          target-morphism = "Cooperate4"/>
        <MORPHSIM-MAPPING source-morphism = "Cooperate5"
                          target-morphism = "Cooperate5"/>
        <MORPHSIM-MAPPING source-morphism = "Cooperate6"
                          target-morphism = "Cooperate6"/>
    </MORPHISM-MAPPING>
</FUNCTOR>
```

Figure 309: XML Specification of Functor *RAC-Self-Configuration-RestartRAOL*

```
<FUNCTOR name = "RAC-Self-Configuration-SubstituteRAOL"
         source-category = "RAC1-5"
         target-category = "RAC1-0'">
    <OBJECT-MAPPING>
        <OBJECT-MAPPING source-object = "RAO1" target-object = "RAO1"/>
        <OBJECT-MAPPING source-object = "RAO2" target-object = "RAO2"/>
        <OBJECT-MAPPING source-object = "RAO3" target-object = "RAO3"/>
        <OBJECT-MAPPING source-object = "RAOL3"
                        target-object = "RAOL1"/>
    </OBJECT-MAPPING>
    <MORPHISM-MAPPING>
        <MORPHSIM-MAPPING source-morphism = "Command4"
                          target-morphism = "Command1"/>
        <MORPHSIM-MAPPING source-morphism = "Command5"
                          target-morphism = "Command2"/>
```

```
        <MORPHSIM-MAPPING source-morphism = "Command6"
                          target-morphism = "Command3"/>
        <MORPHSIM-MAPPING source-morphism = "Report4"
                          target-morphism = "Report1"/>
        <MORPHSIM-MAPPING source-morphism = "Report5"
                          target-morphism = "Report2"/>
        <MORPHSIM-MAPPING source-morphism = "Report6"
                          target-morphism = "Report3"/>
        <MORPHSIM-MAPPING source-morphism = "Cooperate1"
                          target-morphism = "Cooperate1"/>
        <MORPHSIM-MAPPING source-morphism = "Cooperate2"
                          target-morphism = "Cooperate2"/>
        <MORPHSIM-MAPPING source-morphism = "Cooperate3"
                          target-morphism = "Cooperate3"/>
        <MORPHSIM-MAPPING source-morphism = "Cooperate4"
                          target-morphism = "Cooperate4"/>
        <MORPHSIM-MAPPING source-morphism = "Cooperate5"
                          target-morphism = "Cooperate5"/>
        <MORPHSIM-MAPPING source-morphism = "Cooperate6"
                          target-morphism = "Cooperate6"/>
    </MORPHISM-MAPPING>
</FUNCTOR>
```

Figure 310: XML Specification of Functor *RAC-Self-Configuration-SubstituteRAOL*

```
<FUNCTOR name = "RAC-Self-Configuration-Take-over-RAOL"
                source-category = "RAC1-6"
                target-category = "RAC1-0'">
    <OBJECT-MAPPING>
        <OBJECT-MAPPING source-object = "RAO1" target-object = "RAO1"/>
        <OBJECT-MAPPING source-object = "RAO2" target-object = "RAO2"/>
        <OBJECT-MAPPING source-object = "RAO3" target-object = "RAO3"/>
        <OBJECT-MAPPING source-object = "SPM" target-object = "RAOL1"/>
    </OBJECT-MAPPING>
    <MORPHISM-MAPPING>
        <MORPHSIM-MAPPING source-morphism = "Command7"
                          target-morphism = "Command1"/>
        <MORPHSIM-MAPPING source-morphism = "Command8"
                          target-morphism = "Command2"/>
        <MORPHSIM-MAPPING source-morphism = "Command9"
                          target-morphism = "Command3"/>
        <MORPHSIM-MAPPING source-morphism = "Report7"
```

```
                                    target-morphism = "Report1"/>
        <MORPHSIM-MAPPING source-morphism = "Report8"
                                    target-morphism = "Report2"/>
        <MORPHSIM-MAPPING source-morphism = "Report9"
                                    target-morphism = "Report3"/>
        <MORPHSIM-MAPPING source-morphism = "Cooperate1"
                                    target-morphism = "Cooperate1"/>
        <MORPHSIM-MAPPING source-morphism = "Cooperate2"
                                    target-morphism = "Cooperate2"/>
        <MORPHSIM-MAPPING source-morphism = "Cooperate3"
                                    target-morphism = "Cooperate3"/>
        <MORPHSIM-MAPPING source-morphism = "Cooperate4"
                                    target-morphism = "Cooperate4"/>
        <MORPHSIM-MAPPING source-morphism = "Cooperate5"
                                    target-morphism = "Cooperate5"/>
        <MORPHSIM-MAPPING source-morphism = "Cooperate6"
                                    target-morphism = "Cooperate6"/>
    </MORPHISM-MAPPING>
</FUNCTOR>
```

Figure 311: XML Specification of Functor *RAC-Self-Configuration-Take-over-RAOL*

```
<NATURAL-TRANSFORMATION name = "RAC-Evolution-Self-Configuration">
    <ARROW>
        <ARROW name = "Relation1"/>
            <FROM-FUNCTOR name = "RAC-Self-Configuration-Restart"
                                type = "RAC-Evolution-Self-Configuration"/>
            <TO-FUNCTOR name = "RAC-Self-Configuration-Substitute"
                              type = "RAC-Evolution-Self-Configuration"/>
        </ARROW>
        <ARROW name = "Relation2"/>
            <FROM-FUNCTOR name = "RAC-Self-Configuration-Restart"
                                type = "RAC-Evolution-Self-Configuration"/>
            <TO-FUNCTOR name = "RAC-Self-Configuration-Take-Over"
                              type = "RAC-Evolution-Self-Configuration"/>
        </ARROW>
        <ARROW name = "Relation3"/>
            <FROM-FUNCTOR name = "RAC-Self-Configuration-Substitute"
                                type = "RAC-Evolution-Self-Configuration"/>
            <TO-FUNCTOR name = "RAC-Self-Configuration-Take-Over"
                              type = "RAC-Evolution-Self-Configuration"/>
        </ARROW>
```

```
    </ARROW>
</NATURAL-TRANSFORMATION>
```

Figure 312: XML Specification of Natural Transformation *RAC-Self-Configuration*

```
<CATEGORY name = "Relation-Set-of-RAC-Evolution-Self-Configuration">
    <OBJECT>
        <OBJECT name = "RAC-Self-Configuration-Restart"
                    type = "RAC-Evolution-Self-Configuration" />
        <OBJECT name = "RAC-Self-Configuration-Substitute"
                    type = "RAC-Evolution-Self-Configuration"/>
        <OBJECT name = "RAC-Self-Configuration-Take-Over"
                    type = "RAC-Evolution-Self-Configuration"/>
    </OBJECT>
    <MORPHISM>
        <MORPHISM name = "Relation1"
                        type ="RAC-Evolution-Relation-Self-Configuration"/>
            <FROM-OBJECT name = "RAC-Self-Configuration-Restart"
                            type = "RAC-Evolution-Self-Configuration" />
            <TO-OBJECT name = "RAC-Self-Configuration-Substitute"
                        type = "RAC-Evolution-Self-Configuration"/>
        </MORPHISM>
        <MORPHISM name = "Relation2"
                        type ="RAC-Evolution-Relation-Self-Configuration"/>
            <FROM-OBJECT name = "RAC-Self-Configuration-Restart"
                            type = "RAC-Evolution-Self-Configuration" />
            <TO-OBJECT name = "RAC-Self-Configuration-Take-Over"
                        type = "RAC-Evolution-Self-Configuration"/>
        </MORPHISM>
        <MORPHISM name = "Relation3"
                        type ="RAC-Evolution-Relation-Self-Configuration"/>
            <FROM-OBJECT name = "RAC-Self-Configuration-Substitute"
                            type = "RAC-Evolution-Self-Configuration" />
            <TO-OBJECT name = "RAC-Self-Configuration-Take-Over"
                        type = "RAC-Evolution-Self-Configuration"/>
        </MORPHISM>
    </MORPHISM>
</CATEGORY>
```

Figure 313: XML Specification of Functor Category **RAC-Self-Configuration**

```
<CATEGORY name = "RAC-Configuration">
    <OBJECT>
        <OBJECT name = "RAO_i" type = "RAO-Type_i"/>
        <OBJECT name = "RAO_j" type = "RAO-Type_j"/>
    </OBJECT>
    <MORPHISM>
        <MORPHISM name = "Connection_m" type ="Connection-Type_m"/>
            <FROM-OBJECT name = "RAO_i" type = "RAO-Type_i"/>
            <TO-OBJECT name = "RAO_j" type = "RAO-Type_j"/>
        <MORPHISM>
    </MORPHISM>
</CATEGORY>
```

Figure 314: XML Specification of Category **RAC-Configuration**

```
<CATEGORY name = "RACG-Configuration">
    <OBJECT>
        <OBJECT name = "RAC_i" type = "RAC-Type_i"/>
        <OBJECT name = "RAC_j" type = "RAC-Type_j"/>
    </OBJECT>
    <MORPHISM>
        <MORPHISM name = "Connection_m" type ="Connection-Type_m"/>
            <FROM-OBJECT name = "RAC_i" type = "RAC-Type_i"/>
            <TO-OBJECT name = "RAC_j" type = "RAC-Type_j"/>
        <MORPHISM>
    </MORPHISM>
</CATEGORY>
```

Figure 315: XML Specification of Category **RACG-Configuration**

Appendix F: Screen Shots of RASFIT

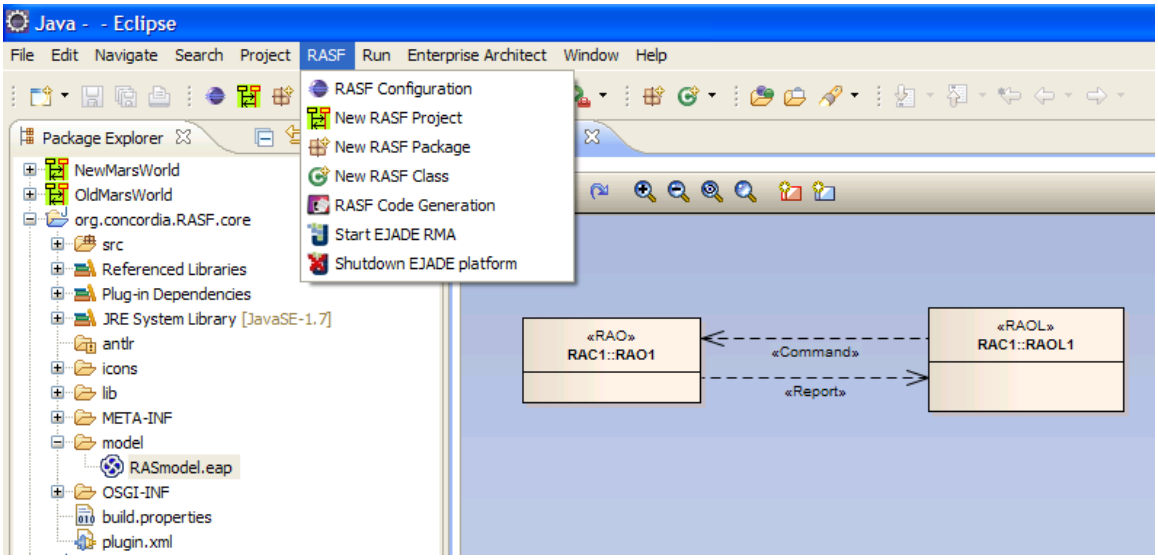


Figure 316: Menu Area for RASFIT

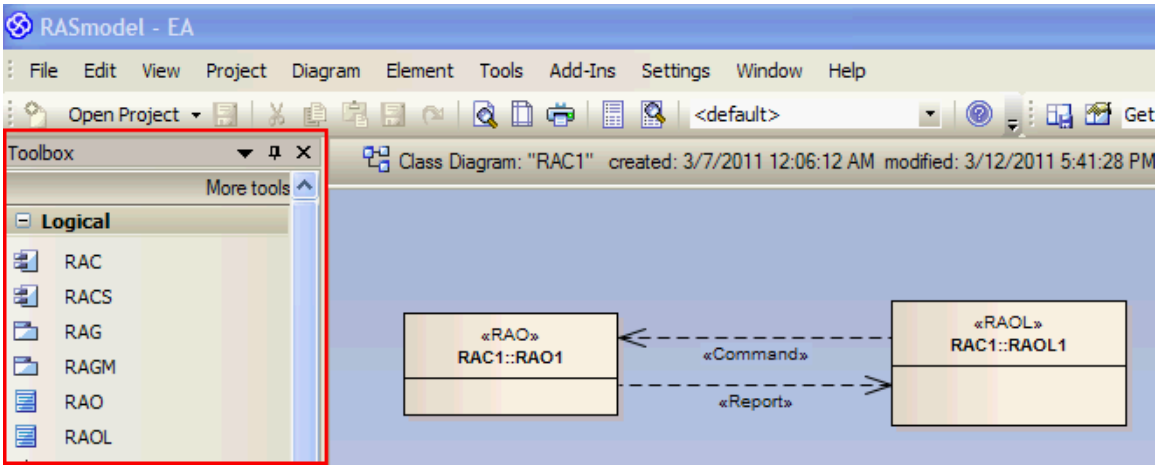


Figure 317: Toolbox Area for Drawing UML Diagrams in RASFIT

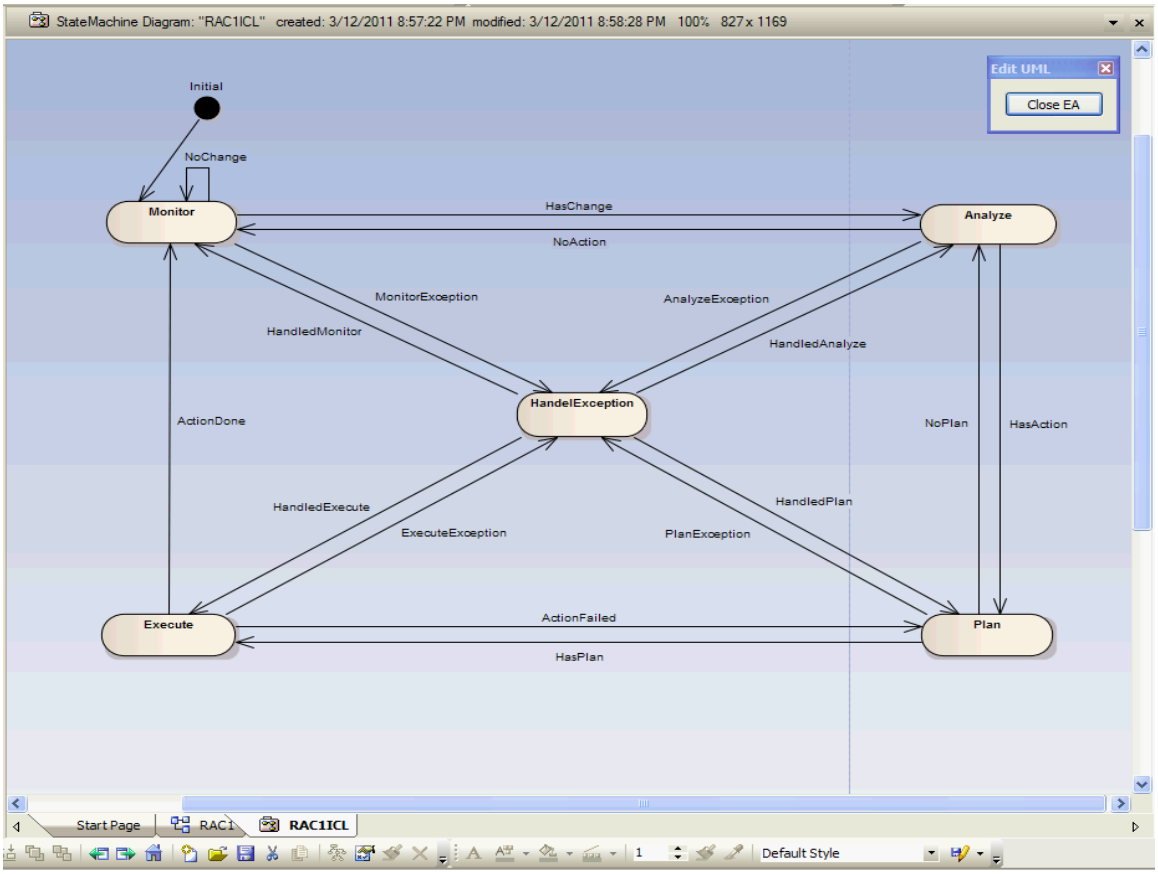


Figure 318: Canvas for Drawing UML Diagrams in RASFIT

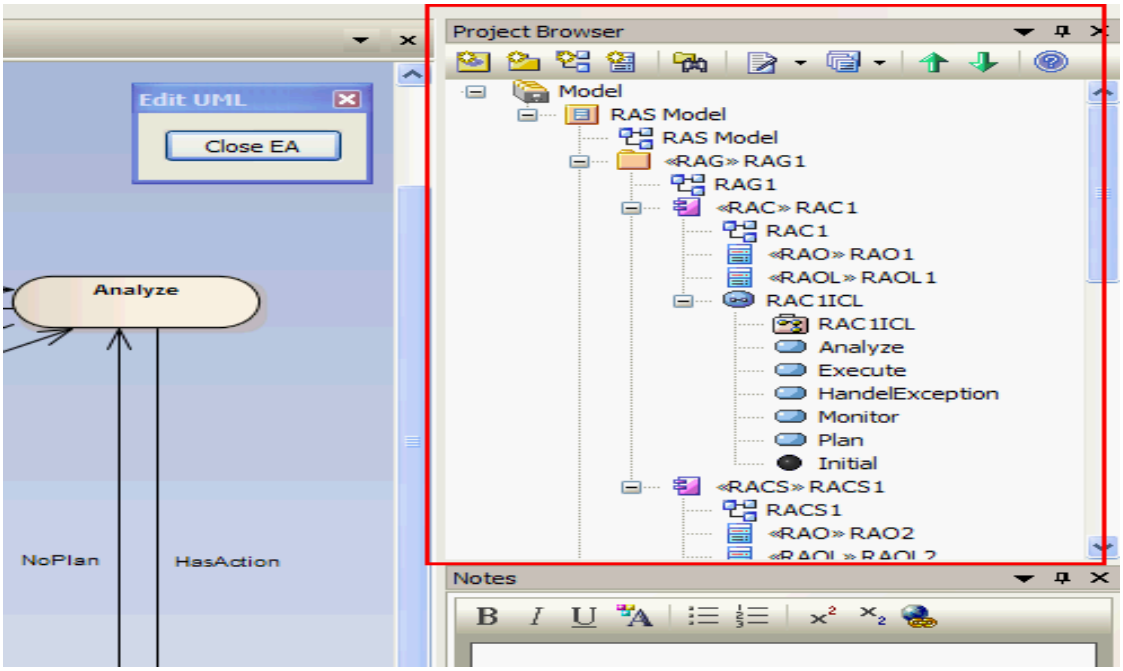
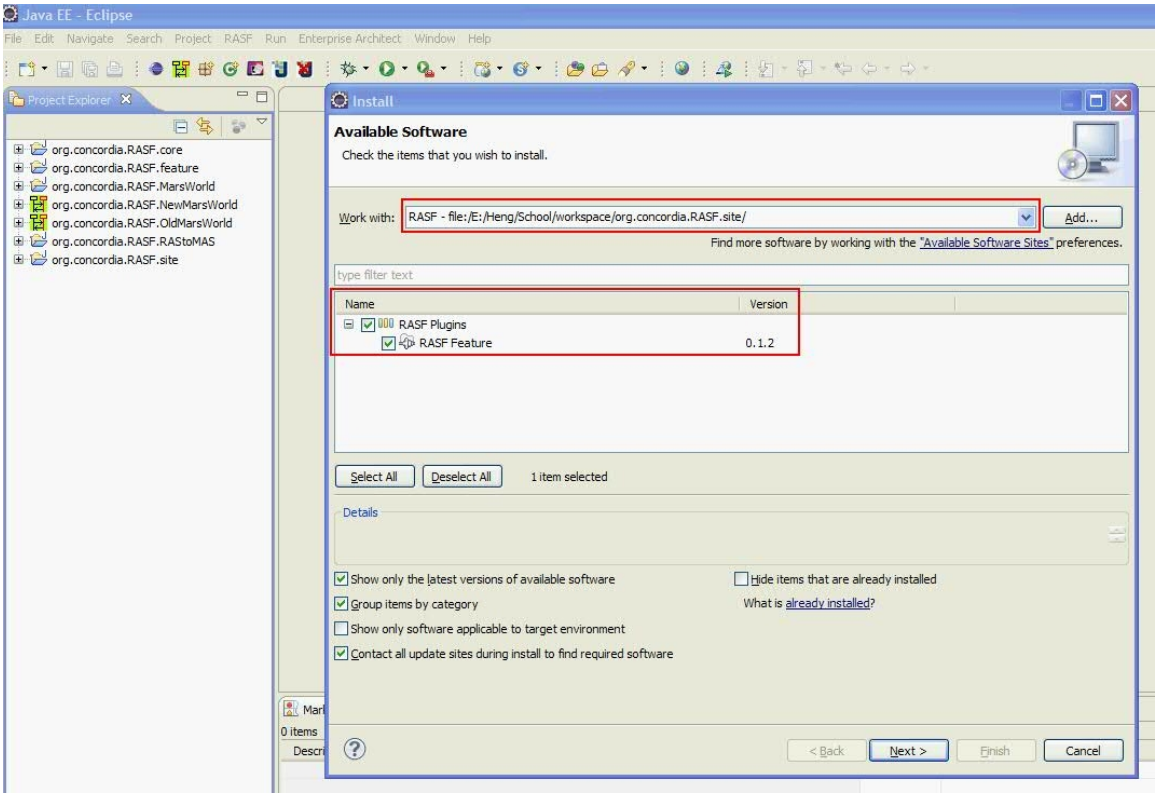
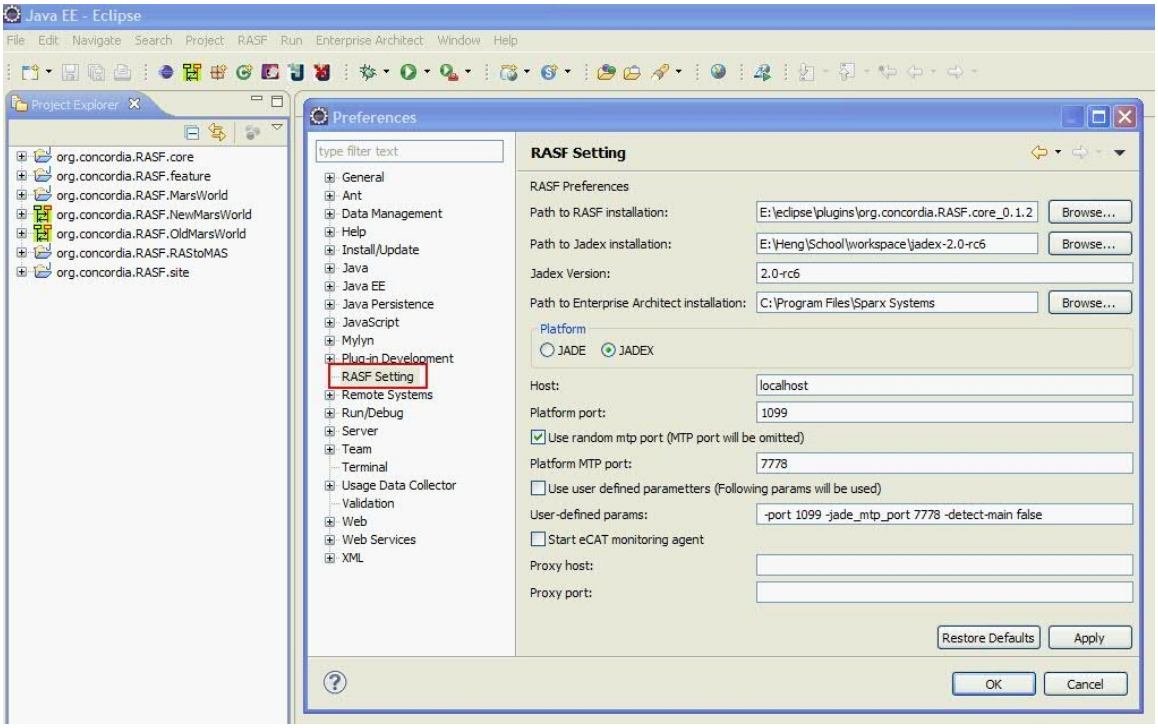


Figure 319: Project Browser for Drawing UML Diagrams in RASFIT

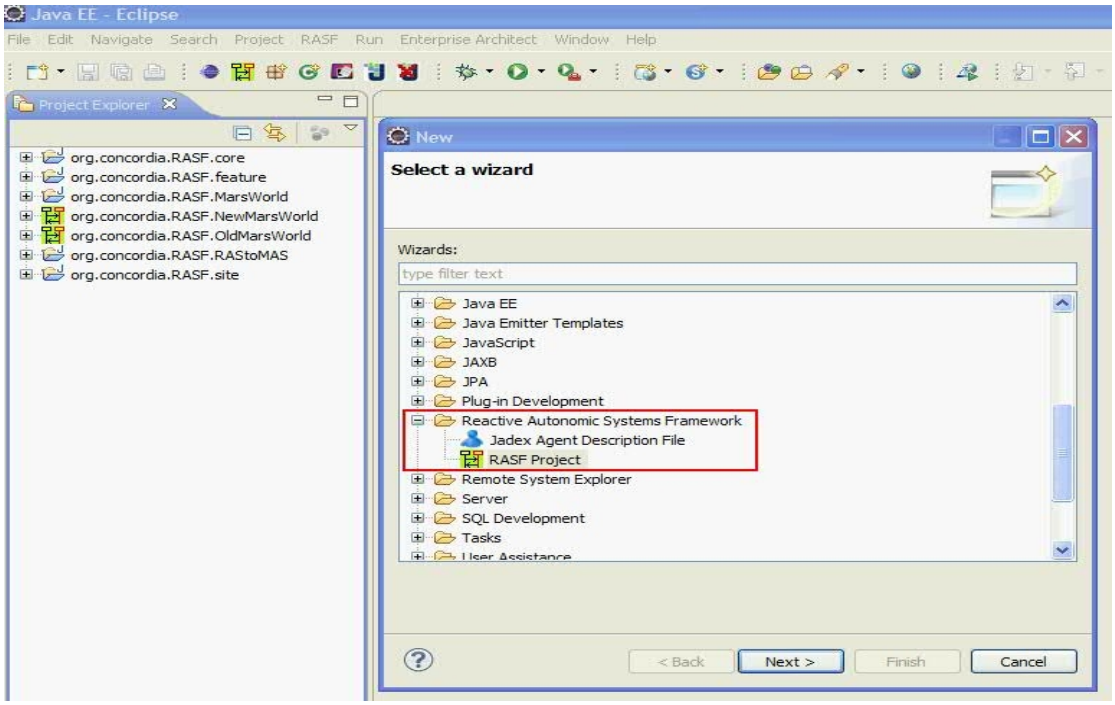
Appendix G: Installation and Configuration of RASFIT



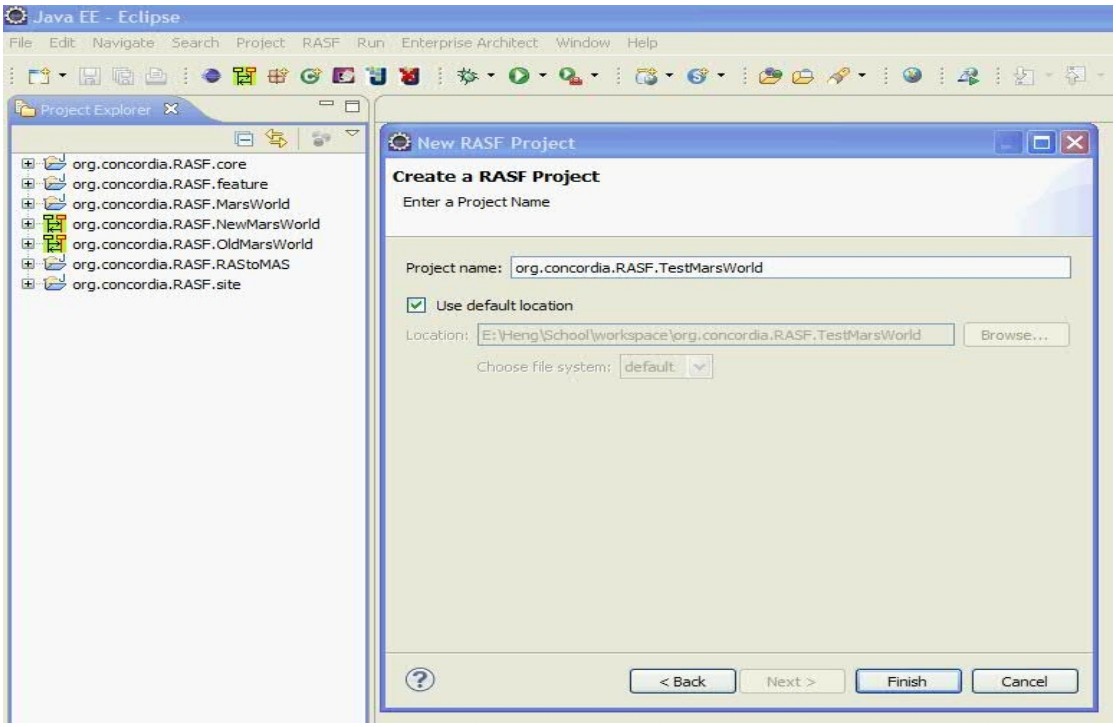


Appendix H: Applying RASF Methodology with RASFIT

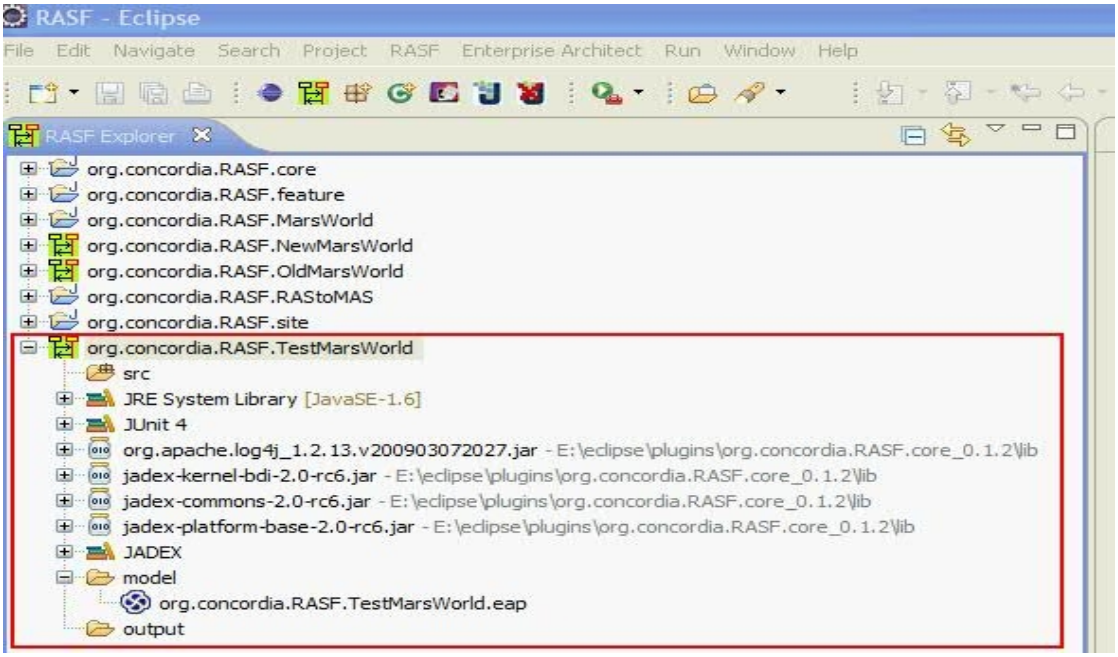
Step 2 (Phase 1): Select “RASF Project” from the new project wizard and click “Next >”.



Step 3 (Phase 1): Enter the project name and click “Finish” (see the figure below).

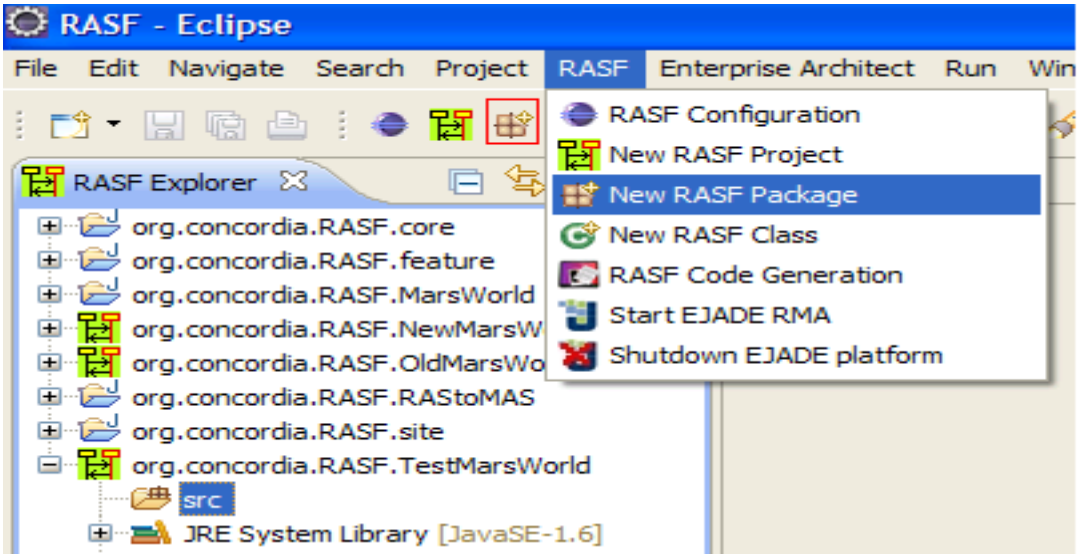


Step 4a (Phase 1): A RASF project and related model files, jars, libraries, folders are created.

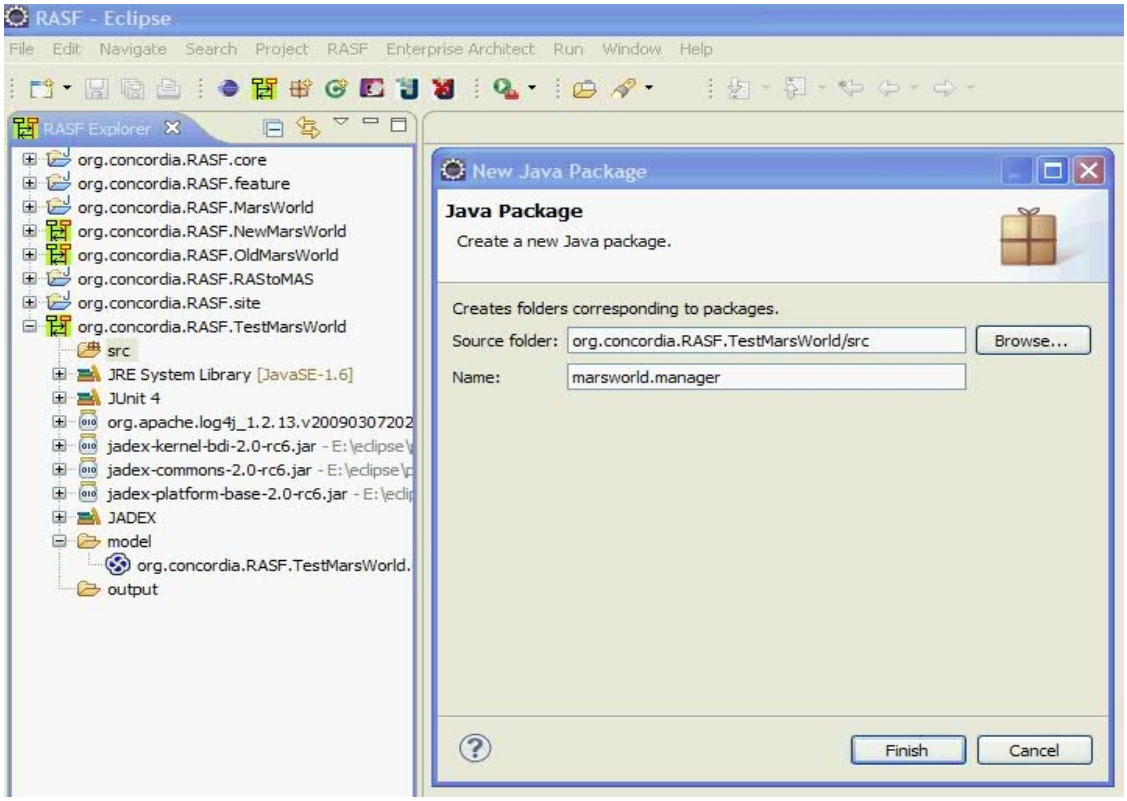


Step 4b (Phase 1): Alternatively, a RASF project can be created by selecting "New RASF Project" or clicking the button highlighted by the red rectangle in the figure below.

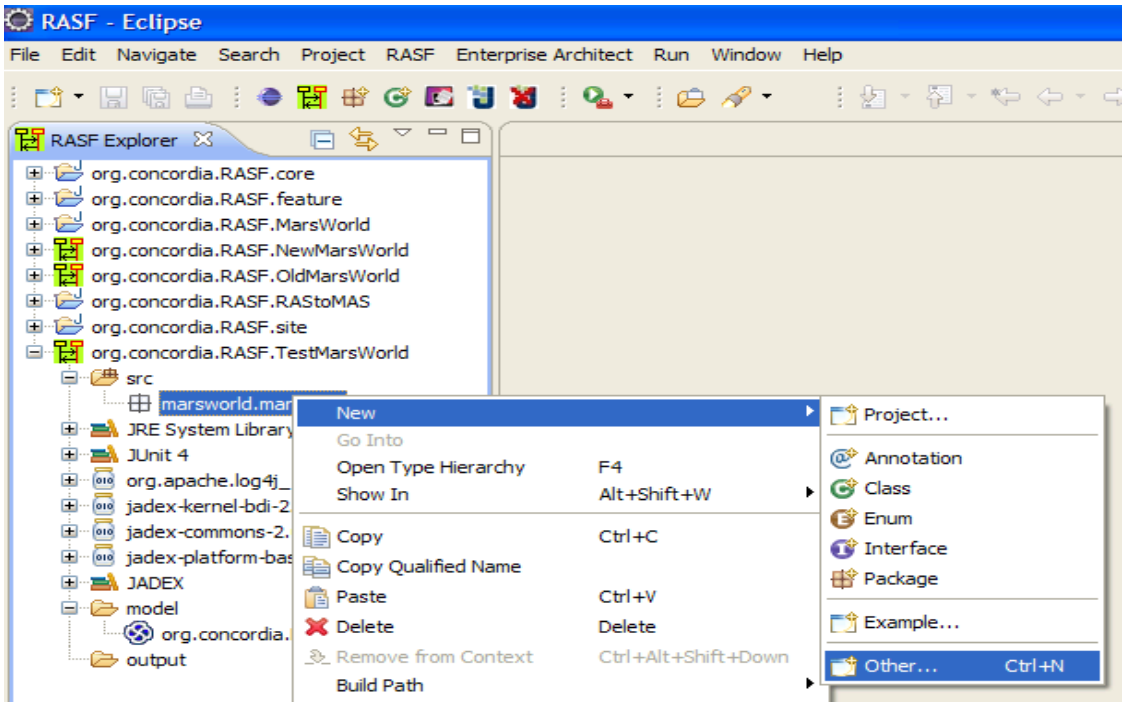
Step 5 (Phase 1): Create a new package by selecting "New RASF Package" or clicking the button highlighted by the red rectangle in the figure below.



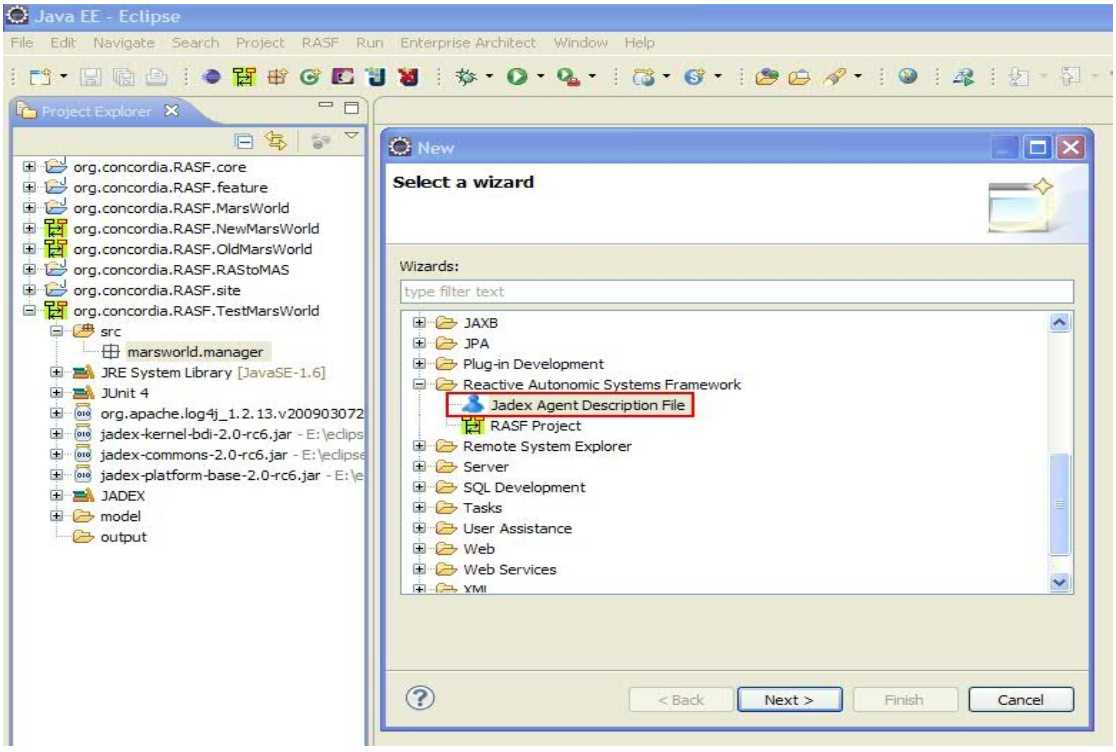
Step 6 (Phase 1): Enter the source folder as well as package name and click "Finish" (see the figure below).



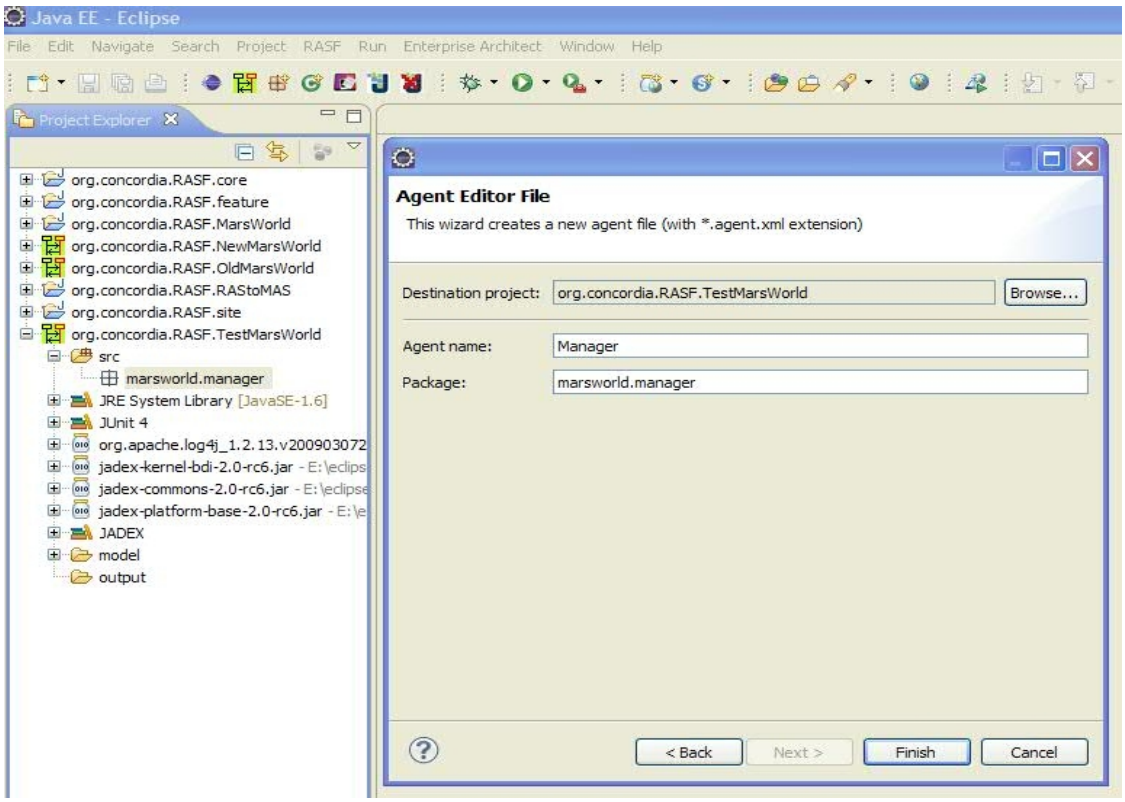
Step 7 (Phase 3): Create an agent description file by selecting "New" → "Other…" on the package.



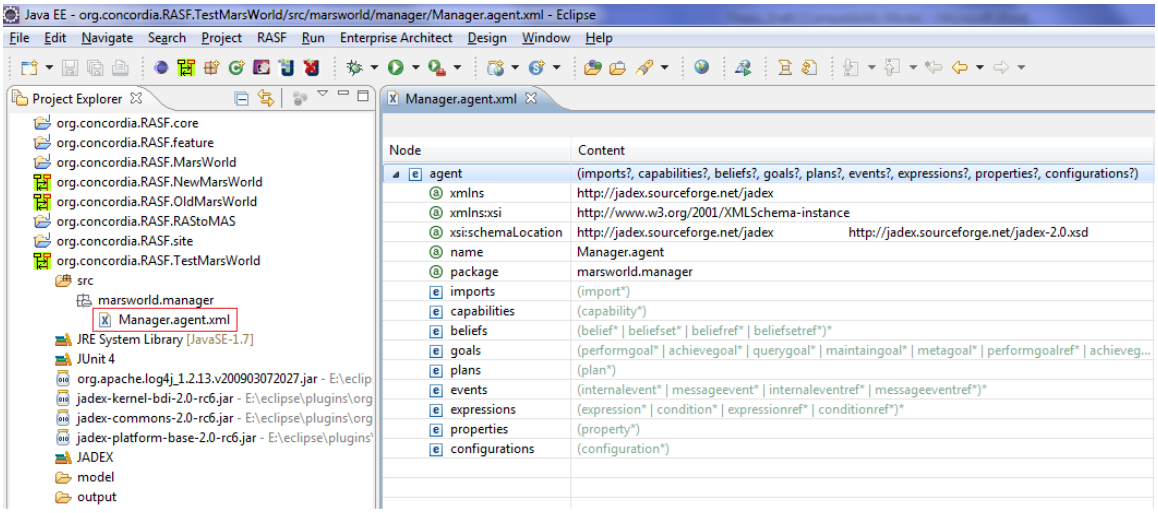
Step 8 (Phase 3): Select "Jadex Agent Description File" and click "Next >".



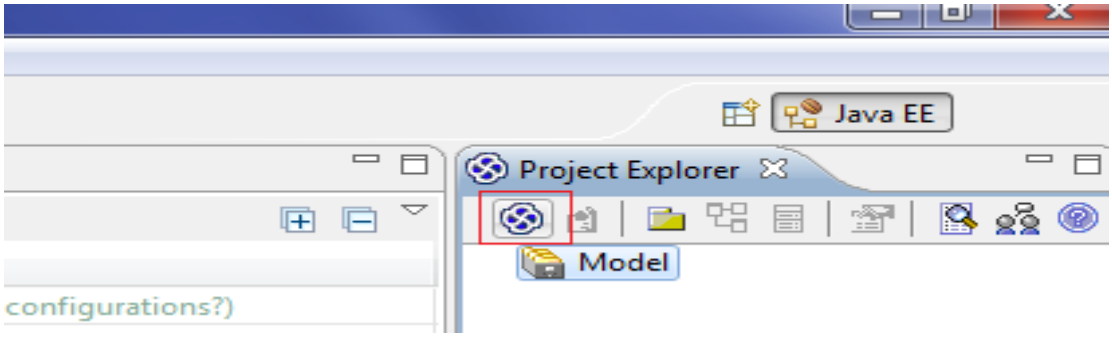
Step 9 (Phase 3): Enter the destination project, agent name, package and click "Finish".



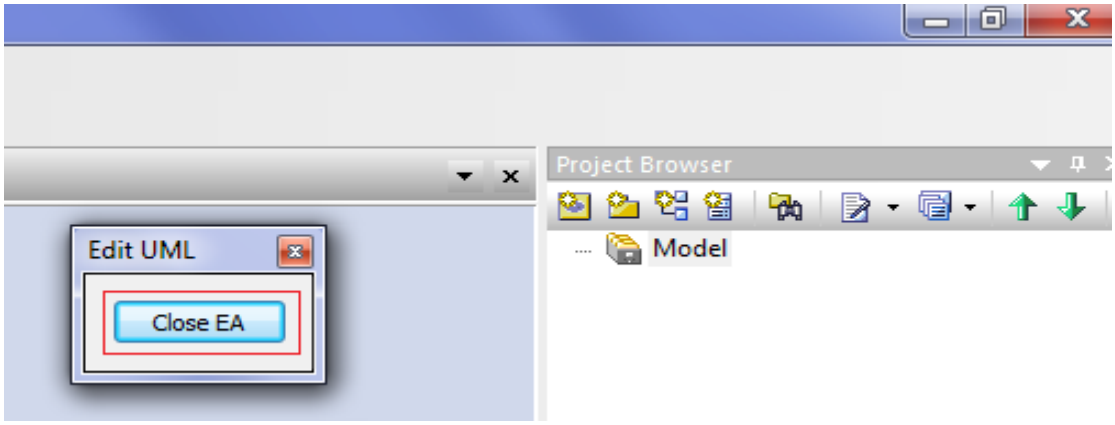
Step 10 (Phase 3): After successfully creating the agent description file, it can be edited in the editor view as the following:



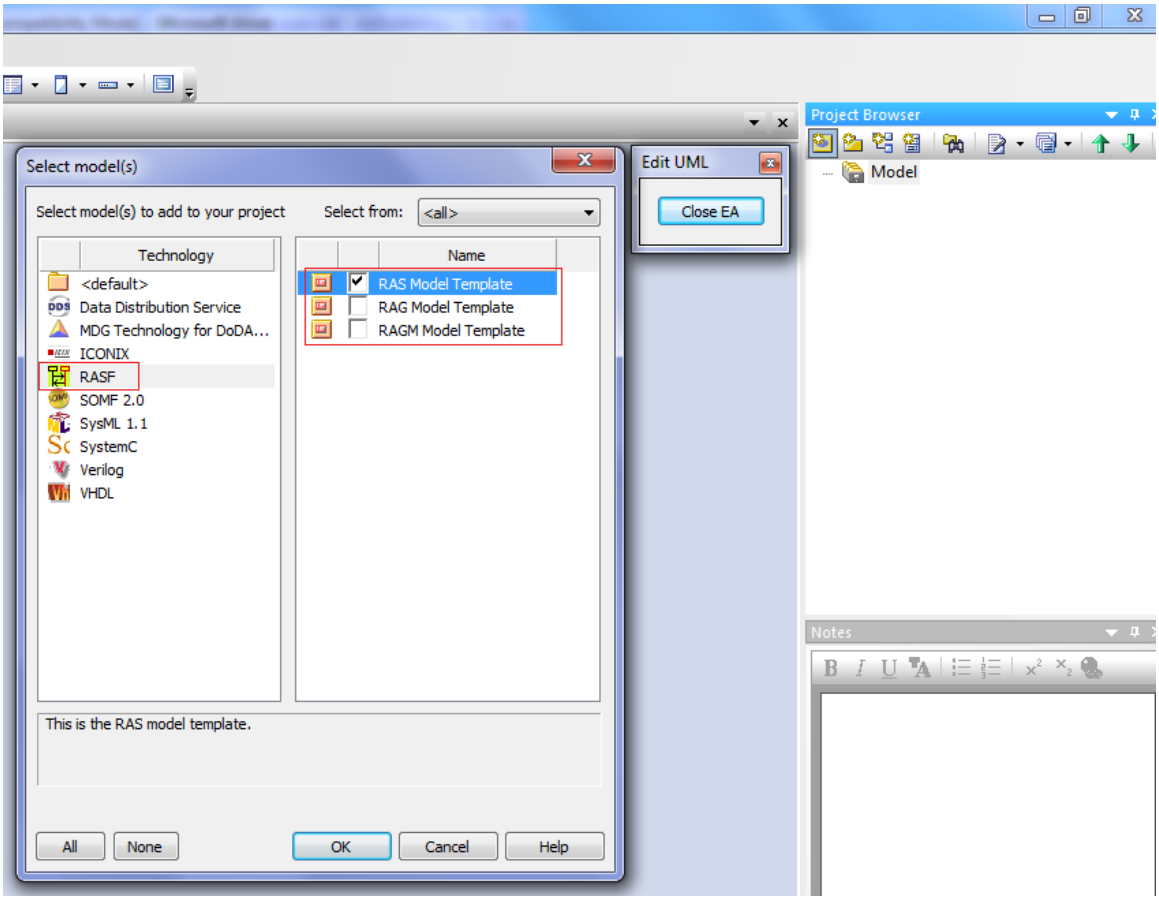
Step 12 (Phase 1): Click the button "Open in Enterprise Architect" to switch from the Eclipse IDE to the EA IDE in the project explorer of EA (see the figure below).



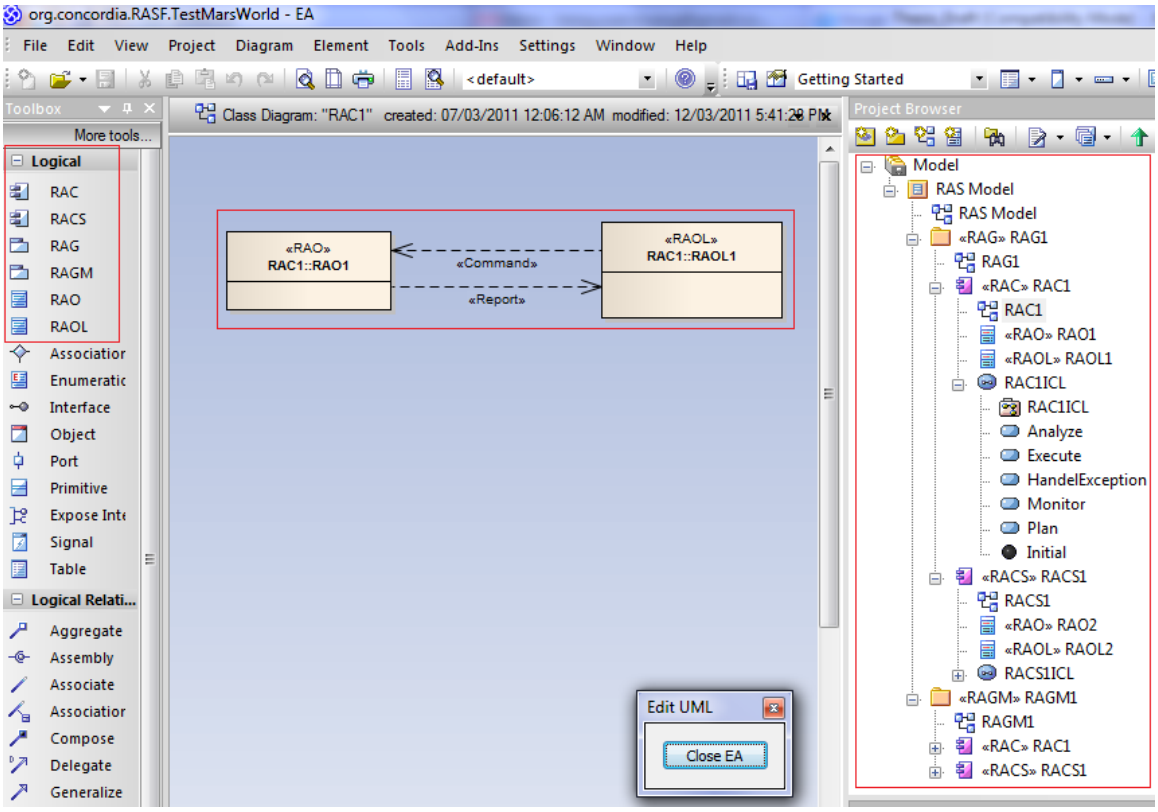
Step 13a (Phase 1): We can start modeling through the toolbar in the project browser depicted below, such as "Add a Package", "New Diagram", "Create Element" and "Import Source Directory". The EA IDE can be switched back to the Eclipse IDE by clicking the button "Close EA".



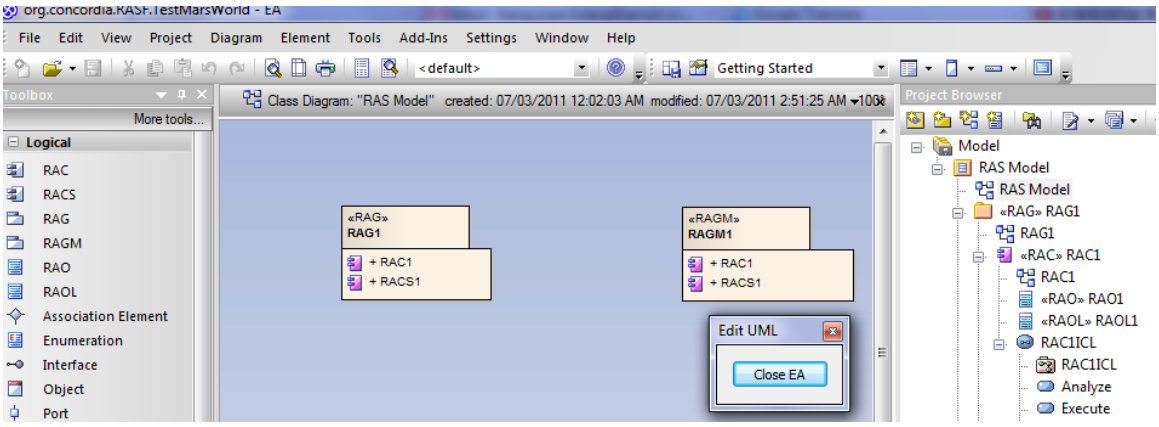
Step 13b (Phase 1): Alternatively, we can import the predefined model templates and patterns by clicking the button "New Model from Pattern" --> selecting "RASF" from the popup dialogue "Select model(s)" --> checking the corresponding model template (see the figure below).

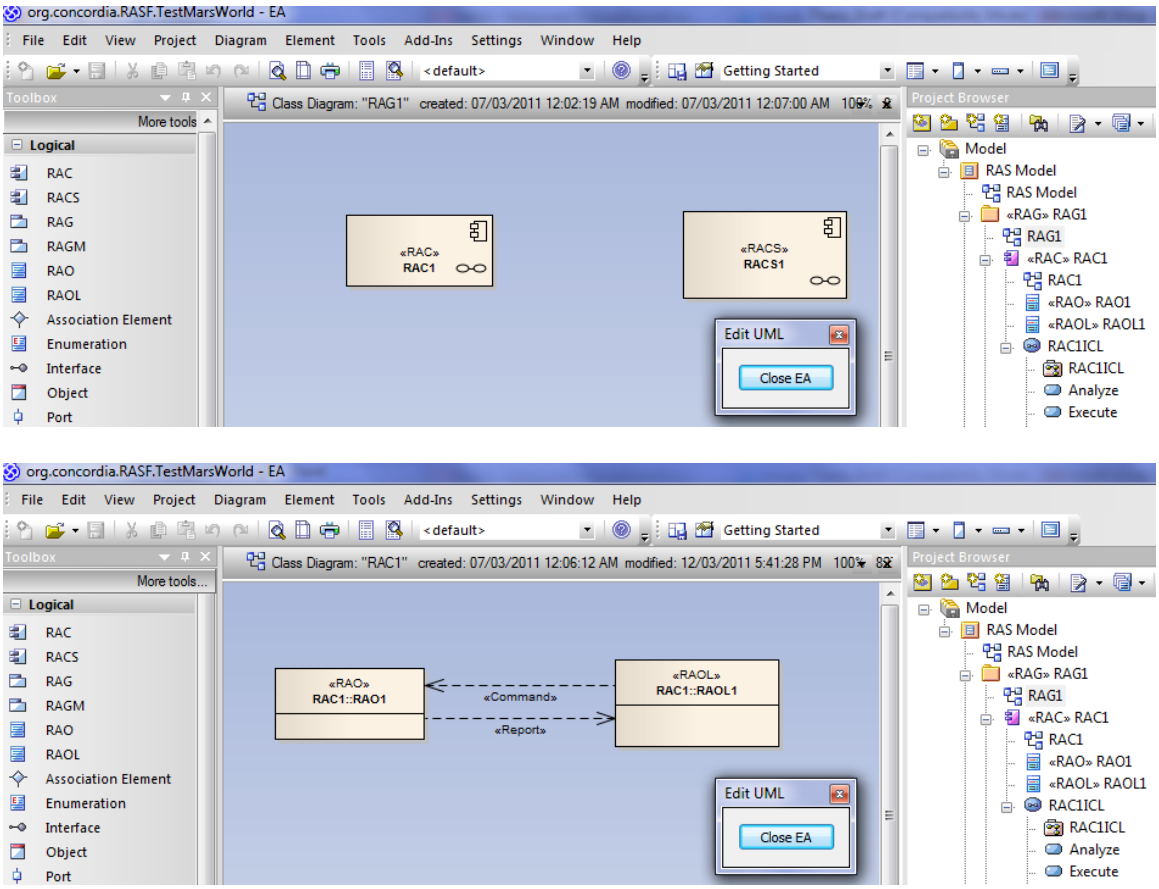


Step 14 (Phase 1): After successfully importing selected model template, the predefined packages and diagrams are created and can be expanded in the view of "Project Browser", such as the package "RAS Model", package diagram "RAS Model", component diagram "RAG1", class diagram "RAC1" and state diagram "RAC1ICL". We can modify those diagrams in the canvas area and add new RAE by dragging them from the toolbox, such as RAO, RAC and RAG (see the figure below).

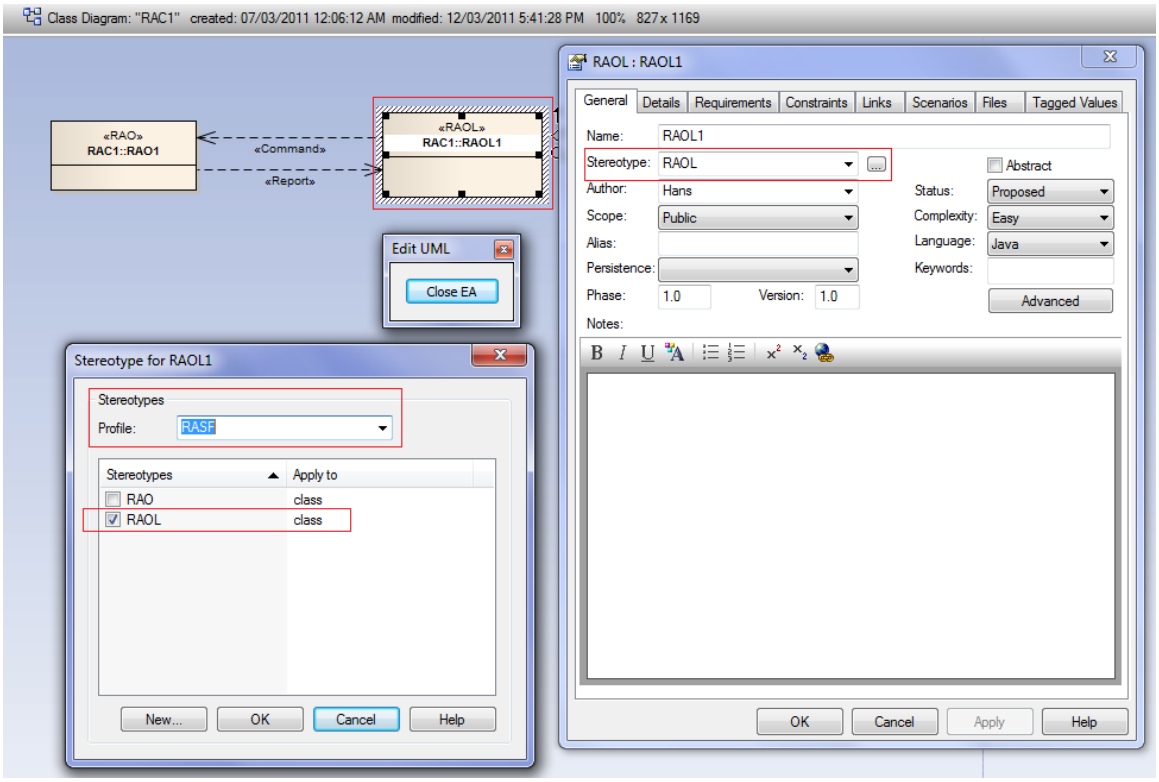


Step 15 (Phase 1): The diagrams mentioned above can be navigated either from the project browser or by double clicking the related RAE on the canvas. For example, we can double click the "RAG1" to navigate from the package diagram "RAS Model" to the component diagram "RAG1" and to the class diagram "RAC1" by double clicking the "RAC1" illustrated in the figures below.

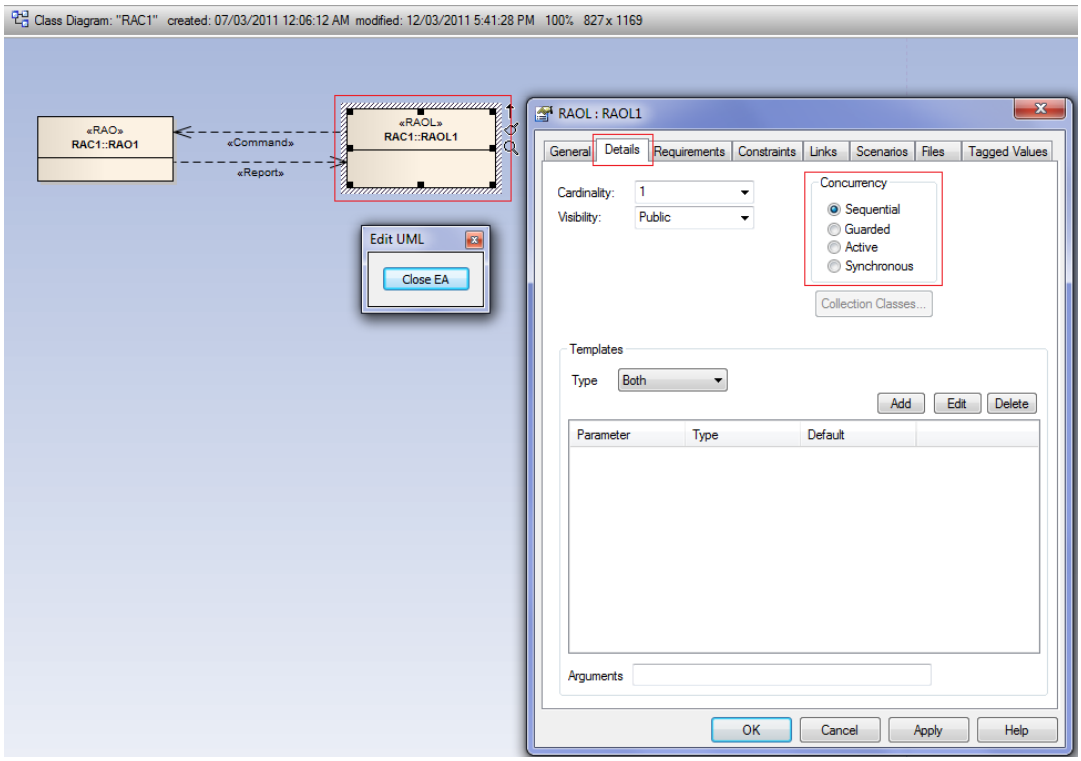




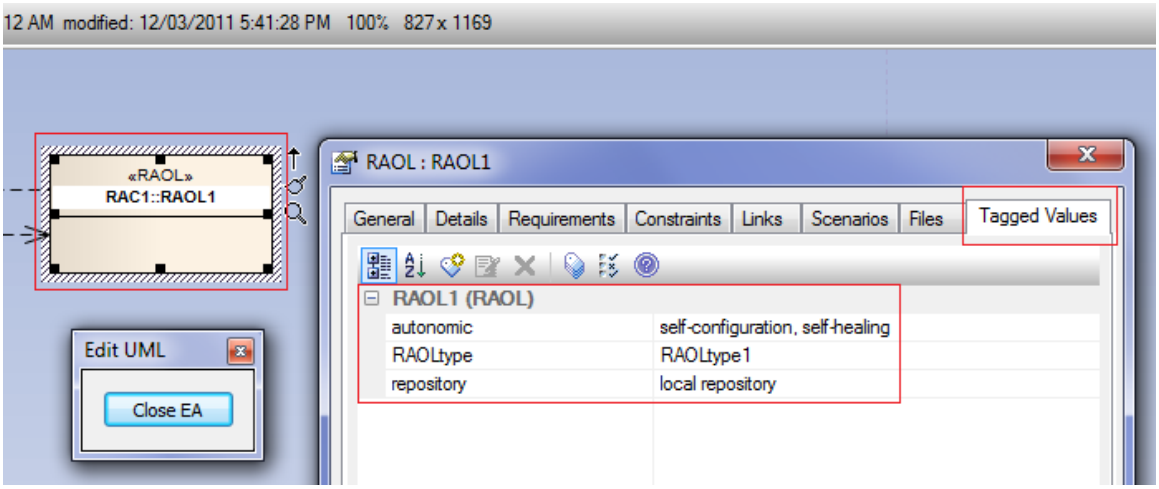
Step 16 (Phase 1): The properties of RAO and RAOL on the figure above can be setup by double clicking them. The figure below depicts how to configure some general properties of RAOL, such as name (RAOL1), scope (Public) and programming language (Java, C++, PHP. etc.). The stereotype of RAOL1 is set to <<RAOL>> by checking "RAOL" from the profile "RASF" on the popup dialogue of the selecting stereotypes. End users can only choose from <<RAO>> and <<RAOL>> that are applied to the RASF class diagram (RAO level modeling) and constrained in the RASF modeling profile.



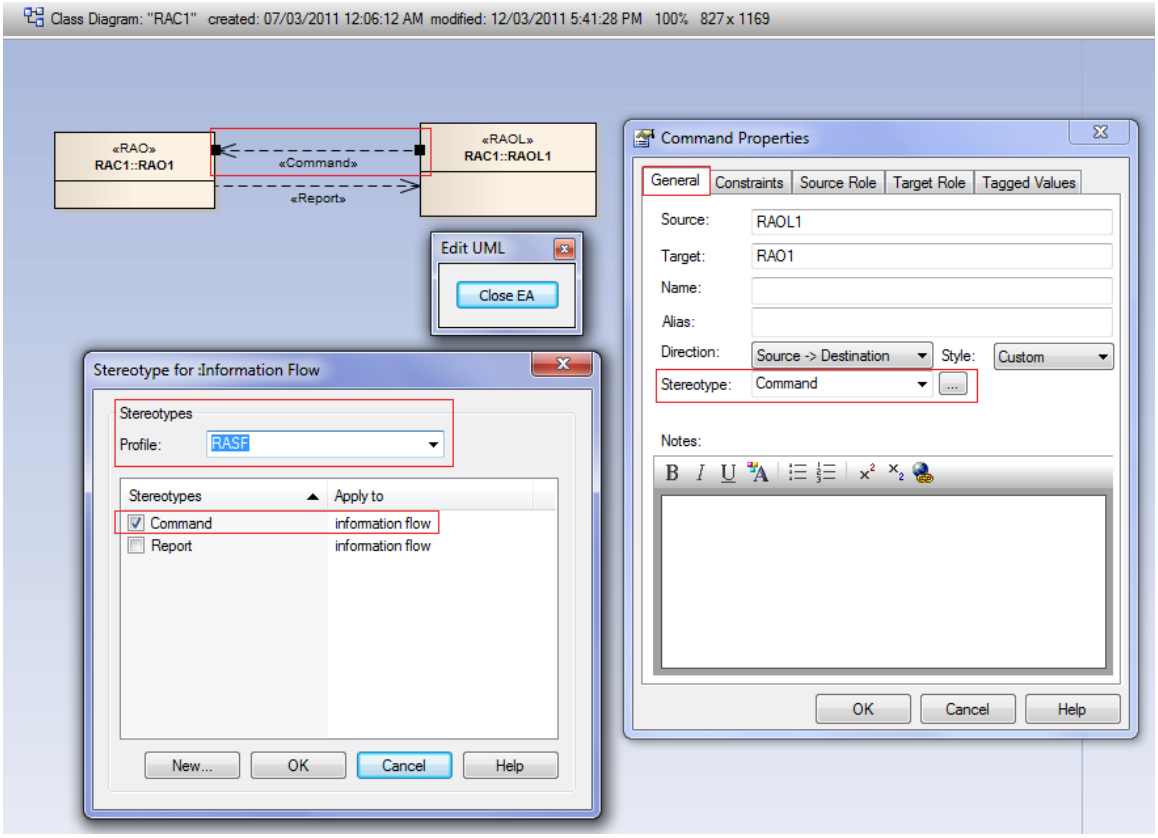
Step 17 (Phase 1): From the tab "Details", we can set the cardinality, visibility and concurrency (synchronous or asynchronous) of each RAE (see the figure below).



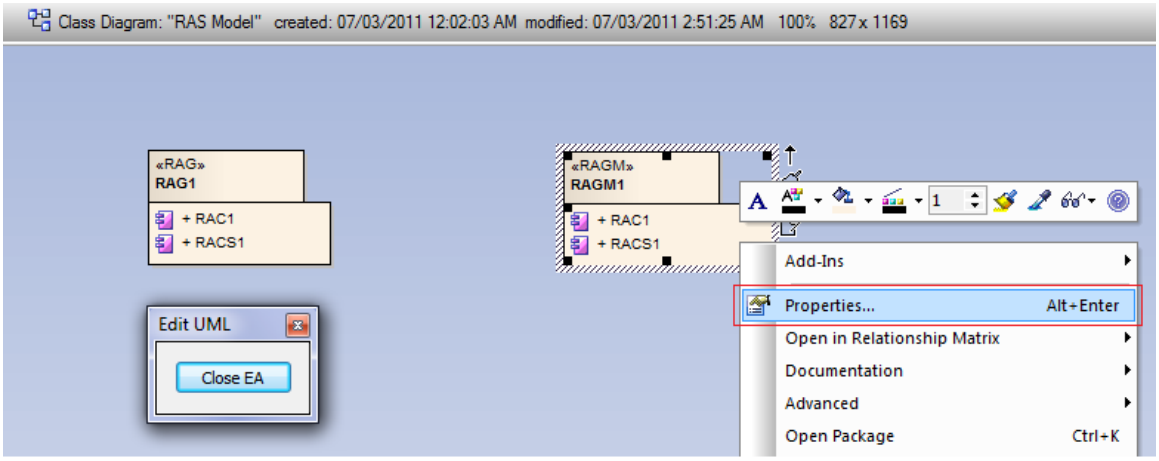
Step 18 (Phase 1): The autonomic properties such as self-configuration and self-healing, RAOLtype (RAOL- type1) and repository (local repository) of RAOL1 can be configured through the tab "Tagged Values" showed below.



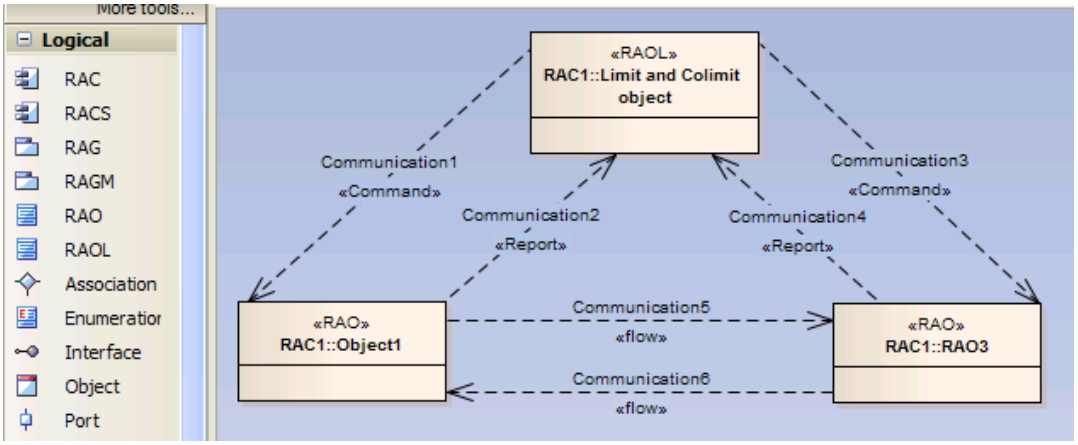
Step 19 (Phase 1): The properties of interactions between RAO and RAOL can be setup by double click them. The figure below depicts how to configure some general properties of the interaction from RAOL to RAO, such as source object (RAOL1), target object (RAO1) and direction (Source -> Destination). The stereotype of that interaction is set to <<Command>> by checking "Command" from profile "RASF" on the popup dialogue of selecting stereotypes. We can only choose from <<Report>> as well as <<Command>> that are applied to the RASF class diagram and constrained in the RASF modeling profile.

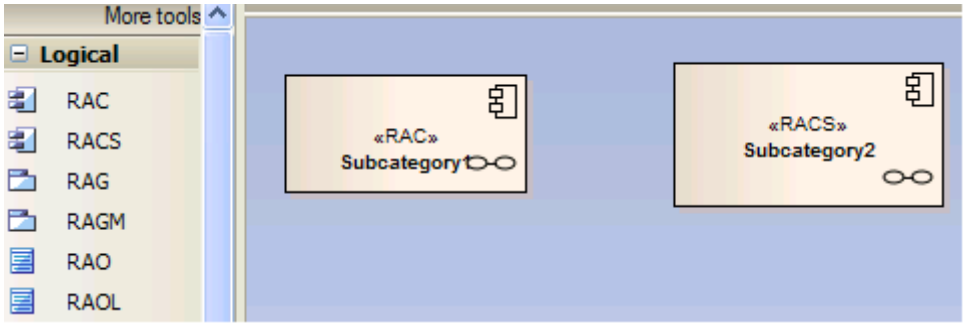


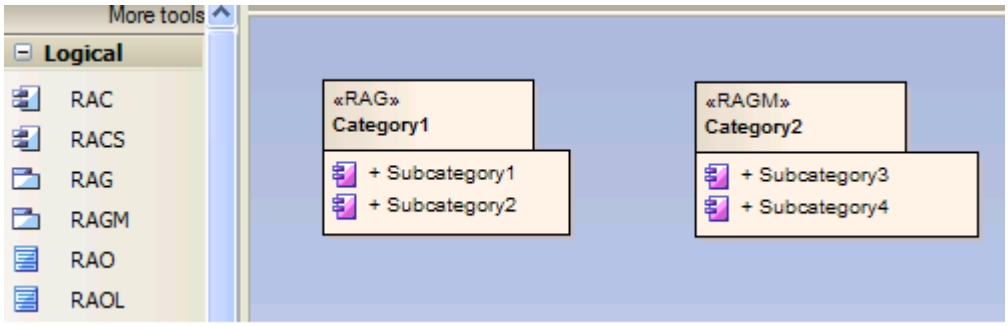
Step 20 (Phase 1): Similarly, we can set other RAE's properties by right clicking them and selecting "Properties..." as the figure below illustrates.

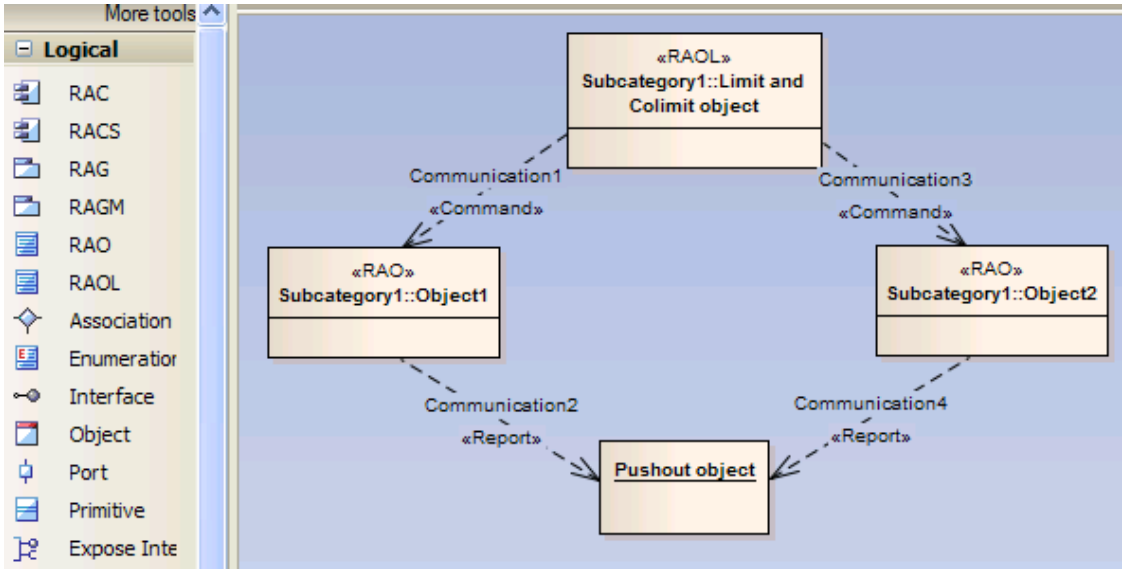


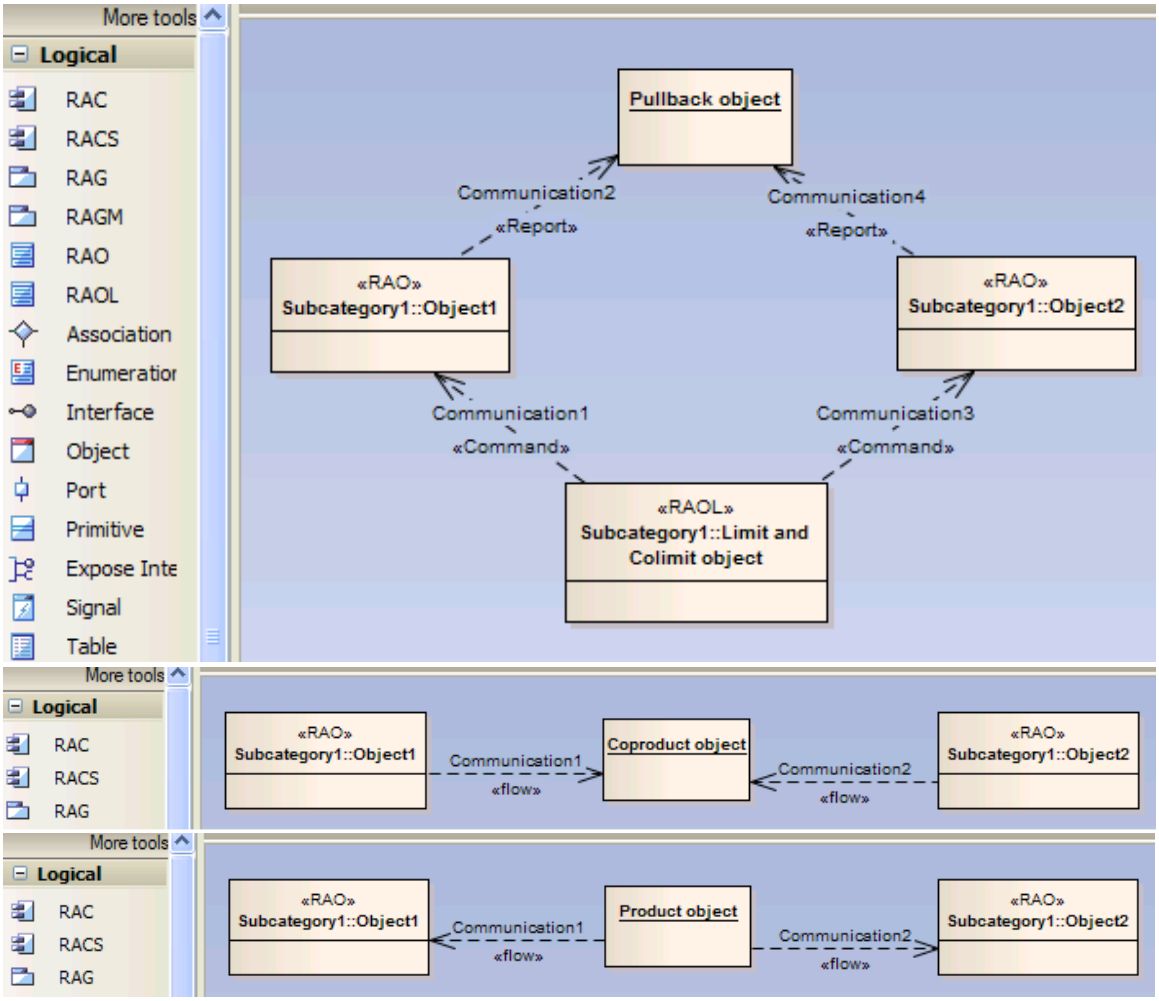
Step 21 (Phase 2): After having specified and drawn the RAS model with corresponding structures, behavior as well as self-* properties, we can specify and draw the categorical RAS model as we described in Chapter 6 (see the figures below as some examples).


More tools...
Logical
RAC
RACS
RAG
RAGM
RAO
RAOL
Association
Enumeratior
Interface
Object
Port
«RAOL»
RAC1::Limit and Colimit object
Communication1
«Command»
Communication2
«Report»
Communication3
«Command»
Communication4
«Report»
«RAO»
RAC1::Object1
Communication5
«flow»
«RAO»
RAC1::RAO3
Communication6
«flow»

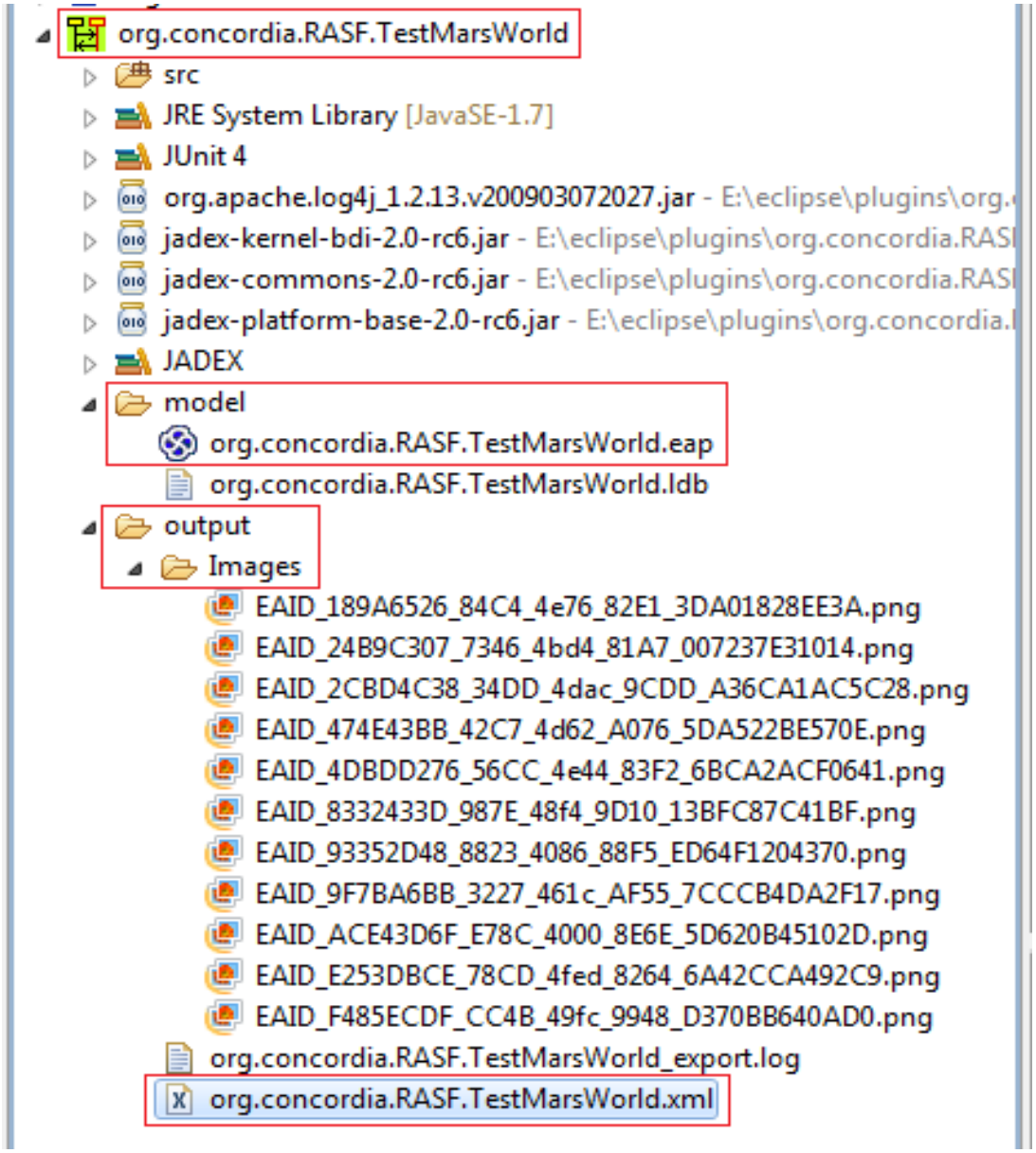

More tools
Logical
RAC
RACS
RAG
RAGM
RAO
RAOL
«RAC»
Subcategory1
«RACS»
Subcategory2


More tools
Logical
RAC
RACS
RAG
RAGM
RAO
RAOL
«RAG»
Category1
+ Subcategory1
+ Subcategory2
«RAGM»
Category2
+ Subcategory3
+ Subcategory4

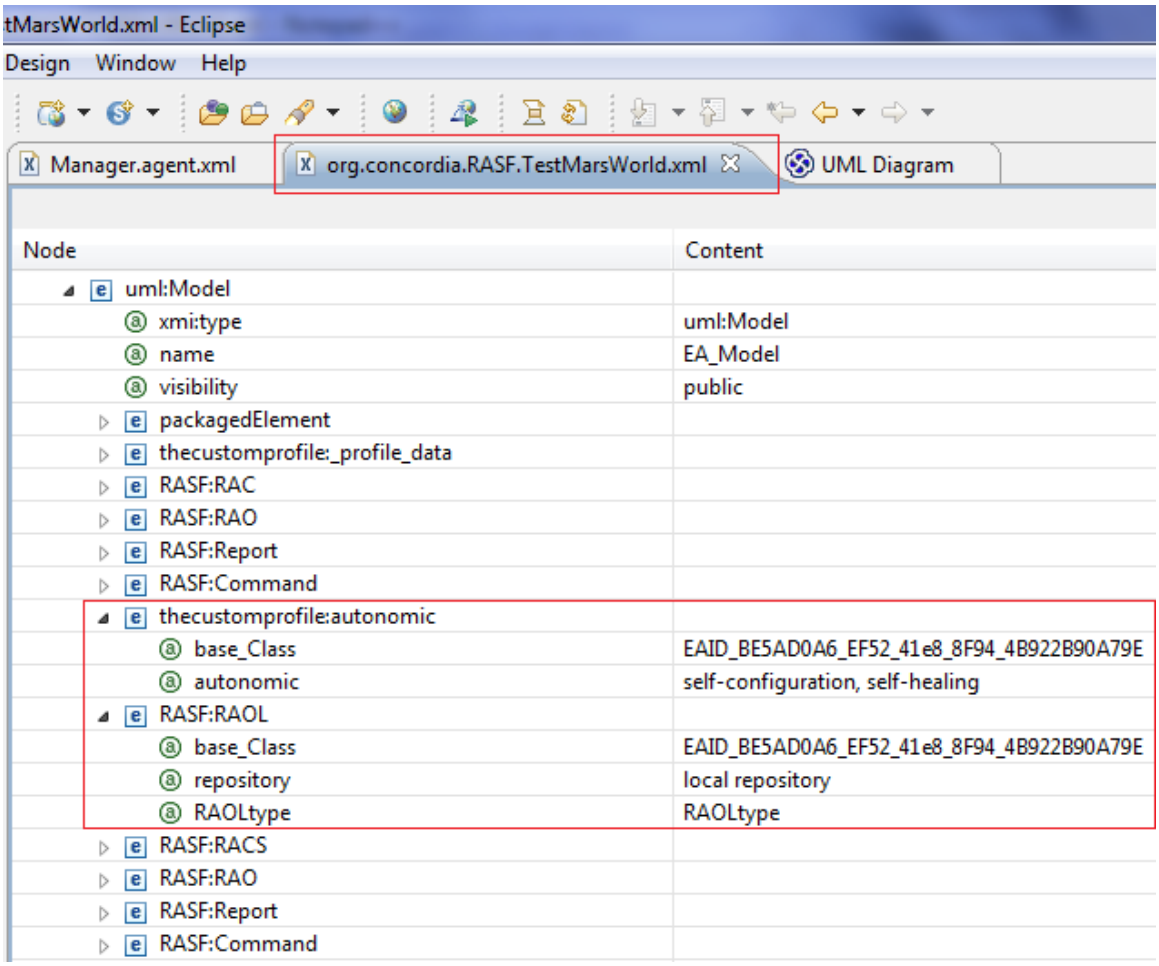

More tools
Logical
RAC
RACS
RAG
RAGM
RAO
RAOL
Association
Enumeratior
Interface
Object
Port
Primitive
Expose Inte
«RAOL»
Subcategory1::Limit and Colimit object
Communication1
«Command»
Communication3
«Command»
«RAO»
Subcategory1::Object1
«RAO»
Subcategory1::Object2
Communication2
«Report»
Communication4
«Report»
Pushout object



Step 23 (Phase 1 & 2): If the XML specification file is generated successfully, a new folder "output" is created under the selected project. There is a subfolder "Images" containing all the image files for each diagram created in the Section 9.3.2. Moreover, a XML file having the RAS specification with the project name is also generated under the project folder (see the figure below).



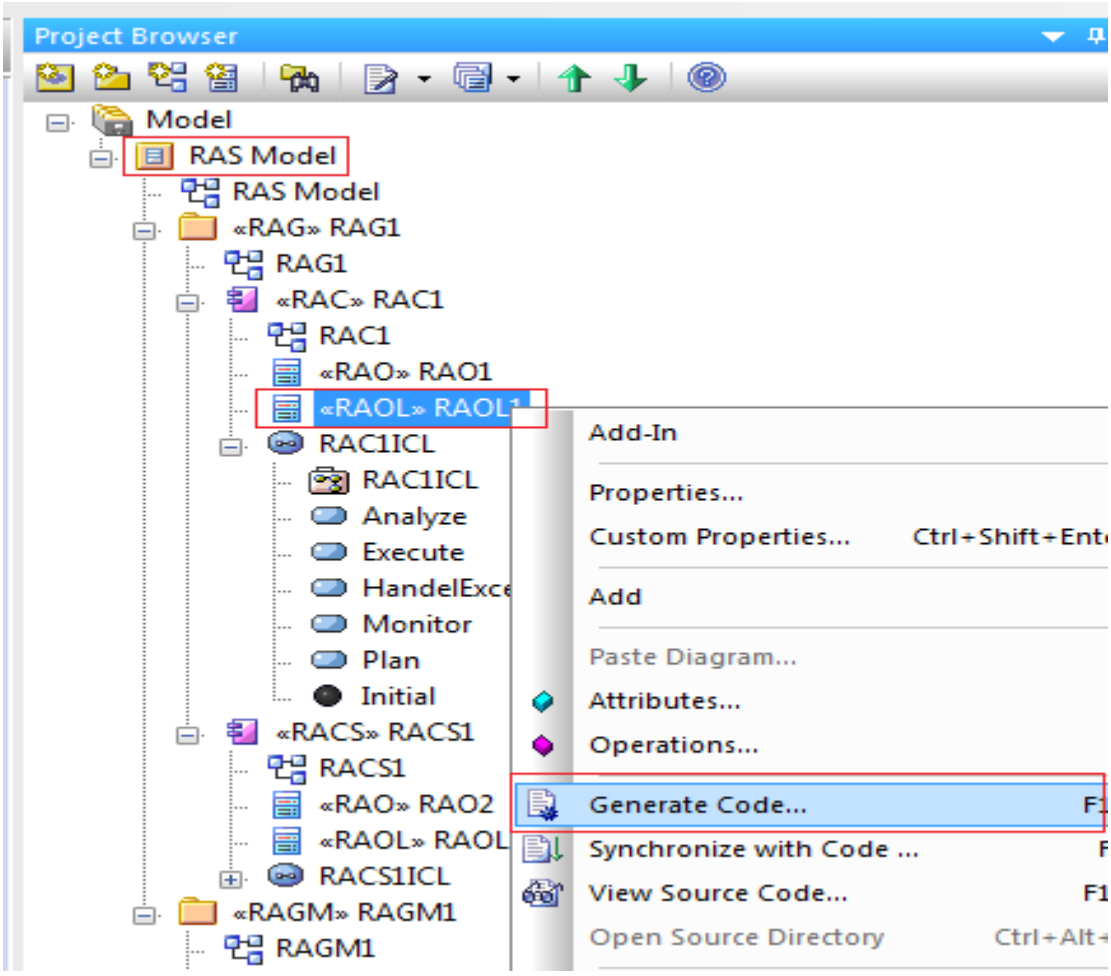
Step 24 (Phase 1 & 2): The following figures depict a perspective of XML specification file and a part of the file content with RAE property configuration.



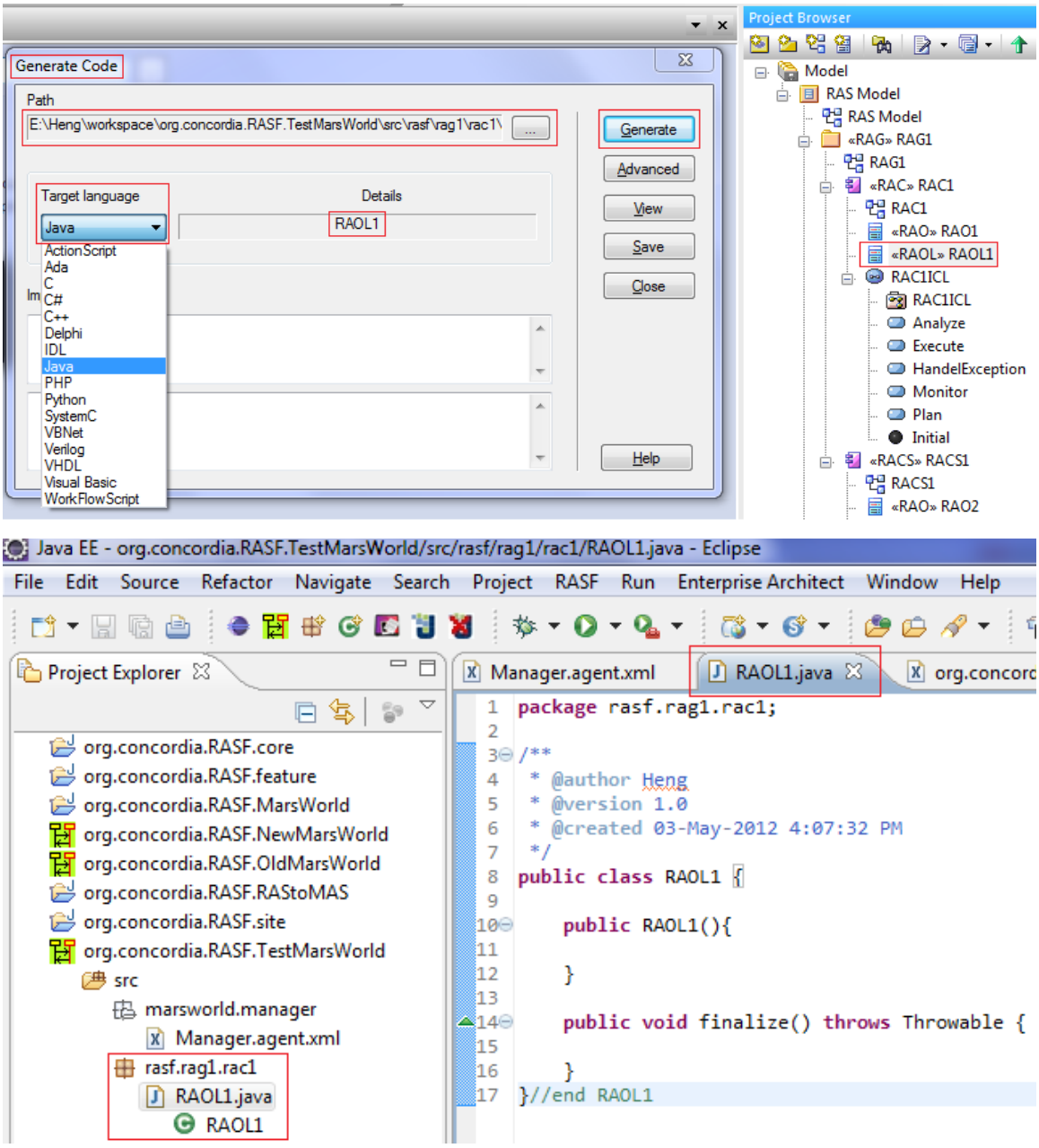

```
<transition xmi:type="uml:Transition" xmi:id="EAID_0BF72B4F_8DEE_4467_9A4E_2A829F5755DA" name="NoPlan" visibil
"EAID_1B93FAD5_24D6_4145_9A4A_9DFE3C222AA2" target="EAID_2A4D054F_9A1B_491d_A84D_910143785194"/>
<transition xmi:type="uml:Transition" xmi:id="EAID_E5946B98_0D22_46f9_B386_B93534747235" name="HasPlan" visibi
"EAID_1B93FAD5_24D6_4145_9A4A_9DFE3C222AA2" target="EAID_1D19A54B_7331_494e_8030_BFDF55A1BED4"/>
<transition xmi:type="uml:Transition" xmi:id="EAID_E86EC055_FBFC_4401_AD2D_AF044780E6CE" name="PlanException"
"EAID_1B93FAD5_24D6_4145_9A4A_9DFE3C222AA2" target="EAID_AB2B535A_7126_42de_9159_C1100B812098"/>
<subvertex xmi:type="uml:Pseudostate" xmi:id="EAID_5A444EF1_3566_4919_B2E2_A2D51C142AB6" name="Initial" visibi
    <outgoing xmi:idref="EAID_43E7CA37_F4F4_4ec7_83BE_E6CA9A27DF5E"/>
</subvertex>
<transition xmi:type="uml:Transition" xmi:id="EAID_43E7CA37_F4F4_4ec7_83BE_E6CA9A27DF5E" visibility="public" k
"EAID_5A444EF1_3566_4919_B2E2_A2D51C142AB6" target="EAID_4A223BA5_5F7C_4e91_94D5_74F7C4DADAEC"/>
region>
edClassifier>
Element>
lement xmi:type="uml:InformationFlow" xmi:id="EAID_065E7EF9_89D1_4f53_B151_4B2B880CD2E3" source="EAID_7DC483DF_A
75CE_F29D_47bf_867C_FC5D7C2C98A6"/>
lement xmi:type="uml:InformationFlow" xmi:id="EAID_BEDC08B7_8F28_47a8_B957_96AF5B00AD18" source="EAID_839075CE_F
33DF_AA9C_4c3b_AE31_8EB538CA59E5"/>
ent>


file_data base_Package="EAPK_5C695FE3_8C91_4107_9D63_94A0EC30CE3E" _profile_data="&lt;memo&gt;"/>
it="EAID_0978D9F2_B318_4bb2_B4C4_6877CD488D89" RACtype="RACtype"/>
AID_1DF84F4C_4FD1_47e5_84E4_1121C23A0788" RAOtype="RAOtype"/>
rmationFlow="EAID_DF42C932_4BA7_448f_AE4F_2B850D312417"/>
ormationFlow="EAID_2C3D86E6_A159_49a5_96AD_34FE577A4457"/>
nomic base_Class="EAID_BE5AD0A6_EF52_41e8_8F94_4B922B90A79E" autonomic="self-configuration, self-healing"/>
'EAID_BE5AD0A6_EF52_41e8_8F94_4B922B90A79E" repository="local repository" RAOLtype="RAOLtype"/>
ent="EAID_594D0F40_FCEE_4e38_8BE2_C0DF1F1D666C" repository="repository" RACStype="RACStype"/>
```

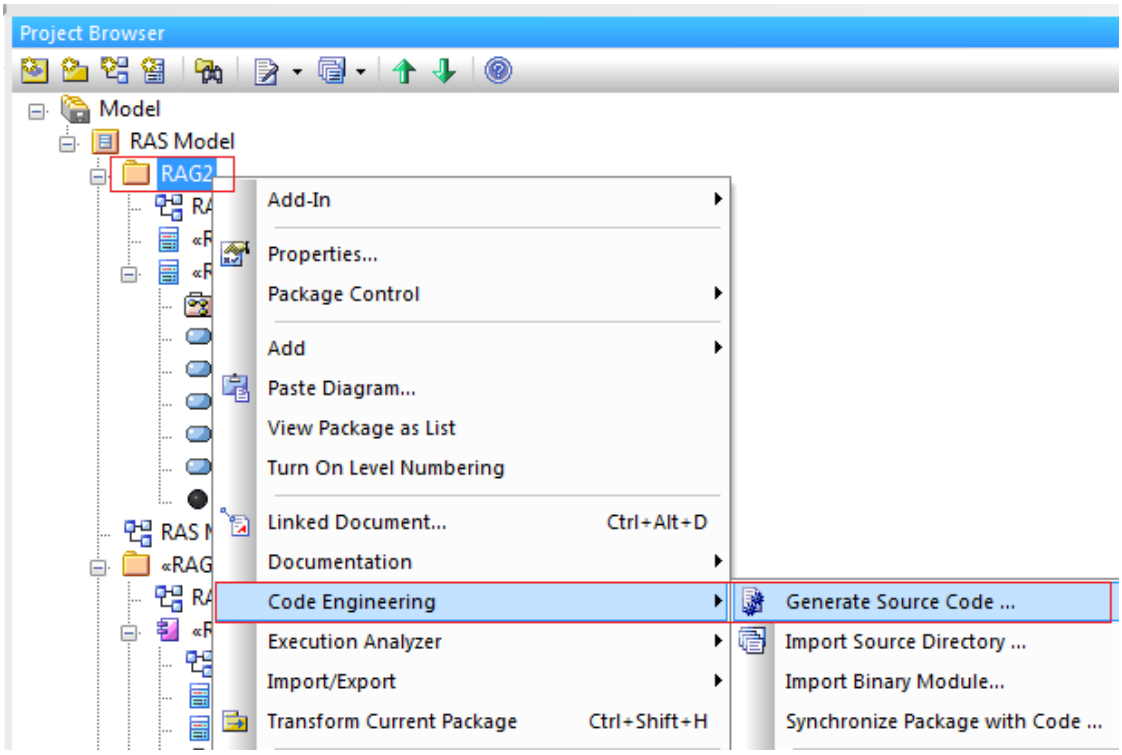
Step 25 (Phase 1): In order to generate the code template for the RAE in the RAS model, we can right click "<<RAOL>> RAOL1" --> "Generate Code..." from the project browser as the figure below illustrates.



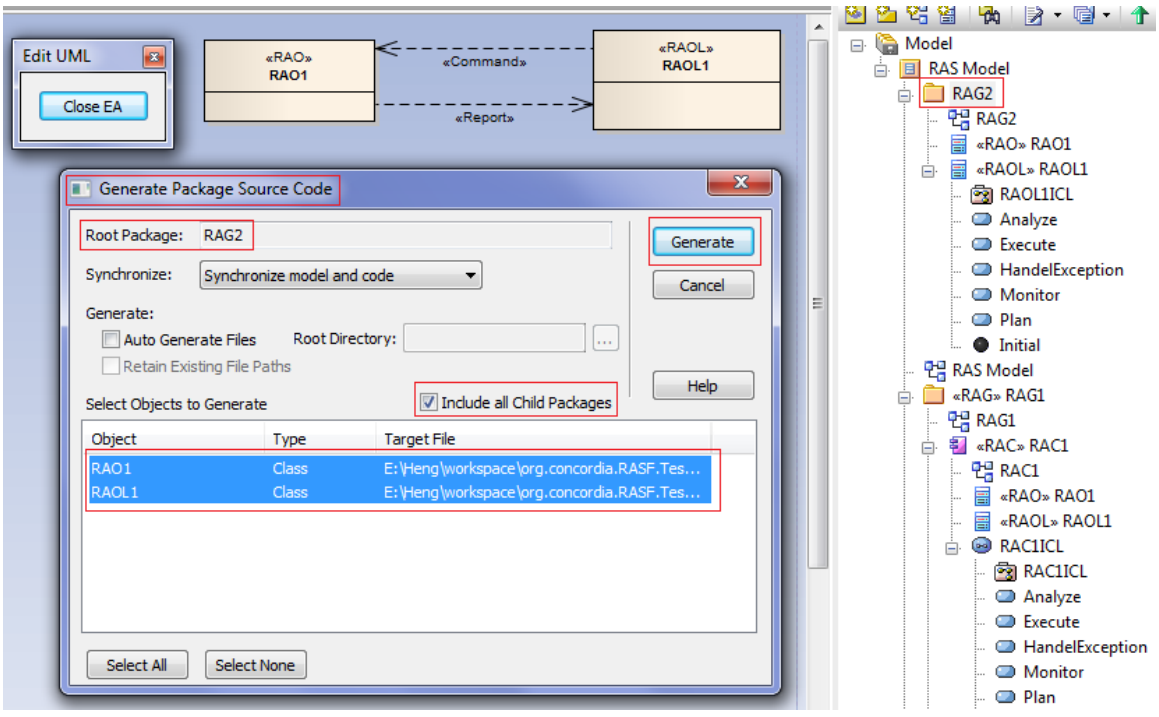
Step 26a (Phase 1): From the popup dialogue "Generate Code", we need to choose the path of the source code template --> select the target language --> click button "Generate", and that template will be created under the specified path (see the figures below).

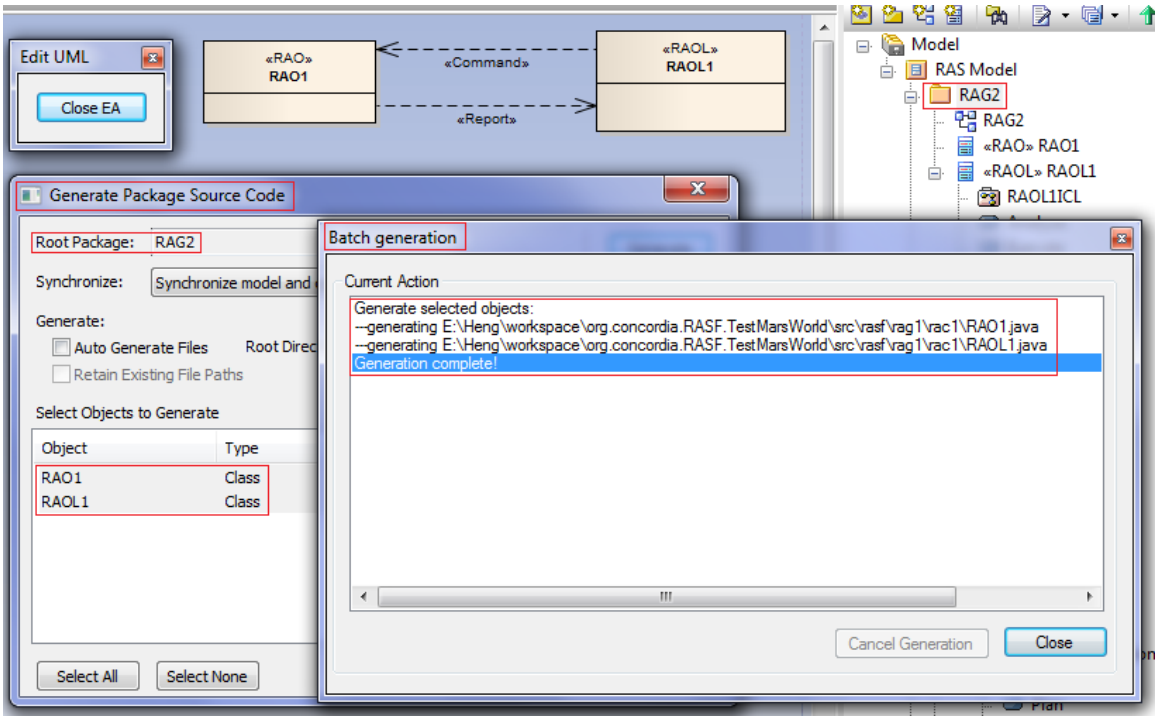


Step 26b (Phase 1): Alternatively, we may generate the source code templates for a group of classes under the same package by right clicking the package name "RAG2" --> selecting "Code Engineering" --> clicking "Generate Source Code...".

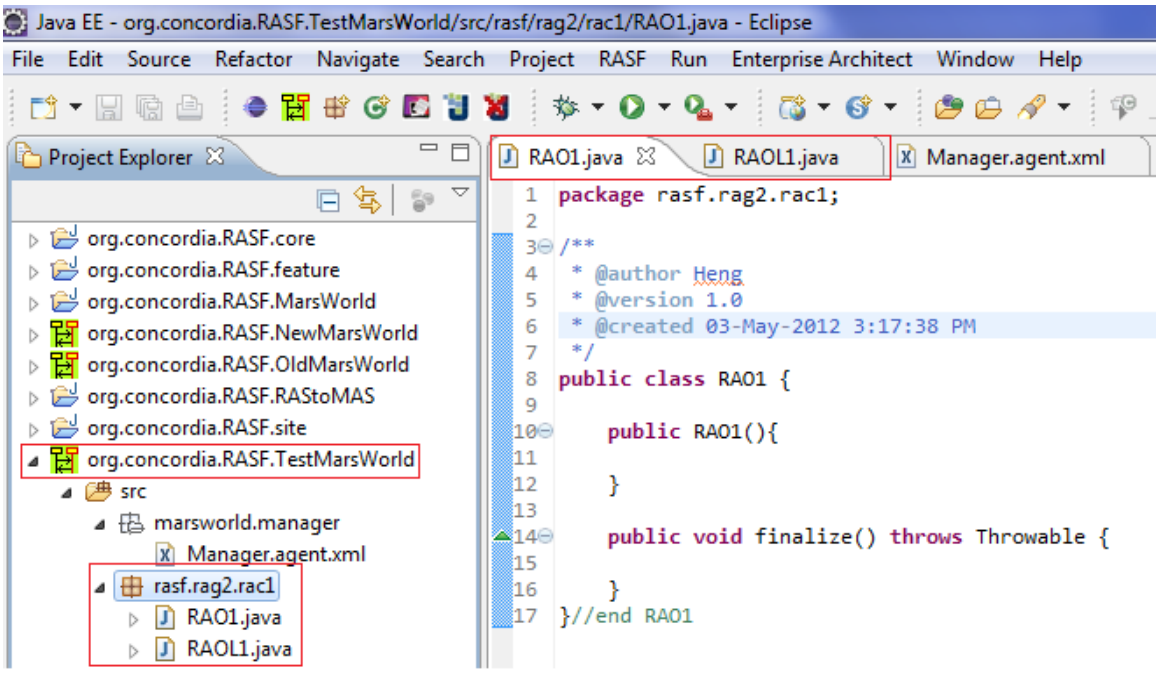


Step 27 (Phase 1): From the popup dialogue "Generate Package Source Code", we need to check "Include all Child Packages" --> select the objects to create --> click button "Generate" (see the figures below).



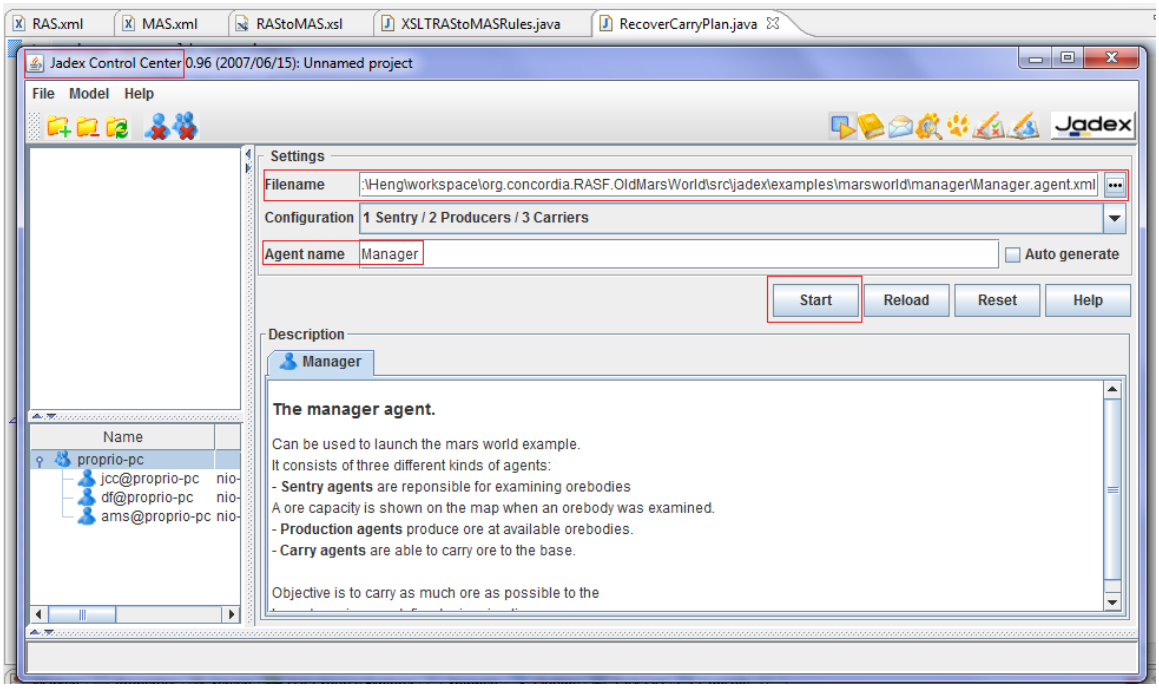


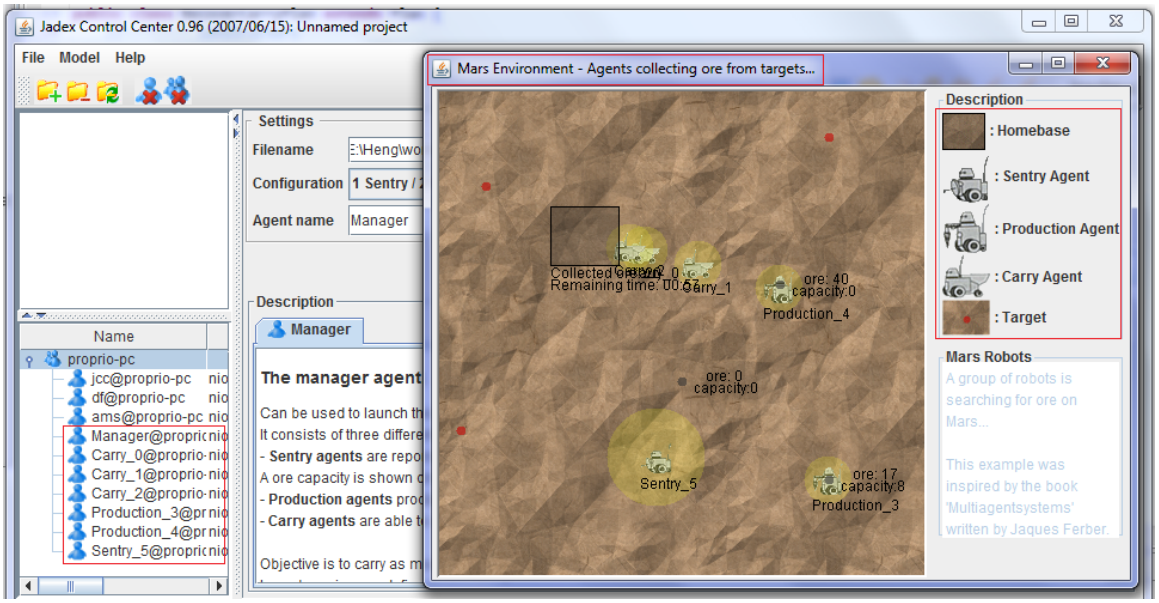
Step 28 (Phase 1): If the generation process is completed successfully, the source code templates will be created under the specified package as the figure below shows.



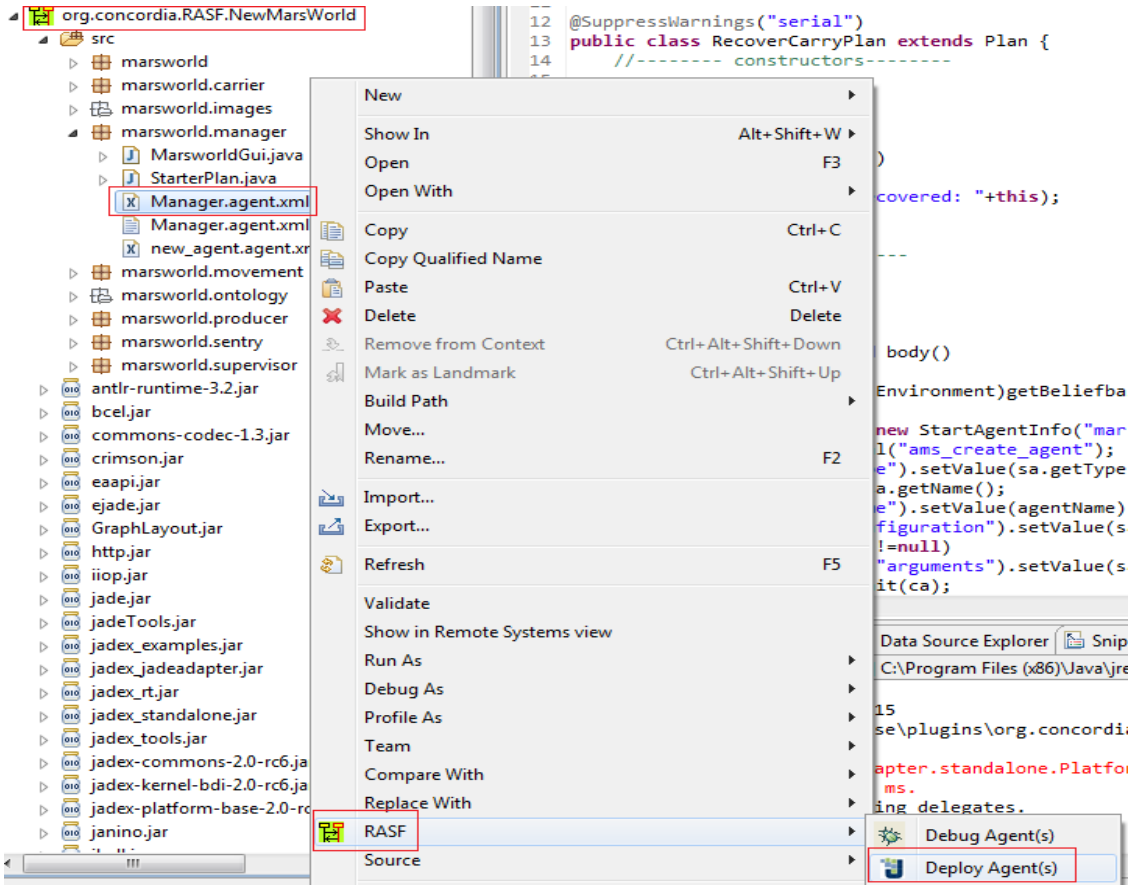
Step 30a (Phase 9): From the popup window "Jadex Control Center", we need to input the path of the agent defintion file (Manager.agent.xml) and click the button "Start" to

deploy the MAS implementation on the Jade and Jadex platform as the figures below.





Step 30b (Phase 9): Alternatively, we can deploy MAS implementation by right clicking the agent definition file --> "RASF" --> "Deploy Agent(s)" (see the figure below).


org.concordia.RASF.NewMarsWorld
src
marsworld
marsworld.carrier
marsworld.images
marsworld.manager
MarsworldGui.java
StarterPlan.java
Manager.agent.xml
Manager.agent.xml
new_agent.agent.xr
marsworld.movement
marsworld.ontology
marsworld.producer
marsworld.sentry
marsworld.supervisor
antlr-runtime-3.2.jar
bcel.jar
commons-codec-1.3.jar
crimson.jar
eaapi.jar
ejade.jar
GraphLayout.jar
http.jar
iiop.jar
jade.jar
jadeTools.jar
jadex_examples.jar
jadex_jadeadapter.jar
jadex_rt.jar
jadex_standalone.jar
jadex_tools.jar
jadex-commons-2.0-rc6.ja
jadex-kernel-bdi-2.0-rc6.ja
jadex-platform-base-2.0-rc
janino.jar
New
Show In Alt+Shift+W
Open F3
Open With
Copy Ctrl+C
Copy Qualified Name
Paste Ctrl+V
Delete Delete
Remove from Context Ctrl+Alt+Shift+Down
Mark as Landmark Ctrl+Alt+Shift+Up
Build Path
Move...
Rename... F2
Import...
Export...
Refresh F5
Validate
Show in Remote Systems view
Run As
Debug As
Profile As
Team
Compare With
Replace With
RASF
Source
Debug Agent(s)
Deploy Agent(s)
@SuppressWarnings("serial")
public class RecoverCarryPlan extends Plan {
//-------- constructors--------
covered: "+this);
body()
Environment)getBeliefba
new StartAgentInfo("mar
l("ams_create_agent");
e").setValue(sa.getType
a.getName();
e").setValue(agentName)
figuration").setValue(s
!=null)
"arguments").setValue(s
it(ca);
Data Source Explorer
Snip
C:\Program Files (x86)\Java\jre
15
se\plugins\org.concordi
apter.standalone.Platfo
ms.
ing delegates.

Appendix I: Representation of Categorical Model in Mars-world

```
<CATEGORY name = "Production-Robot1">
    <OBJECT>
        <OBJECT name = "Sensor1" type = "Sensor"/>
        <OBJECT name = "Drill1" type = "Drill"/>
        <OBJECT name = "CU1" type = "CU"/>
    </OBJECT>
    <MORPHISM>
        <MORPHISM name = "Command1"
                    type ="Command-from-CU-to-Sensor"/>
            <FROM-OBJECT name = "CU1" type = "CU"/>
            <TO-OBJECT name = "Sensor1" type = "Sensor"/>
        </MORPHISM>
        <MORPHISM name = "Command2"
                    type ="Command-from-CU-and-Drill"/>
            <FROM-OBJECT name = "CU1" type = "CU"/>
            <TO-OBJECT name = "Drill1" type = "Drill"/>
        </MORPHISM>
        <MORPHISM name = "Report1" type ="Command-from-Sensor-to-CU"/>
            <FROM-OBJECT name = "Sensor1" type = "Sensor"/>
            <TO-OBJECT name = "CU1" type = "CU"/>
        </MORPHISM>
        <MORPHISM name = "Report2" type ="Command-from-Drill-to-CU"/>
            <FROM-OBJECT name = "Drill1" type = "Drill"/>
            <TO-OBJECT name = "CU1" type = "CU"/>
        </MORPHISM>
        <MORPHISM name = "Cooperate1"
                    type ="Communication-from-Sensor-to-Drill"/>
            <FROM-OBJECT name = "Sensor1" type = "Sensor"/>
            <TO-OBJECT name = "Drill1" type = "Drill"/>
        </MORPHISM>
        <MORPHISM name = "Cooperate2"
                    type ="Communication-from-Drill-to-Sensor"/>
            <FROM-OBJECT name = "Drill1" type = "Drill"/>
            <TO-OBJECT name = "Sensor1" type = "Sensor"/>
        </MORPHISM>
    </MORPHISM>
</CATEGORY>
```

Figure 320: XML Specification of Category **Production-Robot1**

```
<CATEGORY name = "Robot-Group-Formation">
    <OBJECT>
        <OBJECT name = "Production-Robot"/>
        <OBJECT name = "Sentry-Robot"/>
        <OBJECT name = "Carry-Robot"/>
        <OBJECT name = "Supervisor-Robot"/>
    </OBJECT>
     <MORPHISM>
        <MORPHISM name = "Communication-from-Supervisor-to-PR"/>
            <FROM-OBJECT name = "Supervisor-Robot"/>
            <TO-OBJECT name = "Production-Robot"/>
        </MORPHISM>
        <MORPHISM name = "Communication-from-Supervisor-to-SR"/>
            <FROM-OBJECT name = "Supervisor-Robot"/>
            <TO-OBJECT name = "Sentry-Robot"/>
        </MORPHISM>
        <MORPHISM name = "Communication-from-Supervisor-to-CR"/>
            <FROM-OBJECT name = "Supervisor-Robot"/>
            <TO-OBJECT name = "Carry-Robot"/>
        </MORPHISM>
        <MORPHISM name = "Communication-from-PR-to-Supervisor"/>
            <FROM-OBJECT name = "Production-Robot"/>
            <TO-OBJECT name = "Supervisor-Robot"/>
        </MORPHISM>
        <MORPHISM name = "Communication-from-SR-to-Supervisor"/>
            <FROM-OBJECT name = "Sentry-Robot"/>
            <TO-OBJECT name = "Supervisor-Robot"/>
        </MORPHISM>
        <MORPHISM name = "Communication-from-CR-to-Supervisor"/>
            <FROM-OBJECT name = "Carry-Robot"/>
            <TO-OBJECT name = "Supervisor-Robot"/>
        </MORPHISM>
        <MORPHISM name = "Communication-from-SR-to-PR"/>
            <FROM-OBJECT name = "Sentry-Robot"/>
            <TO-OBJECT name = "Production-Robot"/>
        </MORPHISM>
        <MORPHISM name = "Communication-from-PR-to-SR"/>
            <FROM-OBJECT name = "Production-Robot"/>
            <TO-OBJECT name = "Sentry-Robot"/>
        </MORPHISM>
        <MORPHISM name = "Communication-from-PR-to-CR"/>
```

```
            <FROM-OBJECT name = "Production-Robot"/>
            <TO-OBJECT name = "Carry-Robot"/>
        </MORPHISM>
        <MORPHISM name = "Communication-from-CR-to-PR"/>
            <FROM-OBJECT name = "Carry-Robot"/>
            <TO-OBJECT name = "Production-Robot"/>
        </MORPHISM>
    </MORPHISM>
</CATEGORY>
```

Figure 321: XML Specification of Category **Robot-Group-Formation**

```
<CATEGORY name = "Exploration-Group1">
    <OBJECT>
        <OBJECT name = "PR1" type = "Production-Robot" />
        <OBJECT name = "SR1" type = "Sentry-Robot"/>
        <OBJECT name = "CR1" type = "Carry-Robot"/>
        <OBJECT name = "Supervisor1" type = "Supervisor-Robot"/>
    </OBJECT>
    <MORPHISM>
        <MORPHISM name = "Command1"
                    type =" Communication-from-Supervisor-to-PR"/>
            <FROM-OBJECT name = "Supervisor1" type = "Supervisor-Robot"/>
            <TO-OBJECT name = "PR1" type = "Production-Robot"/>
        <MORPHISM>
        <MORPHISM name = "Command2"
                    type =" Communication-from-Supervisor-to-SR"/>
            <FROM-OBJECT name = "Supervisor1" type = "Supervisor-Robot"/>
            <TO-OBJECT name = "SR1" type = "Sentry-Robot"/>
        <MORPHISM>
        <MORPHISM name = "Command3"
                    type =" Communication-from-Supervisor-to-CR"/>
            <FROM-OBJECT name = "Supervisor1" type = "Supervisor-Robot"/>
            <TO-OBJECT name = "CR1" type = "Carry-Robot"/>
        <MORPHISM>
        <MORPHISM name = "Report1"
                    type =" Communication-from-PR-to-Supervisor"/>
            <FROM-OBJECT name = "PR1" type = "Production-Robot"/>
            <TO-OBJECT name = "Supervisor1" type = "Supervisor-Robot"/>
        <MORPHISM>
        <MORPHISM name = "Report2"
                    type =" Communication-from-SR-to-Supervisor"/>
```

```
            <FROM-OBJECT name = "SR1" type = "Sentry-Robot"/>
            <TO-OBJECT name = "Supervisor1" type = "Supervisor-Robot"/>
        <MORPHISM>
        <MORPHISM name = "Report3"
                        type =" Communication-from-CR-to-Supervisor"/>
            <FROM-OBJECT name = "CR1" type = "Carry-Robot"/>
            <TO-OBJECT name = "Supervisor1" type = "Supervisor-Robot"/>
        <MORPHISM>
        <MORPHISM name = "Cooperate1"
                        type =" Communication-from-SR-to-PR"/>
            <FROM-OBJECT name = "SR1" type = "Sentry-Robot"/>
            <TO-OBJECT name = "PR1" type = "Production-Robot"/>
        <MORPHISM>
        <MORPHISM name = "Cooperate2"
                        type =" Communication-from-PR-to-SR"/>
            <FROM-OBJECT name = "PR1" type = "Production-Robot"/>
            <TO-OBJECT name = "SR1" type = "Sentry-Robot"/>
        <MORPHISM>
        <MORPHISM name = "Cooperate3"
                        type =" Communication-from-PR-to-CR"/>
            <FROM-OBJECT name = "PR1" type = "Production-Robot"/>
            <TO-OBJECT name = "CR1" type = "Carry-Robot"/>
        <MORPHISM>
        <MORPHISM name = "Cooperate4"
                        type =" Communication-from-CR-to-PR"/>
            <FROM-OBJECT name = "CR1" type = "Carry-Robot"/>
            <TO-OBJECT name = "PR1" type = "Production-Robot"/>
        <MORPHISM>
     </MORPHISM>
</CATEGORY>
```

Figure 322: XML Specification of Category **Exploration-Group1**

```
<CATEGORY name = "Mars-World-Formation">
    <OBJECT>
        <OBJECT name = "Exploration-Group"/>
        <OBJECT name = "Production-Group" />
        <OBJECT name = "Carry-Group" />
    </OBJECT>
     <MORPHISM>
        <MORPHISM name = "Communication-from-EG-to-PG"/>
            <FROM-OBJECT name = "Exploration-Group"/>
```

```
            <TO-OBJECT name = "Production-Group"/>
        <MORPHISM>
        <MORPHISM name = "Communication-from-PG-to-CG"/>
            <FROM-OBJECT name = "Production-Group"/>
            <TO-OBJECT name = "Carry-Group"/>
        <MORPHISM>
        <MORPHISM name = "Communication-from-CG-to-EG"/>
            <FROM-OBJECT name = "Carry-Group"/>
            <TO-OBJECT name = "Exploration-Group"/>
        <MORPHISM>
        <MORPHISM name = "Communication-from-PG-to-EG"/>
            <FROM-OBJECT name = "Production-Group"/>
            <TO-OBJECT name = "Exploration-Group"/>
        <MORPHISM>
        <MORPHISM name = "Communication-from-EG-to-CG"/>
            <FROM-OBJECT name = "Exploration-Group"/>
            <TO-OBJECT name = "Carry-Group"/>
        <MORPHISM>
        <MORPHISM name = "Communication-from-CG-to-PG"/>
            <FROM-OBJECT name = "Carry-Group"/>
            <TO-OBJECT name = "Production-Group"/>
        <MORPHISM>
    </MORPHISM>
</CATEGORY>
```

Figure 323: XML Specification of Category **Mars-World-Formation**