International Conference on Software Methodologies, Tools and Techniques, 2010
Frontiers in Artificial Intelligence and Applications, IOS Press, pp. 357-380

Formal Specification of Substitutability Property for Fault-Tolerance in Reactive Autonomic Systems

Heng Kuang, Jamal Bentahar, Olga Ormandjieva, Nassir Shafieidizaji and Stan Klasa
Concordia University

Abstract. Multi-Agent Systems (MAS) have been widely proposed and applied to various application domains, where traditional approaches are impractical, such as space exploration missions. MAS can offer greater redundancy, efficiency, and scalability; however, they also raise new challenges, such as complex and often unexpected emergent group behavior, which require a formal specification as well as verification. Therefore, we have proposed a formal approach, named Reactive Autonomic Systems Framework (RASF), based on category theory to tackle those challenges. In this paper, we focus on the formal specification of substitutability property for the fault-tolerance and illustrate our approach through a Mars-world case study implemented as MAS using JADEX.

Keywords. Formal specification, category theory, substitutability property, fault-tolerance, reactive autonomic systems, multi-agent systems.

Introduction

The exploration of the planet Mars has to be carried out with the aid of robots. But radio transmissions in one direction take at least 3 to 5 minutes to travel from Mars to the Earth, so the robots must be autonomic and autonomous enough to carry out tasks using their own resources as well as intelligence with minimum human intervention from the Earth. Moreover, the exploration tasks may be achieved through a large number of inexpensive little robots with very simple capabilities, and the robots work as intelligent swarms. Therefore, the coordination of actions between those robots becomes critical and offers the following advantages: 1) greater redundancy and thus greater protection of assets; 2) greater exploration efficiency; and 3) reduced costs and risk. However, this new mission architecture raises the following new challenges: 1) the aggregation of simple individual behavior of each robot exhibits complex and often unexpected emergent group behavior; 2) the autonomy degree of the missions requires a prohibitive amount of testing in order to achieve system validation; and 3) emergent behavior patterns may not be fully predicated through using traditional development methods. Thus, a formal specification for autonomic behavior with self-management properties and a formal verification for emergent behavior are required to address those challenges.

The authors in [1] have compared current formal methods as well as integrated formal methods for effective specification and emergent behavior predication of the intelligent swarm systems with autonomic and autonomous properties. However, there

is no single formal method satisfying all the required properties, and specifications cannot be easily converted to program code or applied as the input for model checkers when using integrated formal methods. To address this particular issue, we have proposed a new formal approach, the Reactive Autonomic Systems Framework (RASF) [2,4].

RASF is a formal framework based on the Category Theory (CT) [2] to model the Reactive Autonomic Systems (RAS). Our first step is to build a RAS meta-model in terms of transforming it to the RAS models according to different applications, such as the Mars exploration mission, and then implement those RAS models through the Multi-Agent Systems (MAS) approach. The RAS meta-modeling supports a process of constructing the RAS models by providing the categorical *correct by construction* rules, properties, and constraints that are derived from the categorical specification of the RAS meta-model [2]. Therefore, allowable evolutions of the RAS models restricted by those constraints, properties, and rules can be assured.

Our previous proposals [2,3,4] have specified the architecture, reactive as well as autonomic behavior, and the self-monitoring property of the RAS meta-model. In this paper, we will focus on specifying the substitutability property for the fault-tolerance in the RAS meta-model and illustrate our approach through the Mars-world case study in [5], which has been inspired by the Mars exploration mission we stated above. The rest of this paper is organized as follows: Section 1 describes the Mars-world case study by which we illustrate our approach; Section 2 presents the architecture and behavior of the RAS meta-model; Section 3 introduces the category theory that is the theoretical foundation of our methodology; Section 4 demonstrates a categorical specification of substitutability property in terms of fault-tolerance for the RAS meta-model; Section 5 illustrates our approach by the MAS implementation of the case study in JADEX; Section 6 discusses the related work; and Section 7 states our conclusions and outlines directions for future work.

1. Case Study

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. Thus, three types of robots are required, and all the robots have a sensor range to detect occurrences of ore. 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 a mining device. After mining is finished, a group of *carry robots* is requested to transport ore to the home base. In addition to those three types of specialist robots, we consider two types of administration robots to coordinate group level tasks and system level tasks. A *group supervisor robot* is able to form a group of robots to find and exploit ore within the areas requested by a *system manager robot*. It is able to 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 insure fault-tolerance, since they serve as critical roles and store important data, such as repositories and work outcomes.

Figure 1 depicts a sample scenario of the Mars-world. After a *system manager robot* receives the coordinate of exploration area from the *ground station* on Earth, it evaluates that area and the capability of each available *group supervisor robot* and orders two of them to explore subarea1 as well as subarea2 respectively. When a *group supervisor robot* receives an order with the subarea coordinate, 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* as well as *carry robots* because of new detected ore. The amount of remaining ore of each spot is indicated as a percentage number and reported to *supervisor robot* by the *sentry robots*.

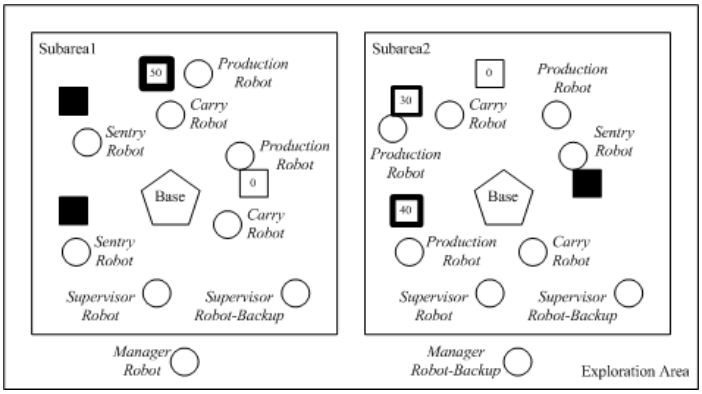


Figure 1. A sample scenario of Mars-world

2. Reactive Autonomic Systems Framework

2.1. Architecture of the RAS Meta-model

The RAS meta-model (see Figure 2) is a four-layer architecture, which consists of the Reactive Autonomic Objects (RAO), the Reactive Autonomic Components (RAC), the Reactive Autonomic Component Groups (RACG) as well as the RAS. The autonomic features are implemented by the RAO Leaders (RAOL), the RAC Supervisors (RACS), and the RACG Managers (RACGM) at RAC, RACG and RAS layer respectively [2]. In this layered meta-model, each tier communicates only with the tier immediately above or below it. Therefore, the independence of those tiers makes their modularity, encapsulation, hierarchical decomposition, and reuse possible. Figure 3 depicts an example of the RAS model built from the RAS meta-model for the simplified scenario of the Mars-world presented in Section 1.

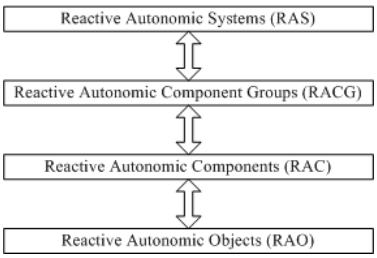


Figure 2. Architecture of RAS meta-model

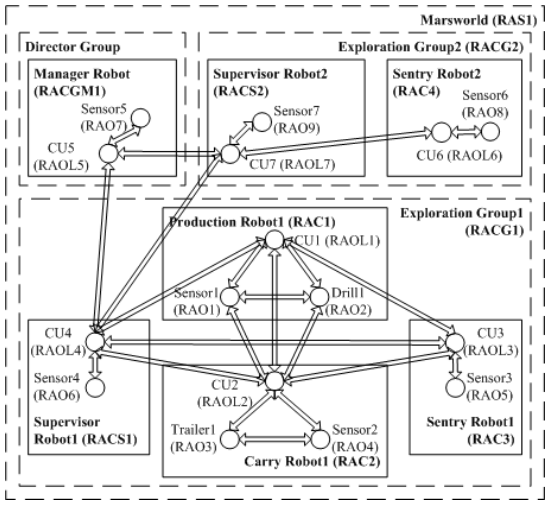


Figure 3. Example of Mars-world modeled in RASF

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), and a *sentry robot* has an enhanced sensor (RAO5) instead of standard one.

The RAO is modeled as a labeled transition system augmented with ports, resources, attributes, logical assertion on those attributes, and time constraints [3]. The RAC is specified by a set of RAOs, where one of them is named as a team leader (RAOL). The team members are responsible for reactive tasks and RAOL works on autonomic tasks. Figure 4 illustrates an example of such a specification as an XML file. The RACG is a set of RAC which cooperate in the fulfillment of group tasks, where one of its RAC is named as the group supervisor (RACS). RACG is the minimum reactive autonomic entity (RAE) that can independently fulfill a complete work in the RAS meta-model; the autonomic behavior at this layer is coordinated by RACS. The

RAS is a set of RACG, where one of them is named as a director group in which one of the RAC is named as a system manager (RACGM). The director group provides an integrated interface to delegate tasks, manage repositories, and monitor systems. RACGM is responsible for coordinating autonomic behavior at this layer. A specification example is given as XML file in Figure 5.

```
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 4. Specification of production robot

```
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 5. Specification of Marsworld

2.2. Behavior of the RAS Meta-Model

The synchronous interaction between RAO is modeled as a synchronous product machine; more details about the reactive behavior specifications for those RAO can be found in [3]. The autonomic behaviors of the RAOL, RACS, and RACGM are modeled as intelligent control loops specified as labeled transition systems (Figure 6), where states specify the task (*Monitor*, *Analyze*, *Plan*, *Execute*, *HandleException*); events specify the triggers from one state to another; and transitions specify state sequence under time constraints. More details about intelligent control loops in the RAS meta-model can be found in [4,6].

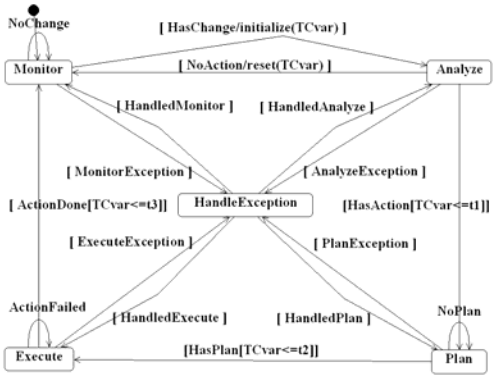


Figure 6. Intelligent control loop model

3. Category Theory

Structure is crucial in large specifications and programs; a well-chosen structure greatly improves understanding, validation, and modification of a specification. In RAS where emergent behavior is one of the most important features, the management of evolving specifications and analysis of changes require a specification structure, which is able to isolate those changes in a small number of components and analyze the impacts of a change on interconnected components [7].

Category theory has been proposed as a framework to offer that structure; it has a rich body of theories to reason on specifications along with their interactions, and it is abstract enough for a wide range of different specification languages. Moreover, automation may be achieved in category theory, for instance, composition of two specifications can be derived automatically and the category of those specifications has some properties, such as co-completeness. Category theory for software specifications has adopted a *correct by construction* approach by which components are specified, proved, and composed in the way of preserving their properties [7].

Complex systems may be represented by diagrams (semi-formal), where system components along with connectors and their interconnections represent nodes and edges respectively. On the other hand, the word *diagram* in this theory has formal meaning and carries all the intuitions that come from practice. Compared to other formalization of software concepts, category theory does not provide a semantic tool to formalize the description of components and connectors, but it allows formalizing and reasoning about interconnections, configurations, instantiations, and compositions that are important aspects of engineering RAS with both autonomous and autonomic behaviors. This can be achieved at a very abstract level, as category theory proposes a toolbox applied to whatever formalism to capture components' behavior, as long as that formalism satisfies certain structure properties [8].

Category theory concentrates on the relationships (called *morphisms*) between objects instead of their representations, and those morphisms are able to determine the nature of interactions established between the objects. Therefore, a particular category may reflect a specific architecture style. Moreover, category theory provides techniques

to manipulate and reason about diagrams for building hierarchies of system complexity, which allows systems to be used as the components of more complex systems and makes inferring the properties of those systems from their configurations possible [8]. Let's recall some useful definitions related to this theory (from [9,10]) and their use in this work.

Definition 3.1. A category ***C*** consists of following data and rules:

- A class of *objects*: A, B, etc. We use $|\boldsymbol{C}|$ to denote the set of all these objects.
- A class of *arrows* (*morphisms*): f, g, etc.
- For each arrow f: $A \rightarrow B$, A is called the *domain* of f, denoted as $dom(f)$, and B is called the *codomain* of f, denoted as $cod(f)$. We use $\boldsymbol{C}(A,B)$ to indicate the set of all arrows in ***C*** from A to B.
- For each pair of arrows f: $A \rightarrow B$ and g: $B \rightarrow C$, a *composite morphism* is denoted as $g \circ f$: $A \rightarrow C$.
- For each object A, an *identity morphism* has both domain A and codomain A denoted as Id_A: $A \rightarrow A$.
- Identity composition: $f \circ Id_A = f = Id_B \circ f$ for each *morphism* f: $A \rightarrow B$.
- Associativity: $h \circ (g \circ f) = (h \circ g) \circ f$ for each set of *morphisms* f: $A \rightarrow B$, g: $B \rightarrow C$, and h: $C \rightarrow D$.
- Inverse of a *morphism* f: $A \rightarrow B$ is a *morphism* g: $B \rightarrow A$ such that $f \circ g = Id_B$ and $g \circ f = Id_A$; we denote the inverse of f as f^{-1} if it exists, and a *morphism* can have at most one inverse.

If f has an inverse, it is said to be an isomorphism; if f: $A \rightarrow B$ is an isomorphism, then A and B are said to be isomorphic, denoted as $A \cong B$. For example, the functions between sets give rise to a category, where the objects are sets and the *morphisms* are all functions between them.

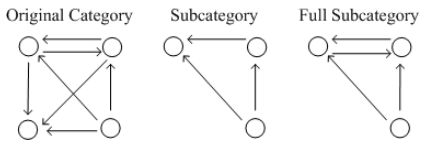
Proposition 3.1. In the RAS meta-model we presented in Section 2, the RAC is a category ***RAC*** having a set of objects (RAO_1, ..., RAO_n) $(\forall 1 \le i \le n\ RAO_i \in |\boldsymbol{RAC}|)$ and their interactions are *morphisms*.

Proof. *All what we need is to prove the existence of composite and identity morphisms. Let RAO_i, RAO_j, and RAO_k be three ROA such that RAO_i can interact with RAO_j, which can interact with RAO_k. Then RAO_i can interact with RAO_k (indirectly through RAO_j), which means the existence of a composite morphism between RAO_i and RAO_k. The identity morphism does exist as a natural representation of internal interactions. Let f and g be the morphisms: f: $RAO_i \rightarrow RAO_j$ and g: $RAO_j \rightarrow RAO_i$. It is clear that $f \circ g = Id_{RAOj}$ and $g \circ f = Id_{RAOi}$.*■

Thus, every robot in the Mars-world, a production robot, for instance, is a category **Production-Robot1** (**PR1**) consisting of objects *Drill1*, *Sensor1*, *Control-Unit1* (*CU1*), as well as their interactions **PR1**(*Drill1*, *Sensor1*), **PR1**(*CU1*, *Drill*), and **PR1**(*CU1*, *Sensorl*).

Definition 3.2. Let ***C*** and ***D*** be two categories. ***C*** is a *subcategory* of ***D*** denoted as $\boldsymbol{C} \lhd \boldsymbol{D}$ if $|\boldsymbol{C}| \subseteq |\boldsymbol{D}|$, and the morphisms of ***C*** are morphisms of ***D*** as $\boldsymbol{C}(A_i, A_j) \subseteq \boldsymbol{D}(A_i, A_j)$

where $A_i, A_j \in |\boldsymbol{C}|$; $\boldsymbol{C}$ is a *full subcategory* of $\boldsymbol{D}$ when $\boldsymbol{C}(A_i, A_j) = \boldsymbol{D}(A_i, A_j)$ for all objects of $\boldsymbol{C}$.



Proposition 3.2. The RACG is a category **RACG** with a set of full subcategories **RAC**, and the RAS is a category **RAS** having a family of full subcategories **RACG**.

Thus, for an *exploration group* in Mars-world, for example, the category **Exploration Group1** (**EG1**) includes a set of full subcategories **PR1**, **Carry Robot1** (**CR1**), **Sentry Robot1** (**SR1**), and **Supervisor Robot1** (**Supervisor1**).

Definition 3.3. A *functor* ("the homomorphism of categories") F: $\boldsymbol{C} \rightarrow \boldsymbol{D}$ between two categories $\boldsymbol{C}$ and $\boldsymbol{D}$ is a mapping of objects to objects and arrows to arrows from $\boldsymbol{C}$ to $\boldsymbol{D}$ in the following way:

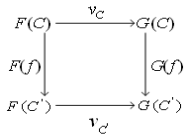
- Object mapping as F: $|\boldsymbol{C}| \rightarrow |\boldsymbol{D}|$.
- Arrow mapping as F: $\boldsymbol{C}(A_i, A_j) \rightarrow \boldsymbol{D}(F(A_i), F(A_j))$.
- Composition mapping as $F(g \circ f) = F(g) \circ F(f)$ where $g, f \in \boldsymbol{C}$ and $F(g), F(f) \in \boldsymbol{D}$.
- Identity mapping: $F(Id_A) = Id_{F(A)}$ where $Id_A \in \boldsymbol{C}$ and $Id_{F(A)} \in \boldsymbol{D}$.

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

Proposition 3.3. The evolutions of a RAC, because of self-adaptation and self-organization during run time, are functors.

For instance, an evolution from RAC to RAC' is a functor F, which includes a mapping of objects (RAO) in **RAC** to the objects (RAO') in **RAC'** (F: $|\mathbf{RAC}| \rightarrow |\mathbf{RAC'}|$), and a mapping of morphisms in **RAC** to morphisms in **RAC'** (F: $\mathbf{RAC}(RAO_i, RAO_j) \rightarrow \mathbf{RAC'}(F(RAO_i), F(RAO_j))$). Similarly, the evolution of RACG and RAS are functors F: **RACG** $\rightarrow$ **RACG'** and F: **RAS** $\rightarrow$ **RAS'** respectively. The evolution of a robot in Mars-world, for example, from **PR1** to **PR1'**, because of the new configuration for its *drill1* or *sensor1*, is a functor F: **PR1** $\rightarrow$ **PR1'**. 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** $\rightarrow$ **EG1'**.

Definition 3.4. For categories $\boldsymbol{C}$, $\boldsymbol{D}$, and their functors F, G: $\boldsymbol{C} \rightarrow \boldsymbol{D}$, a *natural transformation* (v: $F \rightarrow G$) is a family of arrows in $\boldsymbol{D}$ as v_c: $F(C) \rightarrow G(C)$, such that, for any f: $C \rightarrow C'$ in $\boldsymbol{C}$, $v_{C'} \circ F(f) = G(f) \circ v_C$, as in the following diagram. Given such a natural transformation v, the $\boldsymbol{D}$-arrow v_C is called a *component* of v at C, and, if v is invertible, it is known as a *natural isomorphism*.



For example, every group is naturally isomorphic to its opposite group. Because the evolutions of RAC, RACG, and RAS are specified as functors from category **RAC** to **RAC**', **RACG** to **RACG**' as well as **RAS** to **RAS**', the natural transformation may represent the mapping of those evolutions. For example, the relationship between two solutions in terms of fault-tolerance for an *exploration group* in Mars-world, *Solution1*: **EG1** → **EG**' and *Solution2*: **EG1** → **EG1**', can be modeled by a natural transformation *convert1*: *Solution1* → *Solution2*.

Definition 3.5. If ***C*** is a full subcategory of ***D*** and every $D \in \boldsymbol{D}$ is isomorphic to some object in ***C***, then the insertion functor F: $\boldsymbol{C} \rightarrow \boldsymbol{D}$ is an equivalence.

4. Categorical Specification of the RAS Meta-Model

4.1. Categorical Specification of Architecture

Similarly as the categorical specification of RAC we introduced in Section 3, RACG is specified by a category **RACG** having a group of objects (*RAC*) and their interactions as morphism *f*: **RACG**(RAC_m, RAC_n), where RAC_m, $RAC_n \in$ |**RACG**| (see Figure 7); RAS is specified by a category **RAS** consisting of a set of objects (*RACG*) and their interactions as morphism *f*: **RAS**($RACG_x$, $RACG_y$), where $RACG_x$, $RACG_y \in$ |**RAS**|. More details about the categorical specifications of the RAS meta-model architecture can be found in [2].

CAT-RACG <name>
Objects: <list of objects specifying a set of RAC in RACG>
Morphisms: <list of morphisms specifying the interactions between RAC>
Limit: <a limit specifying designated behavior model of those RAC>
Colimit: <a colimit specifying actual behavior model of those RAC>
Product Objects: <list of product objects specifying synchronous communication between RAC>
Coproduct Objects: <list of coproduct objects specifying asynchronous communication between RAC>
Pushout Objects: <list of pushout objects specifying next communication relays between RAC>
Pullback Objects: <list of pullback objects specifying previous communication relays between RAC>
Slice Category: <a category specifying outgoing communication and their relations between RAC>
Coslice Category: <a category specifying incoming communication and their relations between RAC>
Subcategories: <list of subcategories specifying a set of RAO in RAC>
Product Categories: <list of product categories specifying interactions between RAC>
Functors: <list of functors specifying the evolutions of a RACG>
Natural Transformations: <list 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 7. Categorical specification of RACG

Definition 4.1. 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'* ∈ |**RAS**| which have the morphisms with *RAE*, and the morphisms are **RAS**(*RAE*, *RAE'*) as well as **RAS**(*RAE'*, *RAE*).

Definition 4.2. The social life of *RAE* is equivalent to the social life of *RAE'* denoted as Social(*RAE*) ~ Social(*RAE'*) iff **SOCIAL**(*RAE*) ~ **SOCIAL**(*RAE'*).

4.2. Categorical Specification of Behavior

As we stated in Section 2, the internal behaviors of each RAE are specified as labeled transition systems, and the interactive behavior between those RAE are modeled by external event sequences. The labeled transition systems and external event sequences can also be specified as **TRANSITION** and **INTERACTION** categories respectively. For example, the behavior of the Intelligent Control Loop Model (ICLM) depicted in Figure 7 is interpreted by the category **TRANSITION**(ICLM) = <Seq_1, Seq_2, …, Seq_n>, where the objects are sequences of transitions, such as Seq_1 = <$Trans_{1\text{-}1}$, $Trans_{1\text{-}2}$, …, $Trans_{1\text{-}m}$> (n, $m \geq 1$), and the morphisms are *isomorphic* relations between those sequences. Thus, the reflective and transitive conditions on *isomorphic* ensure that **TRANSITION**(ICLM) is a category, where composite morphisms, identity morphisms, identity composition, and associativity exist. Moreover, every transition in a sequence is modeled by the tuple (*state*, *event*, *state*) indicating the source state, trigger event, and destination state, such as $Trans_{1\text{-}1}$ = (*Monitor*, *HasChange*, *Analyze*).

Definition 4.3. Two transitions are equivalent denoted as $Trans_1 \sim Trans_2$ iff both transitions have the same source state as well as destination state, and the event of $Trans_1$ is the first event of $Trans_2$. For instance, $Trans_1$ = (*Monitor*, *HasChange*, *Analyze*) is equivalent to composite transition $Trans_1'$ = <$Trans_1$, $Trans_2$, $Trans_3$> = (*Monitor*, (*HasChange*, *AnalyzeException*, *HandledAnalyze*), *Analyze*), where $Trans_2$ = (*Analyze*, *AnalyzeException*, *HandleException*), $Trans_3$ = (*HandleException*, *Handled-Analyze*, *Analyze*).

Definition 4.4. Two transition sequences are isomorphic denoted as $TransSeq_1 \cong TransSeq_2$ iff their transitions are equivalent as <$Trans_{1\text{-}1}$, $Trans_{1\text{-}2}$, …, $Trans_{1\text{-}i}$> ~ <$Trans_{2\text{-}1}$, $Trans_{2\text{-}2}$, …, $Trans_{2\text{-}j}$>, where $i, j \geq 1$.

Similarly, the interactive behavior between RAE is interpreted by the category **INTERACTION**(RAE), where the objects are sequences consisting of actions, such as Seq_1 = <$Act_{1\text{-}1}$, $Act_{1\text{-}2}$, …, $Act_{1\text{-}n}$>, and the morphisms are *isomorphic* relations between those sequences. In addition, every action in a sequence is specified as the tuple (*sender*, *TE*, *LE*, *receiver*) stating the sender of *TE*, the *TE* triggering an action, the *LE* outputted from the action, and the receiver of *LE*.

Definition 4.5. Two actions are equivalent denoted as $Act_1 \sim Act_2$ iff both actions have the same sender as well as receiver, and the *TE* of Act_1 is the first event of *TE* in Act_2 and the *LE* of Act_1 is the last event of *LE* in Act_2. For example, Act_1 = (*RACS*, *StartRAC*, *HeartbeatRAC*, *RACS*) is equivalent to the composite action Act_1' = <Act_2, Act_3> = (*RACS*, (*StartRAC*, *RestartRAC*), (*NoHeartbeatRAC*, *HeartbeatRAC*), *RACS*),

where Act_2 = (*RACS*, *StartRAC*, *NoHeartbeatRAC*, *RACS*), Act_3 = (*RACS*, *RestartRAC*, *HeartbeatRAC*, *RACS*).

Definition 4.6. Two action sequences are isomorphic denoted as $ActSeq_1 \cong ActSeq_2$ iff their actions are equivalent as $<Act_{1\text{-}1}, Act_{1\text{-}2}, \ldots, Act_{1\text{-}i}> \sim <Act_{2\text{-}1}, Act_{2\text{-}2}, \ldots, Act_{2\text{-}j}>$, where $i, j \geq 1$.

After a RAS is formed, RACGM, RACS as well as RALO start their intelligent control loops and monitor the heartbeat messages sent in a certain time interval by RACS, RAOL, and RAO respectively (see Figure 8). After RACGM receives a task from the user console, it chooses an action sequence (*ActSeqRACGM*) from 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 communicate 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 RAO1 for the action $ActRAOL1_1$ in *ActSeqRAOL1*; eventually, RAO1 chooses an action sequence (*ActSeqRAO1*) from **INTERACTION**(RAO1) in terms of realizing $ActRAOL1_1$ and cooperates with RAO2 due to the action $ActRAO1_1$ in *ActSeqRAO1*.

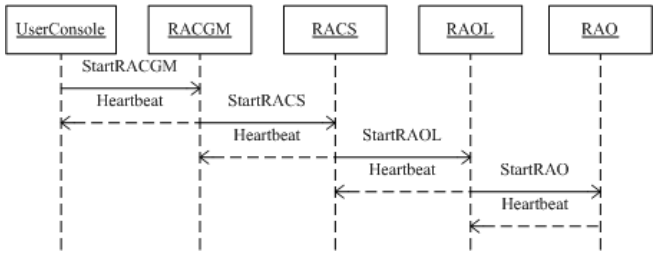


Figure 8. Work flow of forming a RAS

4.3. Categorical Specification of Fault-Tolerance Property

Property 4.7. Internal behavior equivalence in RAE. We state how fault-tolerance is applied to the internal behavior of RAE through the ICLM we described in Section 2.2 (see Figure 7). If there is an exception during $Trans_2$ = (*Analyze*, *HasAction*, *Plan*), RAE transits to *HandleException* state instead of *Plan* triggered by *AnalyzeException* event as $Trans_{2e}$ = (*Analyze*, *AnalyzeException*, *HandleException*); after that 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*, *HandledAnalyze*, *Analyze*) and it proceeds to $Trans_2$ as $TransSeq_1' = <Trans_1, Trans_{2e}, Trans_{2h}, Trans_2, \ldots, Trans_n>$, which is isomorphic to $TransSeq_1 = <Trans_1, Trans_2, \ldots, Trans_n>$ (see Definition 4.4) because of $<Trans_1, Trans_{2e}, Trans_{2h}> \sim <Trans_1>$ (see Definition 4.3). Therefore, the category **TRANSITION**(RAE) consisting of objects $TransSeq_1$, $TransSeq_2$, …, $TransSeq_m$ is a full subcategory of **TRANSITION**(RAE') having objects $TransSeq_1$, $TransSeq_1'$, $TransSeq_2$, …, $TransSeq_m$ (see Definition 3.2)

and both categories are equivalent (see Definition 3.5). This means that two sequences exhibit the same internal behavior in RAE for performing an action and leads to the fault-tolerance property when exceptions occur.

Property 4.8. Interactive behavior equivalence between RAE. We will describe how fault-tolerance is applied to the interactive behavior between RAE through the work flow of forming a RAS (see Figure 10). If a RAC1 cannot be started by a RACS1 due to an exception as $ActRAC1_{1e}$ = (*RACS1*, *StartRAC1*, *NoHeartbeatRAC1*, *RACS1*), RACS1 tries to restart RAC1 later as $ActRAC1_{1h}$ = (*RACS1*, *RestartRAC1*, *Heartbeat-RAC1*, *RACS1*) if the exception can be processed and then it continues to $ActRAC1_2$ as the $ActSeqRAC1_1$**'** = <$ActRAC1_{1e}$, $ActRAC1_{1h}$, $ActRAC1_2$, …, $ActRAC1_n$>, which is isomorphic to $ActSeqRAC1_1$ = <$ActRAC1_1$, $ActRAC1_2$, …, $ActRAC1_n$> (see Definition 4.6) because <$ActRAC1_{1e}$, $ActRAC1_{1h}$> ~ <$ActRAC1_1$> (see Definition 4.5). Thus, the category **INTERACTION**(RAC1) including objects $ActSeqRAC1_1$, $ActSeqRAC1_2$, …, $ActSeqRAC1_m$ is a full subcategory of **INTERACTION**(RAC1') possessing objects $ActSeqRAC1_1$, $ActSeqRAC1_1$**'**, $ActSeqRAC1_2$, …, $ActSeqRAC1_m$ and two categories are equivalent, which 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 4.9. Substitutability of RAE. RAE is isomorphic 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 as Social(RAE) ~ Social(RAE'); 3) they have equivalent internal structures when regarding them as two categories so that **CAT**(RAE) ~ **CAT**(RAE'); and 4) they have equivalent internal along with interactive behavior as **TRANSITION**(RAE)~**TRANSITION**(RAE'), **INTERACTION**(RAE)~**INTERACTION**(RAE'). If RAE≅ RAE', they can be substituted by each other in the RAS meta-model.

Because RACG is the minimum RAE which can independently fulfill a complete work in the RAS meta-model, an illustration of specifying the fault-tolerance property in RACG based on the simplified example in Figure 3 is discussed as follows.

Scenario1. Crashed RAO. After *sensor1* is started by *CU1*, it begins to send its heartbeat messages to *CU1* every *t* ticks (the tick is an abstraction of one time unit under a global clock), while *CU1* is in the first state of its intelligent control loop, monitoring the status of *sensor1*. If *CU1* receives the heartbeat messages from *sensor1* regularly, *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 a *Restart* message to *sensor1*. 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 isomorphic to *sensor1*, such as *sensor8*, since isomorphic objects behave in the same way (see Property 4.9). 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 and 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 Figures 9, 10, and 11).

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 isomorphic *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 isomorphic *production robot* of the original one, or the description before is applicable for remaining process.

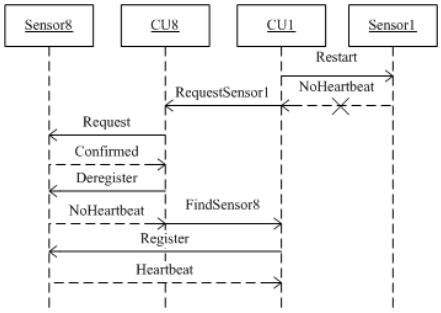


Figure 9. Sensor substitution work flow in production robot

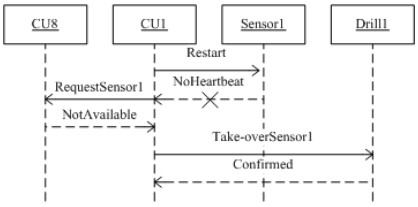


Figure 10. Sensor take-over work flow in production robot

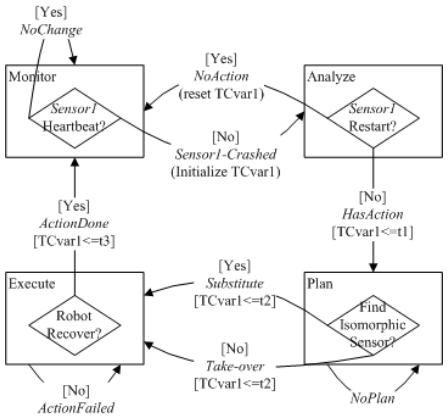


Figure 11. Intelligent control loop of control unit in production robot

Figure 12 depicts a categorical specification of *production robot1* before *snesor1* is crashed. The category **PR1** consists of three objects *drill1*, *sensor1*, and *CU1*. Thus, the bidirectional communications between those objects are six morphisms to specify the working collaboration between *drill1* and *sensor1* (*work1*, *work2*) as well as leadership among *CU1*, *drill1*, and *sensor1* (*order1*, *report1*, *order2*, *report2*). Synchronous communication between *drill1* and *sensor1* is specified as their product. Slice category models *Report* actions (*report1*, *report2*) with their relations (*work1*) from *drill1* and *sensor1* to *CU1*; coslice category interprets *Order* actions (*order1*, *order2*) with their relations (*work2*) from *CU1* to drill*1* and *sensor1*. If **PR1** is recovered by restarting crashed *drill1*, 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 *Restart* functor from **PR1** to **PR1-1**. If **PR1** is recovered by replacing *sensor1* with one of its isomorphic 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 13), but both of them have equivalent behavior, since *drill1'* works as the product object of original *drill1* and *sensor1* through its backup sensor and *drill1'* $\cong$ *drill1* $\times$ *sensor1*. 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. Therefore, a functor category consisting of those functors as objects with their natural transformations as morphisms models all possible evolutions and their relations.

CAT-RAC <PR1>

Objects: <Drill1, Sensor1, CU1>

Morphisms: <Work1(Drill1, Sensor1), Work2(Sensor1, Drill1), Report1(Drill1, CU1), Order1(CU1, Drill1), Report2(Sensor1, CU1), Order2(CU1, Sensor1)>

Limit: <designated behavior model of Drill1, Sensor1, and CU1>

Colimit: <actual behavior model of Drill1, Sensor1, and CU1>

Product Objects: <(Drill1, Sensor1)>

Coproduct Objects: <(Drill1, CU1), (Sensor1, CU1)>

Pushout Objects: <Collect1(Order1, Order2)>

Pullback Objects: <Trace1(Report1, Report2)>

Slice Category: <(Report1, Report2), Work1>

Coslice Category: <(Order1, Order2), Work2>

Functors: <Restart(PR1, PR1-1), Substitute(PR1, PR1-2), Take-over(PR1, PR1-3)>

Natural Transformations: <Convert1(Restart, Substitute), Convert2(Restart, Take-over), Convert3(Substitute, Take-over)>

Functor Category: <(Restart, Substitute, Take-over), Convert1, Convert2, Convert3>

End CAT-RAC

Figure 12. Categorical specification of production robot

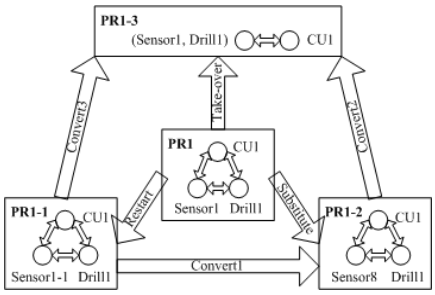


Figure 13. Evolution for self-healing due to crashed sensor in production robot

Scenario 2. Crashed RAOL. Similarly as scenario 1, after a *sentry robot* is started by its *supervisor robot*, it (*CU3*) begins to send heartbeat messages to *supervisor robot* (*CU4*) every *t* ticks. Figures 14 and 15 depict 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 isomorphic 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 isomorphic *production robot*, *carry robot* as well as *sentry robot* of the original ones; otherwise, the illustration before may indicate the remaining process. The evolution for self-healing due to crashed *control unit* in a *sentry robot* is depicted in Figure 16.

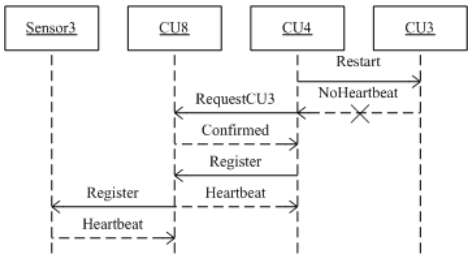


Figure 14. Control unit substitution work flow in sentry robot

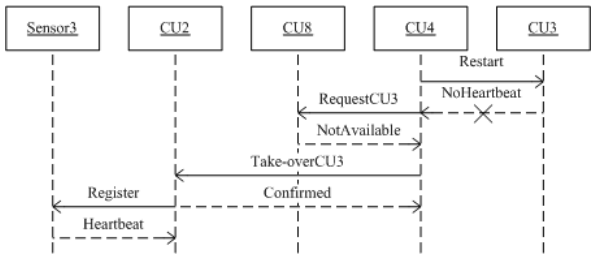


Figure 15. Control unit take-over work flow in sentry robot

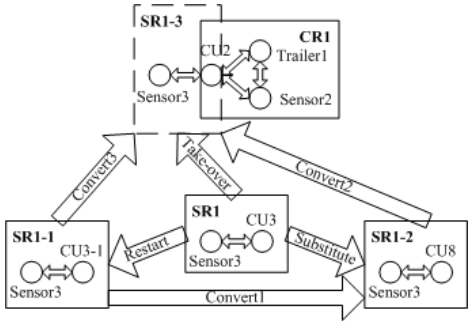


Figure 16. Evolution for self-healing due to crashed control unit in sentry robot

Scenario 3. Crashed RAC. Similarly as the scenario 1 and scenario 2 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 isomorphic *robot* of the original one (see Figure 17). 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 18, 19).

Figure 20 depicts a categorical specification of *exploration group1* before *carry robot1* is crashed. The category **EG1** has four objects *PR1*, *CR1*, *SR1*, and *Supervisor*. **RACG1** has three subcategories and their product categories when zooming in those objects and viewing them as the categories **CAT-PR1**, **CAT-CR1**, and **CAT-SR1** with their interactions. When **EG1** is recovered by restarting *CR1*, it evolves to **EG1-1** that has the same composition and categorical specification as **EG1** except for different initial status of *CR1*. When **EG1** is recovered by substituting *CR1* with its isomorphic object *CR2*, it evolves to **EG1-2** consisting of the same composition and categorical specification as **EG1** except for replacing *CR1* with *CR2*.

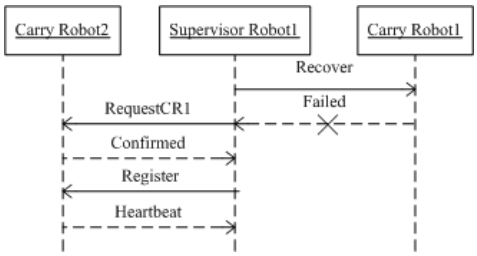


Figure 17. Carry robot substitution work flow in exploration group

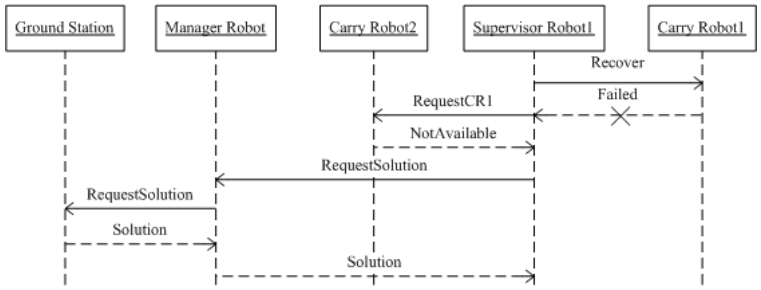


Figure 18. User intervention request work flow in exploration group

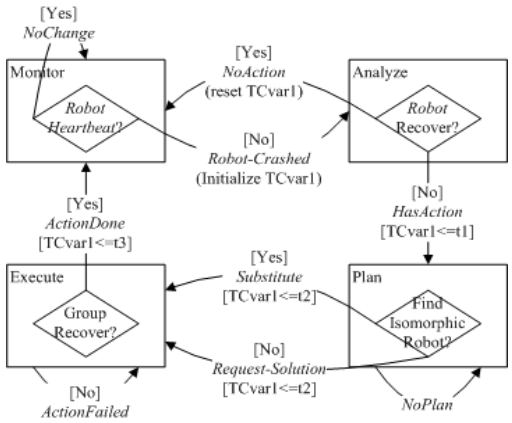


Figure 19. Intelligent control loop of supervisor robot in exploration group

CAT-RACG <EG1>
Objects: <PR1, CR1, SR1, Supervisor1>
Morphisms: <Work1(PR1, CR1), Work2(CR1, PR1), Work3(PR1, SR1), Work4(SR1, PR1),
Work5(CR1, SR1), Work6(SR1, CR1), Report1(PR1, Supervisor1),
Report2(CR1, Supervisor1), Report3(SR1, Supervisor1), Order1(Supervisor1, PR1),
Order2(Supervisor1, CR1), Order3(Supervisor1, SR1)>
Limit Object: <designated behavior model of PR1, CR1, SR1, and Supervisor1>
Colimit Object: <actual behavior model of PR1, CR1, SR1, and Supervisor1>
Product Objects: <(PR1, CR1), (PR1, SR1), (CR1, SR1)>
Coproduct Objects: <(PR1, Supervisor1), (CR1, Supervisor1), (SR1, Supervisor1)>
Pushout Objects: <Collect1(Order1, Order2), Collect2(Work4, Work6)>
Pullback Objects: <Trace1(Report1, Report2), Trace2(Work3, Work5)>
Slice Category: <(Report1, Report2, Report3), Work1, Work3, Work5>
Coslice Category: <(Order1, Order2, Order3), Work2, Work4, Work6>
Subcategories: <CAT-PR1, CAT-CR1, CAT-SR1, CAT-Supervisor>
Product Categories: <(CAT-PR1, CAT-CR1), (CAR-PR1, CAT-SR1), (CAT-CR1, CAT-SR1)>
Functors: <Restart(EG1, EG1-1), Substitute(EG1, EG1-2)>
Natural Transformations: <Convert(Restart, Substitute)>
Functor Category: <(Restart, Substitute), Convert>
End CAT-RACG

Figure 20. Categorical specification of exploration group

5. Implementation of Fault-Tolerance in Case Study

In this section, we will illustrate the implementation (using Jadex) of substitutability property for the fault-tolerance in our case study Mars-world[1]. As we explained before, we will focus on the substitutability of RAC (*robots*), since RACG (*exploration group*)

[1] The application can be executed using Jadex and downloaded from: http://users.encs.concordia.ca/~bentahar/RAS2MAS.jar

is the minimum RAE which can independently fulfill a complete work in RAS (*Mars-world*).

5.1. Mars-world as a MAS

In Mars-world, there are five types of agents (*robots*) as *Manager*, *Supervisor, Sentry*, *Producer* and *Carry* agents. A *Manager* agent can create and manage the *Supervisor* agents; it is a starting point of the Mars-world system and can interact with users. The graphical part of the system is implemented in *Manager* to provide a tool for human interactions. *Supervisor* is in charge of an exploration group to exploit ore mines. After it is created by *Manager*, *Supervisor* initiates a number of *Sentry* agents to search and analyze ore targets in the area assigned by *Manager*. If a target is found, *Supervisor* assigns the task of analyzing 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, *Sentry* requests some available *Producer* agents to exploit the target mine. After finishing the production work, *Producer* calls some available *Carry* agents to carry the produced ore to the *home base*. *Carry* has a limited capacity of ore so that it travels between the target mine and home base [11].

5.2. Fault-Tolerance in Mars-world

First, to simulate the malfunction of one agent, users can click on the agent in GUI that is considered as a signal to disable it (Figure 21). This is done inside the mouse click event listener of the environment panel in the *Mars-worldGUI* plan of the *Manager* agent. If the *x* and *y* of the clicked point falls inside any agent, it creates a message event that tells the agent to shutdown itself. For each agent, there is a *shutdown* plan which consists of taking a snapshot for the agent, pushing the snapshot into a queue in terms of retrieving it later by *Supervisor* for recovery, and shutting down the agent. The snapshot consists of agent snapshot, goal snapshot as well as message snapshot.

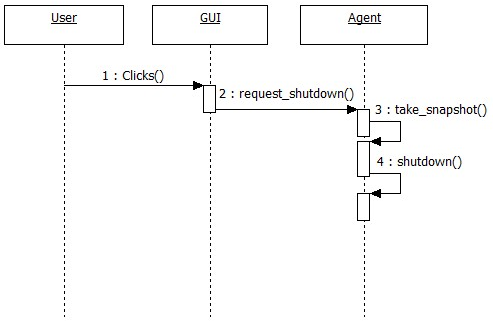


Figure 21. Sequence diagram of shutdown

The *Supervisor* agent has a perform goal named *check_agents* (Figure 22) that checks continuously the state of the agents in the environment belonging to their group. In the plan triggered by this goal, if *Supervisor* detects any inactive agent which type is

checked, and then it selects appropriate recovery plan for that type of agent and creates a top level goal for the recovery of the agent.

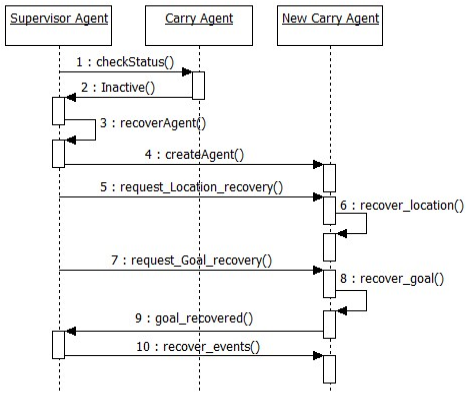


Figure 22. Carry recovery sequence diagram.

The *Supervisor* agent has a recovery plan for each of the group agents including the *Sentry* agent, the *Producer* agent, and the *Carry* agent. For example if a *Carry* agent is 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. For the moment we consider that we are able to create agents at any time that is necessary, but for the future we have to find the most appropriate agent which corresponds to the damaged one and we can consider it as a replacement; 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.

5.3. Agent creation

To create a new agent, the *ams_create_agent* goal of *amscap* capability of *AMS* agent is used. When an agent of a certain type is created, the static initial state of that agent type is recovered automatically. For example, any agent of the type *Carry* has a belief called *my_vision* having the value of *0.05* for all agents of this type. When a new agent of type *Carry* is created, that belief is initiated to *0.05* for it. This kind of information can be considered as static initial status of an agent and is recovered in agent creation phase. There is another kind of status information that is dynamic and is changing by the time, for instance, the location of an agent is dynamic since it is changing when the agent is moving around.

The recovery of dynamic status of agents is based on the status snapshots taken at shutdown moment of the agents. For example, the current location of the *Carry* agent is stored in an object of class *AgentSnapshot* that takes the current location of the agent in shutdown plan. In order to simulate the ongoing access of the *Supervisor* agent to the information of its group, there must be a way to inform it of the current status of its agents. For now the *AgentSnapshot* class contains some important dynamic information like location, the stack of goal snapshots and the name of the damaged agent. The name

of the damaged agent is kept since we will use it to access the previous message event queue to recover it.

5.4. Recovery of location using agent snapshot

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. This is because the goal recovery can be started without the recovery of the location, as the goal recovery will move the agent to a target or home base and the location recovery is very fast.

When the message event is received by the new *Carry* agent, it can trigger its *RecoverLocationPlan*, where, the agent is waiting for the *request_location_recovery* message and when receiving, 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 step of recovery the *carry_ore* goal is recovered because the latter has a higher priority to the former.

5.5. Current goal in hand

The current plan that the agent is pursuing must be recovered. For example, if a *Carry* agent has loaded ore and wants to deliver it to the home base, it is in the middle of the *carry_ore* plan. 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). For example, if the loaded ore is zero, it means that the *Carry* wants to move to target and reload ore; if it is greater than zero, it means that the *Carry* is moving from the target mine to the home base in terms of delivering loaded ore.

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.

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. For our case, the necessary data to recover the goal that can be found in the goal snapshot object consists of:

- Goal type: identifying the type of a goal to be recovered, such as *carry_ore*.
- Target location: identifying the target location from which the *Carry* agent carries ore to the home base.

- Ore load: indicating the ore amount loaded to the *Carry* agent. This variable can be used as an indicator to determine whether the *Carry* agent is carrying ore to the home base or is moving to the target mine for reloading ore.

In the *RecoverGoalPlan* plan, the *Carry* agent restores the target location and ore load from the goal snapshot. If the ore load is zero, it means that the *Carry* agent has to move to the target mine for reloading ore and carry it to the home base. Therefore, the starting point in this case will be moving to the target and reloading ore. If the ore load is greater than zero, it means that the damaged agent was carrying a certain amount of ore to the home base. In this case, the ore is loaded to the new *Carry* agent and then it moves to the home base for delivering ore. The *RecoverGoalPlan* is a special copy of *CarryOrePlan* with a facility of the conditional entrance points according to variable checkpoints. This task continues until all the ore in the target mine is carried to the home base. After finishing a goal, the *Carry* agent pops the goal from the stack. The reason that the goal is pop in this point is that if anything happens to the new *Carry* agent in the middle of the recovery process, we will keep the recovery snapshot record in the stack for another new agent to recover it.

After finishing this task, the *Carry* agent creates its *carry* goal that 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. If this *carry* goal is not started, the *Carry* agent cannot listen to and catch *request_carry* event messages.

In this point that the *Carry* agent is listening to *request_carry* message events, the *Supervisor* can start the recovery of the message events. Therefore, the *Carry* agent creates a reply message event named *reply_goal_recovery* in response to the message *request_goal_recovery* of the *Supervisor* agent. This action activates the recovery plan in the *Supervisor* side.

5.6. Message event queue recovery

Each agent has a message event queue that stores all incoming unprocessed message events for the agent. When a message event is received by an agent and it is doing another job so that it cannot process the message, the agent pushes the message in a queue and handles it later. When the agent is damaged, this message event queue must be recovered because it represents the assigned responsibilities of the agent.

When the *Supervisor* agent receives the reply for *request_goal_recovery* message event, it is sure that the goal has been recovered; thus, it starts to recover the message event queue of the damaged agent. The message events of each agent are stored in a snapshot queue corresponding to its unique ID. This message events snapshot queue is created inside the *ProductionPlan* plan of the *Production* agents. In this plan, after finishing the production task, the *Production* agent calls the existing *Carry* agents by creating and sending the *request_carry* message events to them. At the same time, the created *request_carry* message objects are pushed into the message event snapshot queue of each *Carry* agents.

On recovery, the *Supervisor* agent takes the message event snapshot queue and creates a message event for each element stored in the queue and sends it to the new *Carry* agent. More clearly, for the *request_carry* message event, the *Supervisor* agent restores the message snapshot from the queue, creates a corresponding message event and puts the restored *RequestCarry* object as its content and sends it to the new *Carry* agent. This operation is repeated for all elements of the message event queue snapshot,

and the message event queue has been copied to the new *Carry* agent's message event queue. By restoring the message event queue, the recovery task is completed and the agent can continue its normal process. Although there may be some message events assigned to the new agent which has not been processed yet, but they become to the responsibility of the new agent and will be handled with priority.

5.7. Replacement instead of creation

Until now in this case study, it is considered agents can be created at any time and easily, but in the real world a system has to take advantage of only the existing *robots* (agents) in the environment, and their availability has to be taken into account. For example, if a *Carry* agent crashes, the *Supervisor* agent has to find the most available *Carry* agent instead of simply creating a new *Carry* agent. This availability can be defined as the location and working status.

By considering those parameters, the *Supervisor* can select the proper agent to replace the damaged one and assign the recovery task to it. After choosing an agent, first of all, the *Supervisor* waits for that agent until it achieves any incomplete goal in hand. The selected agent can have any position thus the location recovery is different. The *Supervisor* may ask the selected *Carry* agent to move to the recovered location by creating *move_destination* goal in *RecoverLocationPlan* of that *Carry* agent; otherwise, it can ignore the location recovery and directly start the goal recovery. After the goal recovery and when the *Supervisor* receives the confirmation from the *Carry* agent, it can recover the message event queue. This replacement is possible because we can prove that the selected agent is isomorphic to the crashed one (substitutability: Property 4.9).

6. Related Work

The only published work on modeling autonomous systems using CT [12] served as the structure for our research. Its author defined a specification language of autonomous systems based on CT, but no systematic methodology and specific self-* properties, such as self-healing or fault-tolerance, were proposed in that paper. More recently, CT was applied to model the MAS, and a categorical model of MAS was stated in [13] as the **MAS** category. In [14], the authors focused on a construction of a transformation system for MAS based on categorical notions and previously introduced **MAS** category. There is also some related work regarding the formal specification of intelligent swarm systems as our case study Mars-world, which is one application of them. Authors in [15] illustrate a formal task-scheduling approach and model the self-scheduling behavior for Prospecting Asteroid Mission with an Autonomic System Specification Language.

Our research considerably differs from the related work above, since our goal is to propose a systematic and formal methodology based on CT to model and specify the reactive autonomic systems with self-* properties, which could be applied to model intelligent swarm systems and realized by MAS, service-oriented systems, or object-oriented systems.

7. Conclusions and Future Work

This paper introduced an important direction with respect to the formal aspects of specifying the fault-tolerance property in reactive autonomic systems by CT. Our work is motivated by the importance of compliance with self-management requirements for the increasingly complex MAS, which have unexpected emergent group behavior. Our RAS meta-model employing CT as a unified formal language allows the use of the same constructors to model objects and their relations recursively. We also showed that CT is expressive enough to capture system constructs and their interactions in a single formal representation, where structure and reasoning are bound together.

We are currently working on a formal specification of the Categorical Modeling Language (CML), which can be used to present the categorical specification and self-* properties, such as the fault-tolerance and substitutability property in this paper for MAS, and corresponding graphical tool to capture system modeling. Once the RAS meta-model has been developed, we will transfer it to the MAS model. Eventually, a source code template can be generated according to the MAS model, and this will be discussed in our future work.

References

[1] 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*, Lisbon, Portugal, September 2005, 125-133.

[2] 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.

[3] O. Ormandjieva and J. Quiroz, Methodology for automatic generation of exhaustive behavioral models in reactive autonomic systems, *Proceedings of the International Conference on Software Engineering Theory and Practice*, Orlando, Florida, USA, July 2008.

[4] H. Kuang and O. Ormandjieva, Self-monitoring of non-functional requirements in reactive autonomic system framework: a multi-agent systems approach, *Proceedings of the 3rd International Multi-Conference on Computing in the Global Information Technology*, Athens, Greece, July 2008, 186 – 192.

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

[6] 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, 991 – 999.

[7] 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, 12-21.

[8] J. L. Fiadeiro and T. Maibaum, A Mathematical Toolbox for the Software Architect, *Proceedings of the 8th International Workshop on Software Specification and Design*, Schloss Velen, Germany, March 1996, 46 – 55.

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

[10] S. Mac Lane, Categories for the Working Mathematician, 2nd Edition, Springer, September 1998.

[11] http://jadex.informatik.uni-hamburg.de/xwiki/bin/view/About/Overview

[12] W. M. Lee, Modelling and Specification of Autonomous Systems using Category Theory, *PhD Thesis*, University College of London, London, UK, October 1989.

[13] J. Pfalzgraf, On Categorical and Logical Modeling in Multiagent Systems, *Anticipative and Predictive Models in Systems Science*, Volume 1, IIAS, Windsor, ON, Canada, 2005, 93 – 98.

[14] J. Pfalzgraf, T. Soboll, On a General Notion of Transformation for Multiagent Systems, *Proceedings of the 10th World Conference on Integrated Design & Process Techonology*, Antalya, Turkey, June 2007.

[15] 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*, Belfast, Northern Ireland, March 2008, 54-64.