ISSN No:-2456-2165

A Categorical Approach to Model Formation of Reactive Autonomic Systems Framework

Ming Zhu
College of Computer Science and Technology
Shandong University of Technology
Zibo, China

Heng Kuang
Huawei Canada
Markham, Canada

Abstract:- Software complexity crisis becomes an impediment to further development of software. Specifically, in order to manage increasingly complex and massive software systems, researchers involve in building systems with autonomy. The real-time reactive systems with autonomic behaviors could be more self-managed and more adaptive to the environment. However, formations of some of such systems are not formalized, which may lead systems to be error-prone. In this research, we proposed a formal way to describe formations of reactive autonomic systems framework. Firstly, we introduce how to from reactive autonomic system, components group, and component, then we focused on categorizing the formations. To do so, the basis of reactive autonomic systems can be built with correct by construction.

Keywords:- *Reactive Autonomic System; Category Theory; Formation*

I. INTRODUCTION

Software complexity crisis has been deemed as one of the major obstacles to the progress in software industry, since the management of computing systems with complexity becomes difficult for IT Practitioners. We need to select a target system that can get benefit from applying the autonomic computing paradigm. Usually, real-time reactive systems are considered as one of the most complex systems; the complexity involved comes from their real-time as well as reactive characteristics: 1) concurrency is involved; 2) timing requirements are strict; 3) reliability is a must; 4) software and hardware components are involved; 5) it becomes more and more intelligent and heterogeneous. So, we need to add autonomic features into real-time reactive systems by using Reactive Autonomic Systems Framework (RASF), which helps to specify, model and develop the Reactive Autonomic Systems (RAS). By adding autonomic behaviors, real-time reactive systems become more self-managed and adaptive; the RAS can improve and simplify users' experiences. By using input data and checking results only, it is difficult to find the competitive conditions in the real-time reactive system, because some errors may only occur when the process sends or receives data at a specific time. In order to detect these errors through testing, every state combination of processes must be checked, which may result in an exponential number of states [1]. Formal methods have been proven to be the way to ensure the correctness of operation in complex interactive systems, because formal specifications are proved that it can help to check specific types of errors, and it can also be used as input for model checking [1].

To continue the research [2][3], we proposed an approach to model reactive autonomic system, component group and components in this paper. The rest part of the paper is presented as follows: Section 2 sketches the work related to the research. Section 3 introduces the background knowledge needed to understand the paper. Section 4 introduces formation of reactive autonomic system and its categorical models. Section 5 describes formation of reactive autonomic component group and its categorical models. Section 6 explains formation of reactive autonomic component and its categorical models. In the last, conclusion and future work are provided in Section 7.

II. RELATED WORK

This section introduces related research work to the paper.

A. Real-Time Reactive Systems

In research [4], Caporuscio specifies that there is a trend to switch assembling components into systems to composing autonomous systems into systems-of-systems dynamically, as these kinds of system usually run in highly dynamic environments. In research [5], an approach to solve the problem of synchronous programs that cannot be executed in a time-triggered or event-triggered execution loop easily. By using dynamic tickets method, semantic timing abstraction of the synchronous approach can be reconciled that can help to give the application fine-grained control over its real-time behavior.

B. Formal methods

Formal method can be used to assess risks at the early stages of developing security-critical real-time systems in research [6], where a formal model named Object-Message-Role (OMR) with Z notation is proposed to specify functional and security aspects of systems. Research [7] defined a model using formal method to evaluate the quality of system architecture. By formalizing characteristics of system architecture, good quality attributes can be quantified and can be used in may system architecture tools. In research [8], probabilities are combined with a formal approach to develop safety-critical systems. In this approach, model-driven engineering is used for reactive systems and a tool-set reactive blocks extend the support of modeling and verification of behaviors in real-time systems. In paper [9], instead of reviewing requirements manually, a formal scenario-based method based on LSCs/MSDs are proposed, which support automated analysis by formal realization checking and scenario execution. In research [10], based on a formal language called Biographical Reactive System, the correctness of deploying architecture is able to be guaranteed. By using a multi-scale modelling, automatic construction is supported.

III. BACKGROUND

In this section, background and work related to our research are introduced.

A. *Reactive Autonomic Systems (RAS)*

The framework of Reactive Autonomous System (RASF) shown in Fig.1 consists of four tiers, which include Reactive Autonomic Systems, Reactive Autonomic Component Groups, Reactive Autonomic Components and Reactive Autonomic Objects. In this hierarchical model, each layer only exchange messages with its upper or lower layers. Therefore, the independence of these layers makes it possible to modularize, encapsulate, decompose and reuse them.

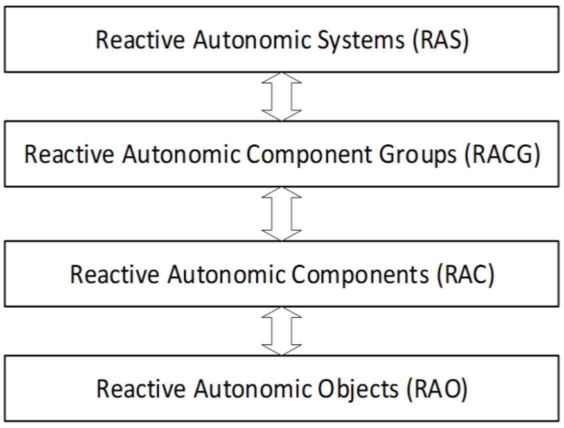
Reactive Autonomic Systems (RAS)
Reactive Autonomic Component Groups (RACG)
Reactive Autonomic Components (RAC)
Reactive Autonomic Objects (RAO)

Fig. 1:- RASF Architecture Model

- RAO is considered as a labeled transformation system [11], which is specified as tuples ($P,\varepsilon,\Theta,X,L,\Phi,\Lambda,\gamma,R$) as follows:

  - P consists of a set of ports, where, for each type of port and type of null, there exist only port P_0.
  - ε consists of a set of events, where silence events are included.
  - Θ consists of a set of states, where Θ_0 indicates the initial states, and final state doesn't exist.
  - X consists of a set of attributes with types. There exists two types, port reference types and abstract data types.
  - L consists of a set of traits, which is used to indicate the abstract data type in X.
  - Φ indicates a pair (Φ_s,Φ_{at}) to denote function-vector relation, where Φ_s attaches a set of sub states to each Θ and Φ_{at} attaches a set of attributes to each Θ.
  - Λ indicates a set of specifications used in transitions.
  - γ indicates a set of constraints to time.
  - R indicates the set of available resources used locally to support object's functionality.

- RAC indicates a group of RAOS that communicate synchronously, one of which is assigned to the remaining workers as a leader, which is usually referred to as RAOL. Workers take care of reactive tasks, while RAOL is responsible for autonomous tasks, such as coordinating and managing component's self-monitoring. Therefore, RAOL is different from workers by using a different set of states. These states are related to autonomic behavior. The reactivity and autonomy of RAOL, which is specified formally, provide a means for them to gain functionalities with autonomic characteristics in reactive systems with real-time property. In order to coordinate the performance of tasks and messages exchanging between RAOs, the specification of RAC specification includes *members*, *configurations*, *leaders*, *supervisors*, *repositories* and *neighbors*. RAC is the smallest centralized reactive autonomous element (RAE) with self-management ability.

  RAC's reactive behavior includes n collaborations. RAO is modeled as tuples (P^s,ε^s,Θ^s,X^s,L^s,Φ^s,Λ^s,γ^s,R^s) [12] as follows:

  - P^s contains a set of types of port, which allows a synchronous message passing between Reactive Autonomic objects.
  - ε^s indicates a union of ε_i where $1 \leq in$.
  - Θ^s contains a set of states which are used for valid Synchronous Production Machine.
  - X^s indicates a union of sets X_i^{syn}, where $1 \leq in$.
  - L^s indicates a union of sets of traits, which are used for Abstract Data Type and specified by Larch Specification Language.
  - Φ^s indicates a triple (Φ_s^s,Φ_{at}^s, Φ_γ^s) to denote function-vector relationship, where Φ_s^s attaches a set of substates to each Θ^s, Φ_{at}^{syn} attaches a set of attributes $\Phi_{at1}(\Theta_1^s)$,…, $\Phi_{atn}(\Theta_n^s)$ to each Θ^s, and Φ_γ^s attach a subset of R^{syn} to each Θ^{syn}.
  - Λ^s contains a set of specifications used for transitions.
  - γ^s contains a set of constraints to time.
  - R^s indicates the set of available resources in RAO, which is represented as a union of R_i, where $1 \leq in$.

- RACG is a group of RAC, which are either distributed or centralized. RAC completes tasks that grouped together through synchronous message passing and cooperation between each other. It is considered as the smallest RAE in RASF that can perform tasks of real-time reactive systems independently. RACS is a supervisor, which coordinate and manage autonomic behaviors at group level.

- RAS consists of a group of RACG, which are either distributed or centralized with asynchronous message passing. It acts as an interface that is integrated for user to delegate tasks for computing, monitoring and managing. A manager of RACG (RACGM) has the main responsibilities to coordinate autonomic behaviors at system level.

B. *Category Theory*

Category theory is considered to have a rich theoretical knowledge to explain objects and relations between objects. It is an abstract framework that is able to be applied to many kinds of languages used for specifications [12]. For software specification, category theory provides a means to construct specifications with properties of correctness, which can help to prove the attributes and relationships maintained during the constructions of different stages [12]. In addition, by equipped with operations and diagrams of reasoning, hierarchical structures of complex systems are able to be formed as components used in other complex systems, and infer system attributes based on their configuration [13].
Since the self-configuration of RAS does not have such formalization, we suggest to achieve the formalization of self-configuration by category theory. In order to learn this paper, we present the basic definitions of category as follows:

Definition 1: A category includes morphisms and objects. Given object A, B and C, if there exists morphisms f: $A \rightarrow B$ and g: $B \rightarrow C$, then there must have a composition morphism $g \circ f : A \rightarrow C$. For the composition, an associativity: $(h \circ g) \circ f = h \circ (g \circ f)$ exists. For each object X, there exists an identity morphism Id_X, and for morphism $f : A \rightarrow B$, $f \circ Id_A = f = Id_B \circ f$.

Definition 2: A subcategory *Sub* of category *Cat* represents by a collection of objects which is a subcollection of *Cat*'s objects, indicated by *obj*(*Sub*), and a collection of morphisms which is a subcollection of *Cat*'s morphisms, indicated by *mor*(*Sub*). For each X in *obj*(*Sub*), there exist an identity morphism Id_X in *mor*(*Sub*). For each morphism f: $X \rightarrow Y$ in *mor*(*Sub*), and morphism g: $Y \rightarrow Z$ in *mor*(*Sub*), there exists a composition $f \circ g$ is in *mor*(*Sub*).

IV. RAS FORMATION AND CATEGORICAL MODELS

In this section, an approach to form RAS is proposed, and categorical models for RAS are illustrated.

A. *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. Fig.2 depicts an example of forming categories RAS1 and RAS2 from their index category RAS Formation.

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 represents abstracted one time unit within **RAS1**), while *RACGM1* is in the initial state of its intelligent control loop for monitoring. If the configuration of those *RACS* conforms to their types, composition rules and communication protocols, *NoViolation* event keeps *RACGM1* in the state of *Monitor*; otherwise, *NeedInvestigation* event is triggered and *RACGM1* transits to state *Analyze*, while a constraint to time variable (*TCvar1*) is initialized to act as a local clock in terms of constraints to time on every transition of the intelligent control loop. The value of *TCvar1* is *t0*, *t1*, *t2*, *t3*... where *t0* < *t1* < *t2* < *t3*.

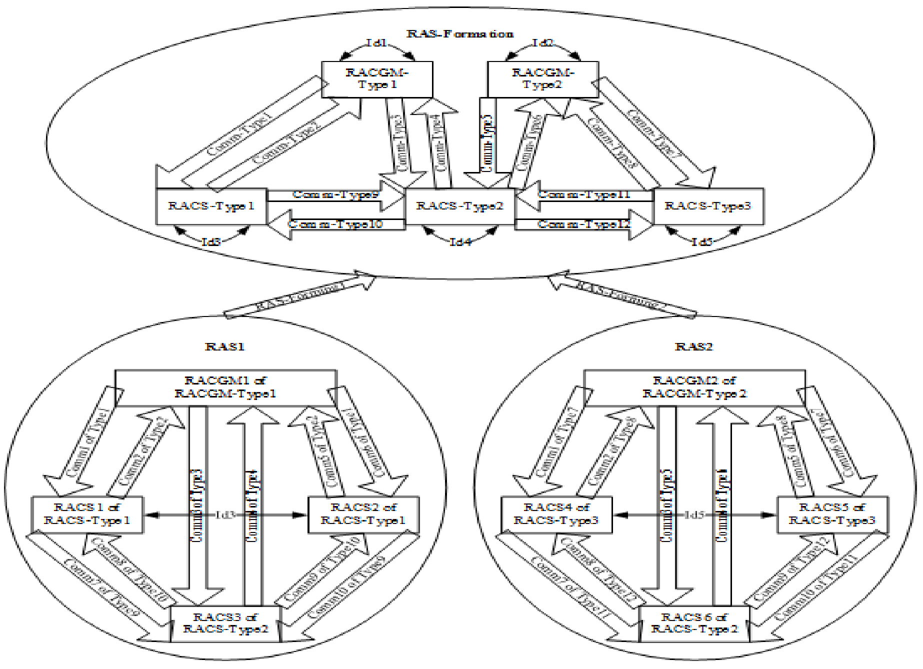

Fig. 2:- Example of Forming RAS from RAS-Formation

After *RACGM1* goes in state *Analyze*, 1) it sends a message *Restart* 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, event *NoAction* occurs and *RACGM1* returns back to state *Monitor*, while the *TCvar1* is reset; otherwise, *RACGM1* moves to state *Plan* triggered by event *LaunchSelfHealing* in *t1*. 2) If the violation is caused by incorrect communication type from other RACS (*RACS3*) to *RACS1*, *RACGM1* sends a message *Restart* to *RACS3*. If the communication conforms to its type, event *NoAction* occurs and *RACGM1* returns back to state *Monitor*, while the *TCvar1* is reset; otherwise, *RACGM1* moves to state *Plan* triggered by the event *LaunchSelfHealing* within *t1*. 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**, event *NoAction* occurs and *RACGM1* returns back to the state *Monitor*, while the *TCvar1* is reset; otherwise, *RACGM1* moves to state *Plan* triggered by the event *LaunchSelfHealing* in *t1*.

When *RACGM1* is in state *Plan*, it chooses either *Substitute* plan or *Take-over* plan, based on the availability of substitutable RACS for *RACS1* or for *RACS3*. *RACGM1* moves to state *Execute* triggered by the event *Substitute* or event *Take-over* respectively in *t2*. *RACGM1* sends a message *selfViolation* to *User Console*, and the latter chooses either *Substitute* plan or *Take-over* plan based on the availability of substitutable RACGM for *RACGM1*. It moves to state *Execute* triggered by event *Take-over* or *Substitute* in *t2*.

When *RACGM1* is in state *Execute* and *Substitute* plan is available, it sends a message *register* to the substitutable RACS of *RACS1* or *RACS3* and initialize it to the status of *RACS1* or *RACS3* according to the previously made checkpoint. When the plan *take-over* is available, *RACGM1* sends a message *take-over* to *RACS2* and update the status of the synchronous product machine of *RACS1* and *RACS2*, or *RACS3* and *RACS2* d on the checkpoint. After the plan execution of plan, *RACGM1* validates the configuration of **RAS1**', an evolution of **RAS1** against its index category **RAS-Formation** according to their categorical specifications. If that configuration conforms to the index category, event *ActionDone* occurs and *RACGM1* moves to the *Monitor* state in *t3*; otherwise, event *ActionFailed* stays it in state *Execute* for the user intervention from the *User Console*.

B. Categorical Models 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. 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.

The transitions in the intelligent control loop of RACGM1 for self-configuration are specified as a category where 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. For example, *NeedInvestigation* = (*Monitor*, *NotConform-RACS*, *Analyze*); the sequences of those transitions are specified as a category where objects are those sequences (<*NoViolation*, *NeedInvestigation*, *RestartRACS1*, *NoAction*>, <*RestartRACS1*, *LaunchSelfHealing*, *Substitute*, *ActionDone*>), and morphisms are *equivalence* relations between those sequences.

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 confirmed to **RAS-Formation** by substituting the *RACS1* or *RACS3* with their isomorphic objects *RACS7* or *RACS9*, 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 querying *RACS2* to acquire 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 Fig. 3).

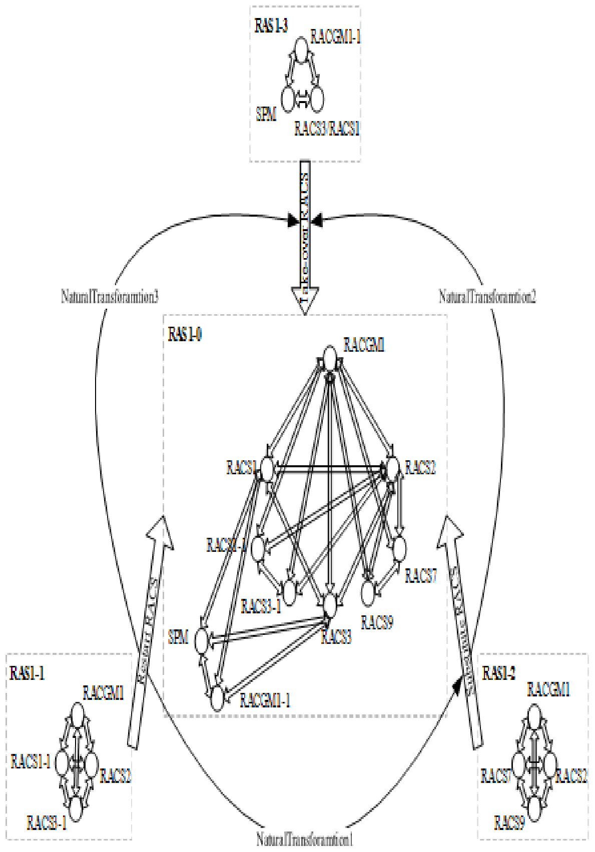

Fig 3:- Evolution for Self-Configuration in RAS1

V. RACG FORMATION AND CATEGORICAL MODELS

In this section, an approach to form RACG is proposed, and categorical models for RACG are illustrated.

A. 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**. Fig.4 illustrates an example of forming the categories **RACG1** and **RACG2** from their index category **RACG-Formation**.

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 each *t* (a tick represents abstracted one time unit within **RACG1**), while *RACS1* is in the initial state of its intelligent control loop for monitoring. If the configuration of those *RAOL* conforms to their types, composition rules as well as communication protocols, event *NoViolation* keeps *RACS1* in state *Monitor*; otherwise, event *NeedInvestigation* is triggered and *RACS1* transits to state *Analyze*, while a time constraint variable (*TCvar2*) is initialized to work as a local clock in terms of constraints to time on every transition of the control loop. The value of *TCvar2* is *t0*, *t1*, *t2*, *t3*... where *t0* < *t1* < *t2* < *t3*.

After *RACS1* goes in state *Analyze*, 1) it sends a message *Restart* 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, event *NoAction* occurs and *RACS1* goes back to state *Monitor*, while the *TCvar2* is reset; otherwise, *RACS1* moves to state *Plan* triggered by event *LaunchSelfHealing* 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, event *NoAction* occurs and *RACS1* returns back to state *Monitor*, while the *TCvar2* is reset; otherwise, *RACS1* moves to state *Plan* triggered by the event *LaunchSelfHealing* within *t4*. 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**, event *NoAction* occurs and *RACS1* returns back to the state *Monitor*, while the *TCvar2* is reset; otherwise, *RACS1* moves to state *Plan* triggered by event *LaunchSelfHealing* in *t4*.

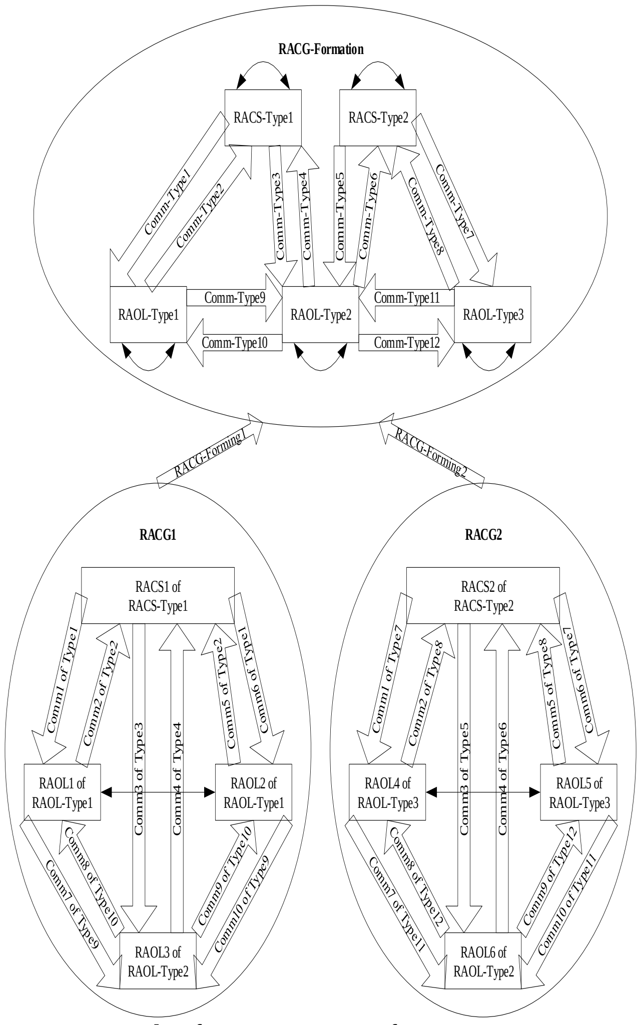

Fig 4:- Example of Forming RACG from RACG-Formation

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

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

B. Categorical Models 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. For example, *LaunchInvestigation* = (*RACS1*, *NotConfrom-RAOL*, *InvestigateRAOL*, *RAOL1*), and the sequences of those actions are specified as the categories where objects are those sequences (<*InitializeRACS*, Heartbeat, *InitializeRAOL*, *Heartbeat*>, <*ValidateRACS*, *Conform*, *ValidateRAOL*, *NotConform*>), and morphisms are the *equivalence* relationship between those sequences.

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. 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.

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 functor *Restart* (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*, 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 querying *RAOL2* to acquire 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 Fig.5).

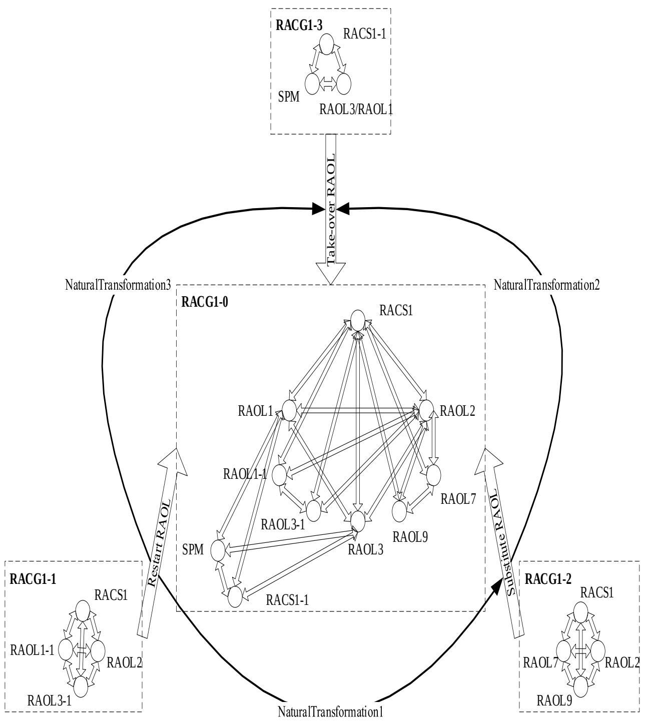

Fig 5:- Evolution for Self-Configuration in RACG1

VI. RAC FORMATION AND CATEGORICAL MODELS

In this section, an approach to form RAC is proposed, and categorical models for RAC are illustrated.

A. *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. Fig.6 depicts an example of forming the categories RAC1 and RAC2 from their index category RAC-Formation.

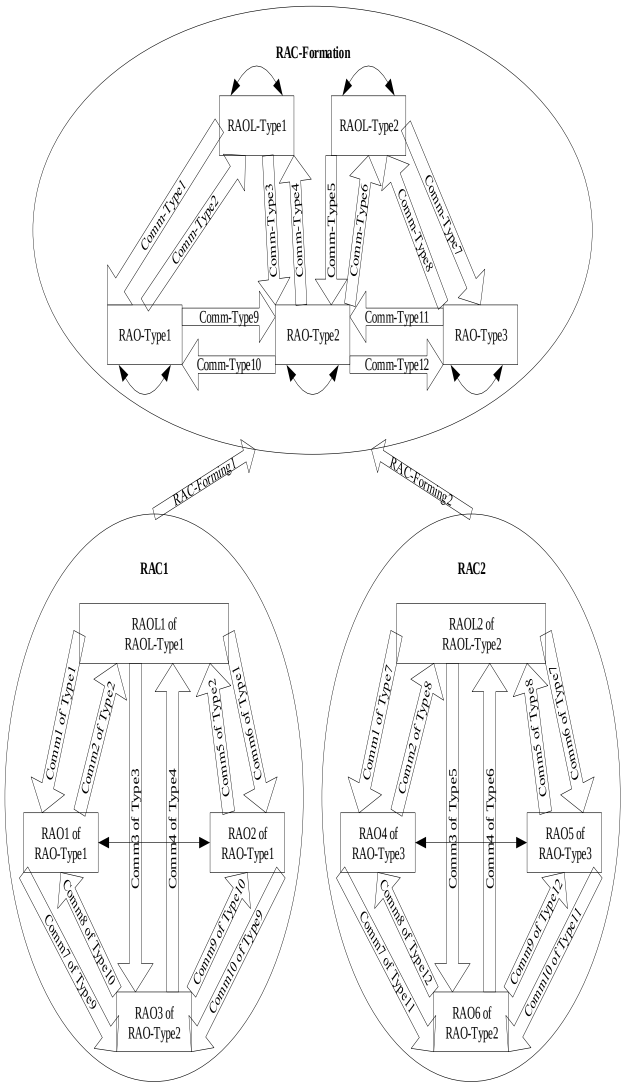

Fig 6:- Example of Forming RAC from RAC-Formation

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 each *t* ticks (a tick represents abstracted one time unit within **RAC1**), while *RAOL1* is in the initial state of its intelligent control loop for monitoring. If the configuration of those *RAO* conforms to their types, composition rules as well as communication protocols, event *NoViolation* keeps the *RAOL1* in state *Monitor*; otherwise, event *NeedInvestigation* is triggered and *RAOL1* moves to state *Analyze*, while a constraint to time variable (*TCvar3*) is initialized to work as a local clock in terms of time constraints on every transition of the intelligent control loop. The value of *TCvar2* is *t0*, *t1*, *t2*, *t3*... where $t0 < t1 < t2 < t3$.

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

When *RAOL1* is in state *Plan*, it chooses either plan *Substitute* or plan *Take-over*, based on the availability of substitutable RAO for *RAO1* or for *RAO3*. *RAOL1* moves state to *Execute* triggered by the event *Substitute* or event *Take-over* respectively in *t8*. *RAOL1* sends a message *selfViolation* to *RACS1*, and the latter chooses either plane *Substitute* or plan *Take- over* according to the availability of substitutable RAOL for *RAOL1*. It moves to state *Execute* triggered by event *Substitute* or *Take-over* in *t8*.
When *RAOL1* is in state *Execute* and plan *Substitute* is available, it sends a message *register* to the substitutable RAO of *RAO1* or *RAO3* and then initialize it to the status of *RAO1* or *RAO3* based on the previously made checkpoint. When the plan *take-over* is available, *RAOL1* sends a message *take-over* to *RAO2* and update 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** against the index category **RAC-Formation** based on their categorical specifications. If that configuration conforms to the index category, event *ActionDone* occurs and *RAOL1* moves to the state *Monitor* within *t9*; otherwise, event *ActionFailed* keeps it in state *Execute* for *RACS1*'s intervention.

B. *Categorical Models 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. 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.

The transitions in the intelligent control loop of RAOL1 for self-configuration are 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. 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.

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 structure as **RAC1** except for the different initial status of *RAO1* or *RAO3*. This process 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*, 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 querying *RAO2* to acquire 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 Fig.7).

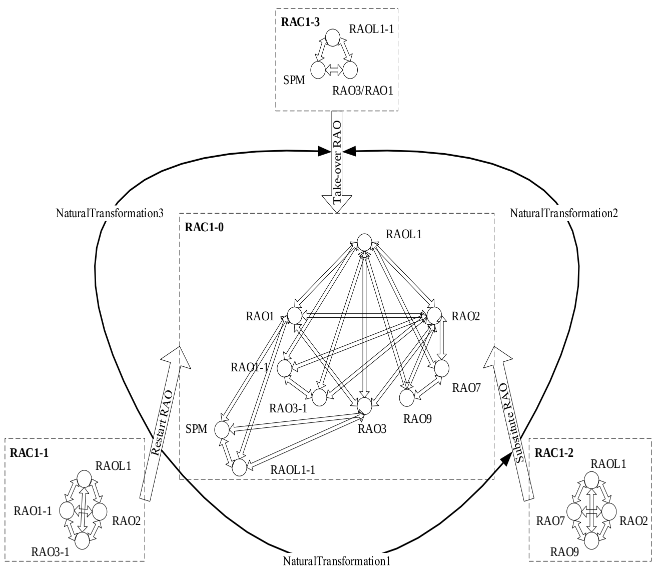

Fig 7:- Evolution for Self-Configuration in RAC1

VII. CONCLUSION

To provide the formation of reactive autonomic system with correct by construction, in this research, we suggested to use categorical means to model formation of reactive au-tonomic system framework. We described three scenarios regarding the self-configuration that are forming a RAS, forming a RACG and forming a RAC using intelligent control loops. In addition, we presented the categorical illustration for the for-mations respectively. Through the process of modeling and construction, category theory is able to provide formalization to the formation of reactive autonomic system frame-work. In future, we will work toward analyzing Evolution for Communication Self-Configuration in the RAS, RACG and RAC in the framework.

REFERENCES

[1]. Rash J. L., Hinchey M. G., Rouff C. A. and Truszkowski W. F. (2005). Requirements of an integrated formal method for intelligent swarms. *Proceedings of the 10th International Workshop on Formal Methods for Industrial Critical Systems*, Lisbon, 5-6 Sept 2005, 125–133.

[2]. Zhu, M. and Li, J. (2018) Towards a Categorical Framework for Verifying Design and Implementation of Concurrent Systems. Journal of Computer and Communications, 6, 2018, 227-246.

[3]. Zhu, M. and Li, J. (2019) Using Category Theory to Explore and Model Label Event Structures. Journal of Computer and Communications, 7, 2019, 49-60.

[4]. Caporuscio, M. (2017) Towards Fully Decentralized Self-Adaptive Reactive Systems. *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering Companion*, L'Aquila, 22-26 April 2017, 17-17.

[5]. Hanxleden, R.V., Bourke, T. and Girault A. (2017) Real-time Ticks for Synchronous Programming. *Proceedings of 2017 Forum on*

Specification and Design Languages (FDL), Verona, 18-20 September 2017, 1-8.

[6]. Ni, S., Zhuang Y., Gu, J. and Huo, Y. (2016) A formal model and risk assessment method for security-critical real-time embedded systems. Comput. Secur., 58: C, 2016, 199-215.

[7]. Rodano, M. and Giammarco, K. (2013) A Formal Method for Evaluation of a Modeled System Architecture, Procedia Computer Science, 20, 2013, 210-215.

[8]. Han, F., Blech, J.O., Herrmann, P. and Schmidt, H. (2014) Towards Verifying Safety Properties of Real-Time Probabilistic Systems. *Proceedings of Formal Engineering Approaches to Software Components and Architectures*, Grenoble, 05-13 April 2014, 1-15.

[9]. Greenyer, J., Haase, M., Marhenke, J. and Bellmer, R. (2015) Evaluating a Formal Scenario-based Method for the Requirements Analysis in Automotive Software Engineering, *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering,* Bergamo, 1-4 September 2015, 1002-1005

[10]. Gassara, A.， Rodriguez, I.B. and Jmaiel, M. (2015) A multi-scale modeling approach for software architecture deployment, *Proceedings of the 30th Annual ACM Symposium on Applied Computing*, Salamanca, 13-17 April, 2015, 1405-1410.

[11]. Ormandjieva, O., Bentahar, J., Huang, J. and Kuang, H. (2015) Modelling Multi-agent Systems with Category Theory. Procedia Computer Science, 52, 2015, 538-545.

[12]. Barr, M. and Wells, C. (2012) Category Theory for Computing Science. Prentice-Hall, Upper Saddle River.

[13]. Goubault, E. and Mimram, S. (2010) Formal Relationships between Geometrical and Classical Models for Concurrency. *Electronic Notes in Theoretical Computer Science*, 283, 77-109.