Architecture for Reactive Autonomic Systems: AS-TRM Approach

Heng Kuang

A Thesis

in

The Department

of

Computer Science and Software Engineering

Presented in Partial Fulfillment of the Requirements

for the Degree of Master of Computer Science at Concordia University

Montreal, Quebec, Canada

March 2006

Your file Votre référence
ISBN: 0-494-14326-6
Our file Notre référence
ISBN: 0-494-14326-6

Abstract

Architecture for Reactive Autonomic System: AS-TRM Approach

Heng Kuang

Real-time reactive systems are some of the most complex systems, so the modeling and development of real-time reactive systems becomes a very challenging and difficult task. The TROMLAB framework makes this modeling and development easier and more rigorous by integrating the Timed Reactive Object Model (TROM) formalism with a practical development methodology. However, current TROM formalism does not have the appropriate mechanism for specifying distributed autonomic reactive systems, which systems can simplify and enhance the end-users experience by anticipating their needs in a complex, dynamic, and uncertain environment.

This thesis is the first step towards extending TROM to Autonomic Systems Timed Reactive Model (AS-TRM) for modeling and supporting distributed, autonomic, and reactive behavior. The thesis presents the following: 1) extending TROM to AS-TRM for supporting distributed autonomic behavior; 2) defining AS-TRM's characteristics for determining AS-TRM systems' requirements, design, and implementation; 3) building AS-TRM's architecture and communication mechanism for implementing both autonomic and real-time reactive functionalities; 4) modeling AS-TRM's reliability assessment for monitoring AS-TRM's evolution.

To my grandfather.

Acknowledgments

I really appreciate many people who help with my thesis work and provide valuable support.

First, I would like to express my profound thanks to my esteemed supervisor, Dr. Olga Ormandjieva, for her guidance, help, and support through the stages of this work as well as my graduate study. Her good technical and financial support motivates me to work hard and produce a high quality result.

I would like to thank all the members of TROMLAB and AS-TRM team for their comments and support; a special thank goes to Emil Vassev, his supervisor Dr. Joey Paquet, and Irina Croitoru.

In addition, a warm thank you goes to my examiner, Dr. Vangalur Alagar and Dr. Juergen Rilling, for their precious time to review my thesis and give me helpful advice.

I also would like to thank Concordia University and Faculty of Computer Science for offering me the precious opportunity and excellent academic environment to achieve this work.

Finally, I would like to thank all my family members for their encouragement and never ending support.

Table of Contents

List of Figures

Chapter 1: Introduction

This thesis presents the results of the research and practical work made towards the nature of reactive autonomic systems, definition and implementation of architecture for them, and monitoring their reliability. This thesis lays the grounds for the Autonomic Systems Timed Reactive Model (AS-TRM) project, which provides rigorous development and quality monitoring of reactive autonomic systems. This research builds upon existing TROMLAB framework [AAM96], and extends an architectural model developed in the GIPSY project [PK00] as the prototype of AS-TRM Communication System.

1.1 Context

An Autonomic System is a system that can regulate itself much in the same way as our autonomic nervous system regulates and protect our bodies. The autonomic system's components and the system as a whole should be capable of anticipating computer system needs and resolving problems when they appear, without human intervention. Examples of such systems are: IBM Trivoli Management Suite, SUN Microsystems – N1, Hewlett-Packard's Adaptive Enterprise, and Microsoft's Dynamic Systems Initiative.

A reactive system is a system that continuously interacts with its environment through stimulus and response. The reactive system's stimulus-response behavior is regulated by strict time constraints which concern real-time aspects. Examples of such systems are: workshop automation like robotics, strategic defense systems like nuclear power plants, and transportation like railroad-crossings. Generally, a reactive system's

behavior is infinite and must satisfy two important requirements:

- Stimulus synchronization: processes are always able to react to a stimulus from environment;
- Response synchronization: the time-lapse between a stimulus and its response is acceptable to environmental relative dynamics so that the environment is still receptive to the response.

1.2 Research Goals

Real-time reactive systems are some of the most complex systems. The complexity involved comes from their real-time and reactive core characteristics: 1) involves concurrency; 2) have strict timing requirements; 3) must be reliable; 4) involves software and hardware components. Thus, the modeling and development of real-time reactive systems becomes a very challenging and difficult task. The TROMLAB framework makes this modeling and development easier and more rigorous by integrating the TROM formalism with a practical development methodology; moreover, the TROMLAB framework has an easy interface for modular design of system components, and this framework is also a formal basis for rigorous analysis.

However, current TROM formalism does not have an appropriate mechanism to specify distributed autonomic reactive systems, which systems are faster than regular reactive systems to adapt environmental changes. In addition, a distributed autonomic reactive system can simplify and enhance the end-users' experience by anticipating their needs in a complex, dynamic, and uncertain environment, which qualities are key

characteristics of a real-time environment.

Furthermore, recent real-time reactive systems have become increasingly heterogeneous and increasingly intelligent; therefore, we need to extend the TROMLAB framework to capture those new aspects. One of the ways to remove the complexity barrier is to model and develop complex computer systems that are autonomic. Autonomic computing is a new research area led by IBM Corporation, which area concentrates on making complex computing systems smarter and easier to manage. However, according to our knowledge, autonomic computing technology has not been applied to model and develop real-time reactive systems, which systems have high demand for autonomic computing technology to remove the complexity of modeling and development. With autonomic behavior, real-time reactive systems will be more self-managed to themselves and more adaptive to their environment.

As a result, our research work's objective is to extend the TROMLAB framework to have distributed autonomic behavior by adding two aspects: 1) the specification of distributed autonomic reactive components along with their relationships; 2) the non-functional properties constraining systems' behavior, such as reliability. In order to achieve our research goal, we need: 1) to define AS-TRM's characteristics to determine AS-TRM systems' requirements, design, and implementation; 2) to extend the TROM formalism within the TROMLAB framework to support distributed autonomic behavior; 3) to build corresponding architecture along with communication mechanism to implement distributed autonomic behavior; 4) to model a reliability assessment as one of

the non-functional properties because the reliability assessment is very important for constraining autonomic reactive systems' behavior. Our future work will include modeling and assessment of other non-functional requirements, such as performance.

1.3 Major Contributions

The main contributions of this thesis are:

- To build a five-tiers based architecture for the AS-TRM formalism's implementation in the future, which formalism is the formal foundation for integrating distributed autonomic behavior into the TROM methodology;
- To define AS-TRM's characteristics which will determine the AS-TRM's requirement specification, design, and implementation;
- To build the AS-TRM's architecture according to the five-tiers based architecture of the AS-TRM formalism and AS-TRM's characteristics. The AS-TRM's architecture is the backbone for implementing AS-TRM systems' autonomic and real-time reactive functionalities;
- To model the AS-TRM's reliability assessment which establishes the theoretical foundation for monitoring the AS-TRM's evolution;
- To collaborate with the GIPSY research group for building the AS-TRM Communication System's architecture (Based on Demand Migration Framework [VP05]). The AS-TRM Communication System is the basis for implementing the AS-TRM's communication mechanism.

1.4 The Scope of The Thesis

- Chapter 2: introduces autonomic computing technology's background and conceptual view. This chapter indicates some possible architecture perspectives and corresponding requirement specification for our AS-TRM;
- Chapter 3: introduces the TROM formalism and the TROMLAB framework's conceptual view. This chapter identifies the AS-TRM's formalism foundation;
- Chapter 4: provides a comprehensive conceptual view of the AS-TRM approach. This chapter intends to capture and convey the significant architectural decisions for further design and implementation;
- Chapter 5: provides the AS-TRM's architectural implementation. This chapter begins with a brief review of autonomic computing systems' software architecture as the AS-TRM architecture' rationale;
- Chapter 6: presents conclusions to be drawn from this thesis work and offers future work' directions on the AS-TRM project.

Chapter 2: Autonomic System

This chapter introduces the background of modeling Autonomic System (AS), which was introduced by Paul Horn at AGENDA 2001 Conference [Hor01]. The AS's conceptual view indicates several possible architecture perspectives and related requirements for extending current TROM to AS-TRM.

2.1 Characteristics of Autonomic System

Autonomic System's standardized definition and characteristics are established through the application of the Quality Metrics Methodology (QMM) [IEEE98]. While following QMM's steps, AS's characteristics are identified as quality factors illustrated in Figure 1.

Quality Factor	Definition
Anticipatory	The AS must have a projection of the user needs and actions in the future.
Context-awareness	The AS must find and generate rules for how best to interact with neighboring systems.
Openness	The AS must function in a heterogeneous world and implement open standards.
Self-awareness	The AS must be aware of its internal state.
Self-configuring	The AS must adapt automatically to the dynamically changing environments.

Self-healing	The AS must detect, diagnose, and recover from any damage that occurs.
Self-management	The AS must free system administrators from the details of system operation and maintenance.
Self-optimizing	The AS must monitor and tune resources automatically.
Self-protection	The AS must detect and guard itself against damage from accidents, equipment failure, or outside attacks by hackers and viruses.

Figure 1: AS Characteristics Identified as Quality Factors [LML05]

Then, AS's characteristics can be mapped to Quality Metrics Framework (QMF) [IEEE98] by applying the quality factors from Figure 1 to generic QMF as shown in Figure 2. In this thesis work, we will focus on self-configuration, self-healing, self-optimization, and self-protection as the core characteristics of AS-TRM.

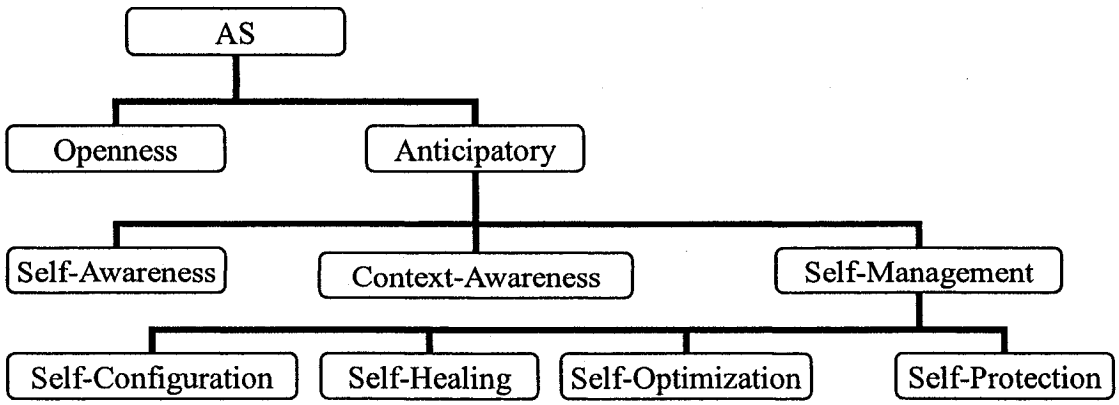

Figure 2: AS Characteristics Mapped to Quality Metrics Framework [LML05]

2.1.1 Self-Configuration

Autonomic system can automate the installation and setup of its own software in a manner responsive to the needs of platforms, users, peer groups, and enterprise [B03]. Personal computing often involves user-initiated configuration changes, and a self-configuration system understands these changes' implication and accommodates them automatically.

Self-configuration includes the capability to contact external services if needed. This capability spawns over autonomic components and autonomic system as a whole; moreover, it follows high-level information technology policies. Self-configuring components along with the system dynamically adapt to initial installation, configuration, and subsequent maintenance. The maintenance addresses new components' deployment or existing ones' removal, and increase or decrease in workload.

In an autonomic system that implements self-configuration, a common solution knowledge capability eliminates complexities by capturing installation and configuration information. Solutions are combination of platform capabilities and application elements to solve a particular customer's problem. For example, IBM Trivoli Configuration Manager [LPPC03] is an integrated inventory and software distribution solution where self-configuration is possible by using software package reference models to match desired software configuration.

2.1.2 Self-Healing

Autonomic system can monitor its own platform, detects errors or situations that may later manifest themselves as errors, and automatically initiates remediation [B03]. Self-healing includes the capability to contact external services in case of new problems and to learn those new problems as well as their resolutions. Figure 3 shows a systemic approach to self-healing.

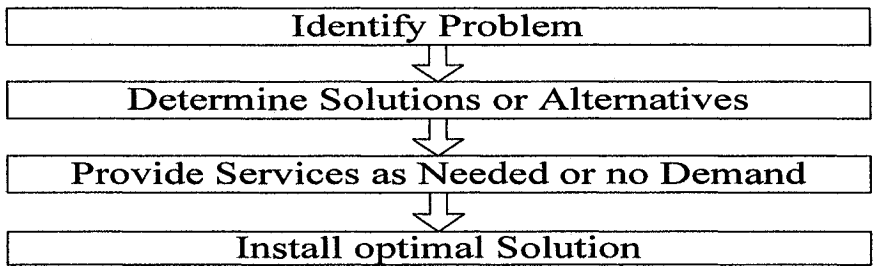

Figure 3: Self-Healing Algorithm [Cro06]

2.1.3 Self-Optimization

Autonomic system can automatically optimize its use of its own resources [B03]. This optimization must be done with respect to criteria relevant to the needs of a specific user, his or her peer group, and enterprise. Self-optimization translates itself into a high standard of service and in the end into quality of service (QoS). According to self-optimization's definition, a self optimizing system is capable to:

- Assign a solution or additional resources in order to complete a user transaction in a given time;
- Adapt to dynamically changing workload;
- Improve overall utilization of the system.

2.1.4 Self-Protection

Autonomic system can automatically configure and tune itself to achieve security, privacy, function, and data protection goals [B03]. More specifically, this characteristic protects against unauthorized access and use, worms, viruses, denial of service attacks, as well as internal threats.

2.1.5 Autonomic vs. Usability

Autonomic systems concentrate on self-management rather than the simplicity of user interface. In addition, autonomic systems have the following advantages [T03]:

- Reducing the number of low-level system administrator tasks;
- Handling the exceptions which otherwise would result in wide alerts within systems;
- Learning from the actions taken by administrators.

2.1.6 Autonomic vs. Dependability

Autonomic computing will increase dependability through the self-healing, self-configuring, self-optimizing, and self-protecting [SB03].

2.1.7 Autonomic vs. Smart Adaptive Computing

Autonomic computing system and Smart Adaptive System (SAS) share required self-adaptive behavior. The SAS has been classified into the following levels:

- Adaptation to a changing environment;
- Adaptation to a similar setting without explicitly being ported to it;
- Adaptation to a new or unknown application.

2.1.8 Autonomic vs. Proactive Computing

A true autonomic computing system may incorporate proactive computing for example through evolutionary learning. The goals of proactive computing are described as the following, and Figure 4 illustrates the relationship between proactive and autonomic computing paradigms [WPT03].

- Connecting to the physical world;
- Real-time and closed loop operation;
- Techniques that allow computers to anticipate user needs;
- Addressing security and privacy concerns;
- Dealing with uncertainty;
- Planetary scale system;
- Deep networking.

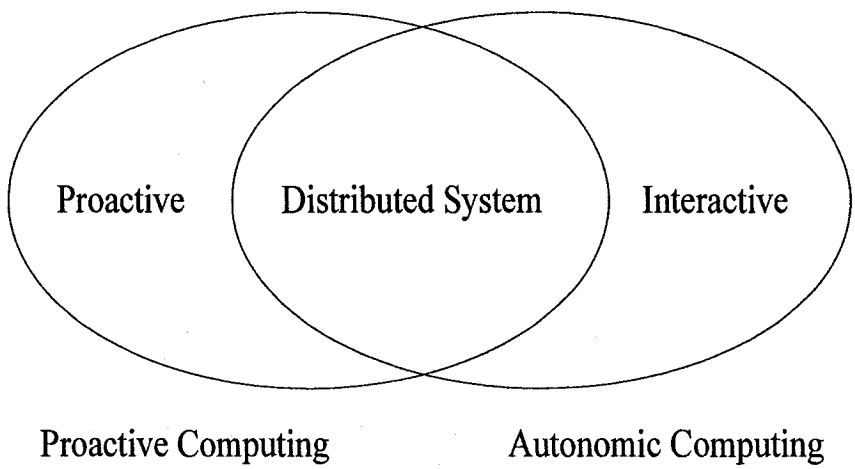

Figure 4: Relationship between Two Computing Paradigms [WPT03]

2.1.9 Autonomic vs. Introspective Computing

Autonomic computing implies a system reacting to events whereas introspective computing involves both reactive and proactive behavior [WMK03].

2.2 Autonomic System Modeling

The architectural concepts presented in this section will give a direction for achieving our research goal. These concepts are mainly based on IBM Corporation's blueprints and on-going research for autonomic computing [IBM03, IBM04, IBM05].

The basis of the autonomic computing architecture is the autonomic computing control loop as illustrated in Figure 5, and a more complex control loop is called an intelligent control loop [IBM03].

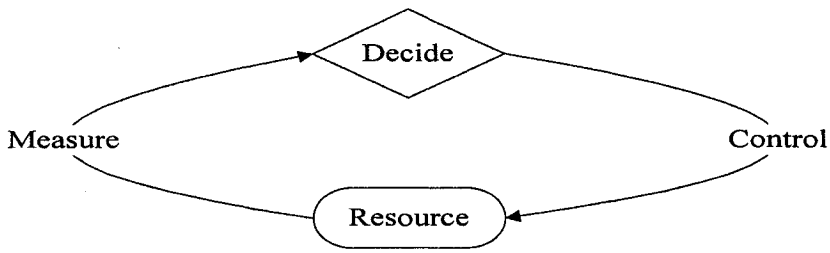

Figure 5: Autonomic Computing Control Loop [IBM03]

2.2.1 Architecture

The autonomic computing architecture is organized into two parts: layers corresponding to decision-making contexts and parts with distributed infrastructure [IBM04]. The decision-making contexts are used to clarify the purpose and role of a control loop within the autonomic computing architecture. The distributed infrastructure can be visualized as a service bus that integrates: managed resources, touchpoints, autonomic managers, and an integrated solution console as illustrated in Figure 6.

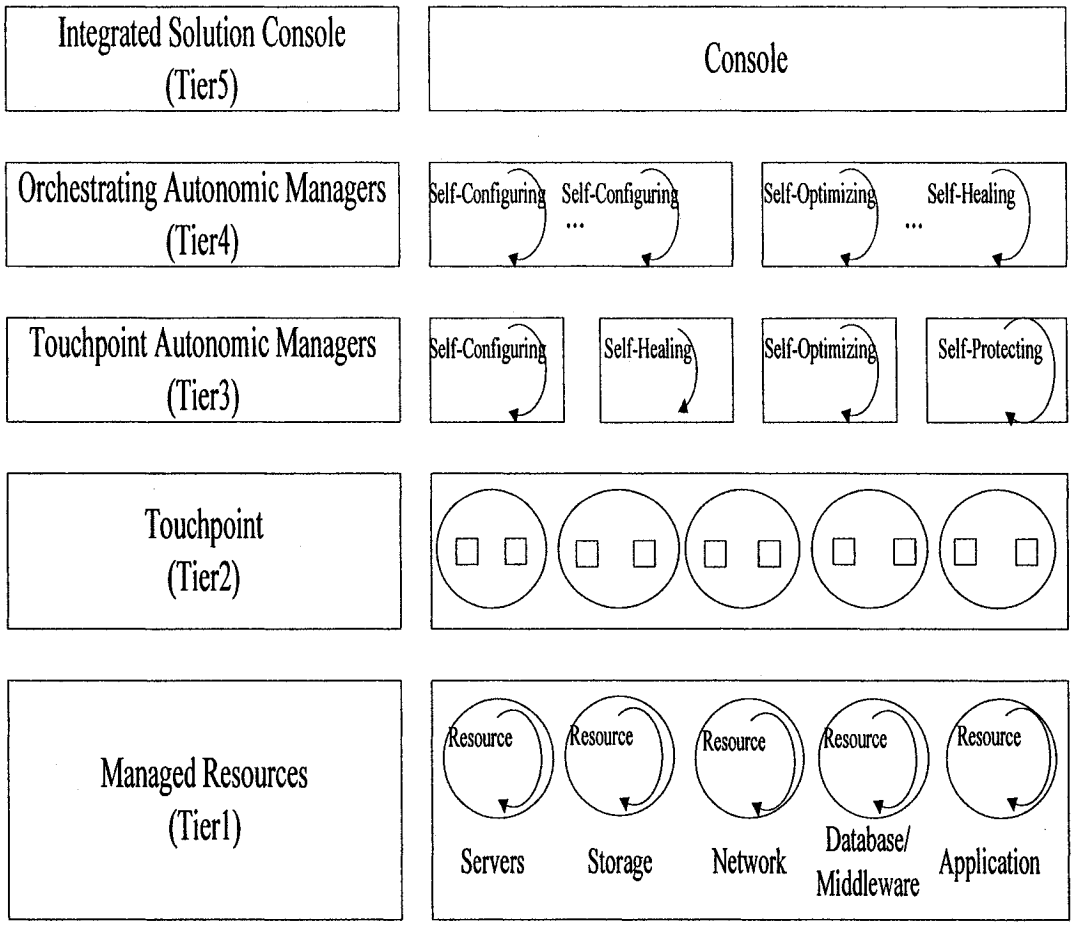

Figure 6: Autonomic Computing Reference Architecture [IBM04]

The first layer contains managed resources that exist in the real-time environment or in the Information Technology environment and that can be managed. The next layer contains standard interfaces acting as touchpoints for the management of resources from the first layer. Layer three contains autonomic managers each implementing a particular control loop. The fourth layer provides the system wide autonomic capability because the orchestrating managers have the broadest view of the overall IT infrastructure. The top layer provides an interface for common system management as an integrated solution console.

2.2.2 Intelligent Control Loop

As stated above, the basis of autonomic architecture is the control loop. Enhanced with decision-making components that monitor, analyze, plan, and execute using shared knowledge create an intelligent control loop. An intelligent control loop is also referred to autonomic element or autonomic component. In an intelligent control loop, the autonomic manager analyses, models and learns about managed elements; moreover, the manager also plans and takes actions to achieve desired goals and objectives of the managed elements [IBM03].

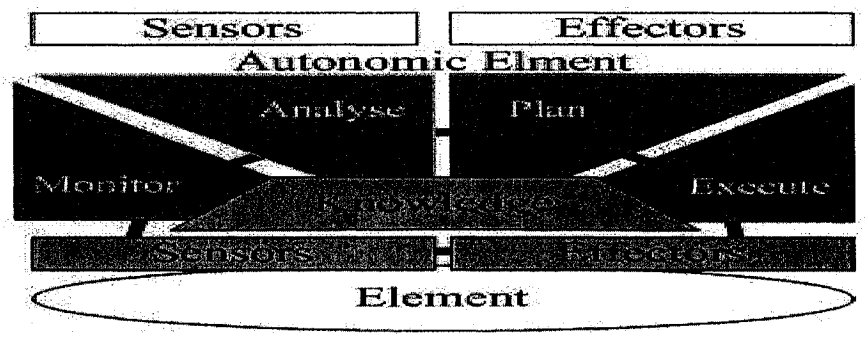

Figure 7: Intelligent Control Loop [IBM03]

The managed element is a controllable system component. It can be a single resource (a server, a database server, etc) or a collection of resources (a pool of servers, cluster or business application). It is controlled through its sensors and effectors.

The sensors provide the mechanisms to collect information about the state and state transition of an element. They retrieve the information about current state or a set of management events (unsolicited, asynchronous messages or notifications) that flow in when the state of an element changes in a significant way.

The effectors are mechanisms which changes the state (configuration) of an element. They are a collection of application programming interfaces which change the configuration of managed resource in some important way.

The autonomic manager is a component which implements the control loop. It is broken down into the following distinct parts which share the knowledge, and the four parts collaborates using asynchronous communication techniques, such as messaging bus.

- Monitor: provides the mechanisms which collect, aggregate, filter, manage and report details (for example, metrics and topologies) collected from elements;
- Plan: provides the mechanism to correlate and model complex situation, such as time-series forecasting and queuing models. Those mechanisms allow the autonomic manager to learn about the IT environment and help to predict future situation. The plan part also provides the mechanisms to structure the action needed to guide its work. For an autonomic manager, it is responsible for interpreting and translating policy details;
- Execute: provides the mechanisms which control the execution of a plan with the consideration of on-the-fly updates;
- Analysis: it is responsible for determining if the autonomic manager can abide by the policy, now and future.

Shared knowledge stores the data like metrics, commands, topology information, events, logs, performance data, and policies. The data is collected by sensors and effectors and analyzed in analysis phase.

Autonomic managers are required to work together, communicate and negotiate the self-management of elements. The sensors and effectors provided by the autonomic manager facilitate collaborative interaction with other autonomic managers. Furthermore, autonomic managers can communicate with each other in both peer-to-peer and hierarchical arrangements.

2.3 Autonomic Manager Development

There are seven core capabilities available for autonomic manager development: 1) policy determination; 2) solution knowledge; 3) common system administration; 4) problem determination; 5) autonomic monitoring; 6) complex analysis; 7) transaction measurement [IBM04].

2.3.1 Policy Determination

Policies are basically the key part of the knowledge used by autonomic managers to make decision, for they contain the criteria for achieving goals or determining the directions of actions. Policies are essentially controlling the planning components of autonomic managers. Figure 8 shows the policy management within the autonomic components.

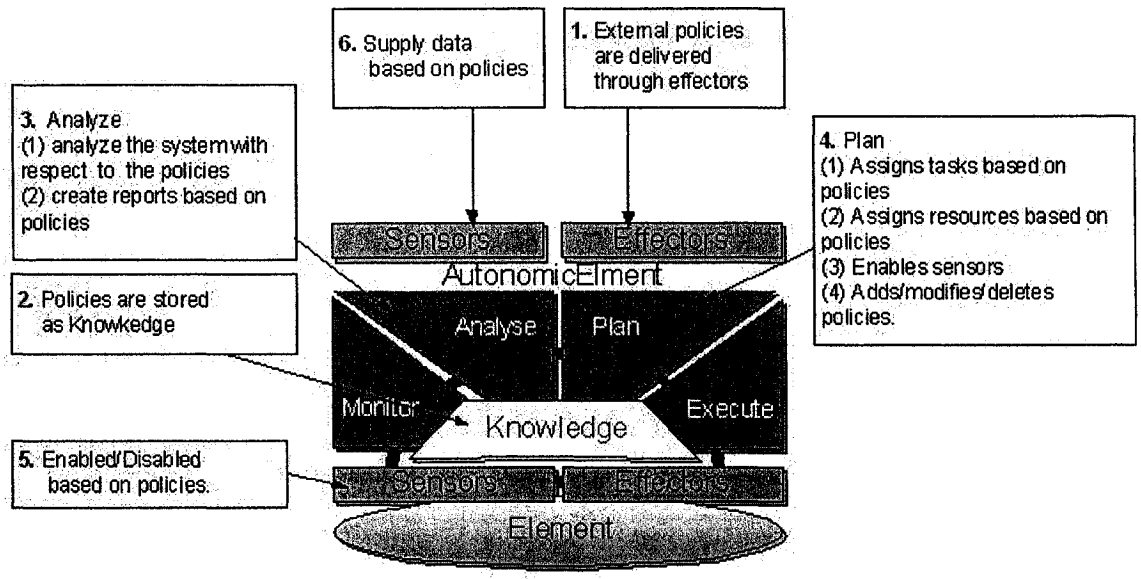

Figure 8: Policy Management [Cro06]

The IBM's autonomic computing blueprints define the specifications and capabilities for policy-based autonomic managers as the following [IBM04]:

- Specification of canonical configuration parameters for management elements;
- Format and schema used to specify user requirements or criteria;
- Mechanisms used, including wire formats, for sharing and distributing policies;
- Schema used to specify and share policy between autonomic managers.

2.3.2 Solution Knowledge

Solution knowledge contains many types of data from multiple resources, such as operating system, application languages, system utilities, and performance data. It can be used in all areas of autonomic computing like configuration, optimization, healing, and protection.

Common solution knowledge removes the complexity introduced by differences in formats and install tools; moreover, the knowledge acquired in a consistent way can be used by autonomic managers in contexts other than configuration, such as problem determination or optimization.

Solutions are the combination of platform capabilities (operating system as well as middleware) and application elements. The idea is to get the information to support install, configuration, and maintenance processes at the solution level instead of using proprietary mechanisms.

The autonomic computing blueprint defines a set of constructs for composing installable units and design patterns that make it possible to standardize solution knowledge, and the following are three categories of them [IBM03]:

- Smallest installable unit: contains one atomic artifact;
- Container installable unit: aggregates a set of artifacts for a particular container type;
- Solution module installable unit: contains multiple instances of container installable units.

Furthermore, the autonomic computing blueprint identifies a number of supporting technology components for solution knowledge as the following:

- Dependency checker: determines whether the dependency of an artifact is satisfied in the target hosting environment;

- Installer: knows how to extract the artifacts in the installable units and invoke appropriate operations on the target hosting environment;
- Installable unit database: a library for installable units;
- Deploy logic: knows how to distribute an installable unit to an installer component;
- Installed unit "instances" database: stores the configuration details about installable units and the target hosting environments.

2.3.3 Common System Administration

Common system administration can be achieved by the common console approach which consists of the framework for reuse and a set of console-specific components provided by other product development groups [IBM05]. The primary goal of a common console is to provide the single platform, which can host all the administrative console functions in a manner that allows users to manage solutions rather than managing an individual system or product.

By enabling the increased consistency of presentations and the behaviors across the administrative functions, ranging from setup and configuration to solution runtime monitoring and control, the common console creates the familiar user interface which promotes reusing learned interaction skills instead of learning new product interfaces [IBM05].

2.3.4 Problem Determination

In order to address the diversity of the data collected, the autonomic computing blueprint defines the architecture of common problem determination as the following:

- Normalizes the data collected in terms of format, content, organization, and sufficiency by defining the base set of data which must be collected or created when a problem or event occurs. This definition includes the information on both data and the format that must be used for each field;
- Categorizes the data into a set of situations, such as start or stop;
- Accommodate legacy data sources like logs and traces by defining an adapter/agent infrastructure which will provide: a) the ability to plug in adapters to transform data from a component specific format to standard format; b) sensors to control data collection, such as filtering and aggregating.

2.3.5 Autonomic Monitoring

Autonomic monitoring is a capability which provides the extensible runtime environment for an autonomic manager to gather and filter data collected by sensors, and this capability includes [IBM03]:

- A common way to capture the information from managed elements by sensors;
- Built-in sensor data filtering functions;
- A set of pre-defined resource models (such as machine memory and connectivity) as well as the mechanism for creating new models which enable the combination of different pieces of sensor data to describe the state of a logical resource;

- The ability to incorporate policy knowledge;
- The ability to plug in analysis engines which can provide basic event isolation, root cause analysis, server level correlation across multiple IT system, and the automate initiation of corrective actions.

2.3.6 Complex Analysis

The autonomic computing blueprint defines the complex analysis building blocks which autonomic managers can use to represent knowledge, perform analysis and planning; moreover, the components and tools of complex analysis technology provide the power and flexibility needed for building practical autonomic managers [IBM03]. It is important for an autonomic manager to quickly analyze the data which is dynamic and changing continuously through the time.

Common data analysis tasks include classification, the clustering of data to characterize complex states and detect similar situations, the prediction of workload as well as throughput according to empirical data, and the reasoning for causal analysis, problem determination and the optimization of resource configurations.

Complex analysis techniques use rule languages which supports reasoning by the procedural and declarative rule-based processing of managed resource data. Application classes can be directly imported into rule sets so that data can be accessed (using sensors) and control actions can be directly invoke from rules (using effectors). Rule sets can include multiple rule blocks so that the mix of procedural as well as inference methods can be used to analyze data and define the behavior of autonomic managers [IBM03].

2.3.7 Transaction Measurement

Transaction measurement represents the knowledge of transaction flow across an autonomic architecture [IBM03]. Autonomic managers need the transaction measurement capability, which spans system boundaries to understand how heterogeneous systems' resources combine to a distributed transaction execution environment. By monitoring the measurements, autonomic managers can analyze and plan to change resource allocations, for optimizing performance across the multiple systems based on policies, and determine potential bottlenecks in systems.

2.4 Evolutionary Approach

To implement autonomic computing, IBM proposes an evolutionary approach to improve current existing complex systems. This process will eventually implemented by every enterprise by adoption of technologies as well as supporting processes. Figure 9 shows the evolution toward more highly autonomic capabilities [IBM05].

- Basic level: every infrastructure element is independently managed by IT professionals who set up, monitor and replace it;
- Managed level: systems management technologies can be used to collect information from disparate systems onto fewer consoles;
- Predictive level: provide correlation among several infrastructure elements, and these elements can recognize patterns, predict optimal configuration, and provide the advice on what course of action the administrator should take;

- Adaptive level: people become more comfortable with the advice and predictive power of these elements, and IT environment can automatically take the right actions according to available information as well as the knowledge of what is happening in the environment;
- Autonomic level: IT infrastructure operation is governed by business policies as well as objectives, and users interact with autonomic technologies to monitor business process or alter objectives.

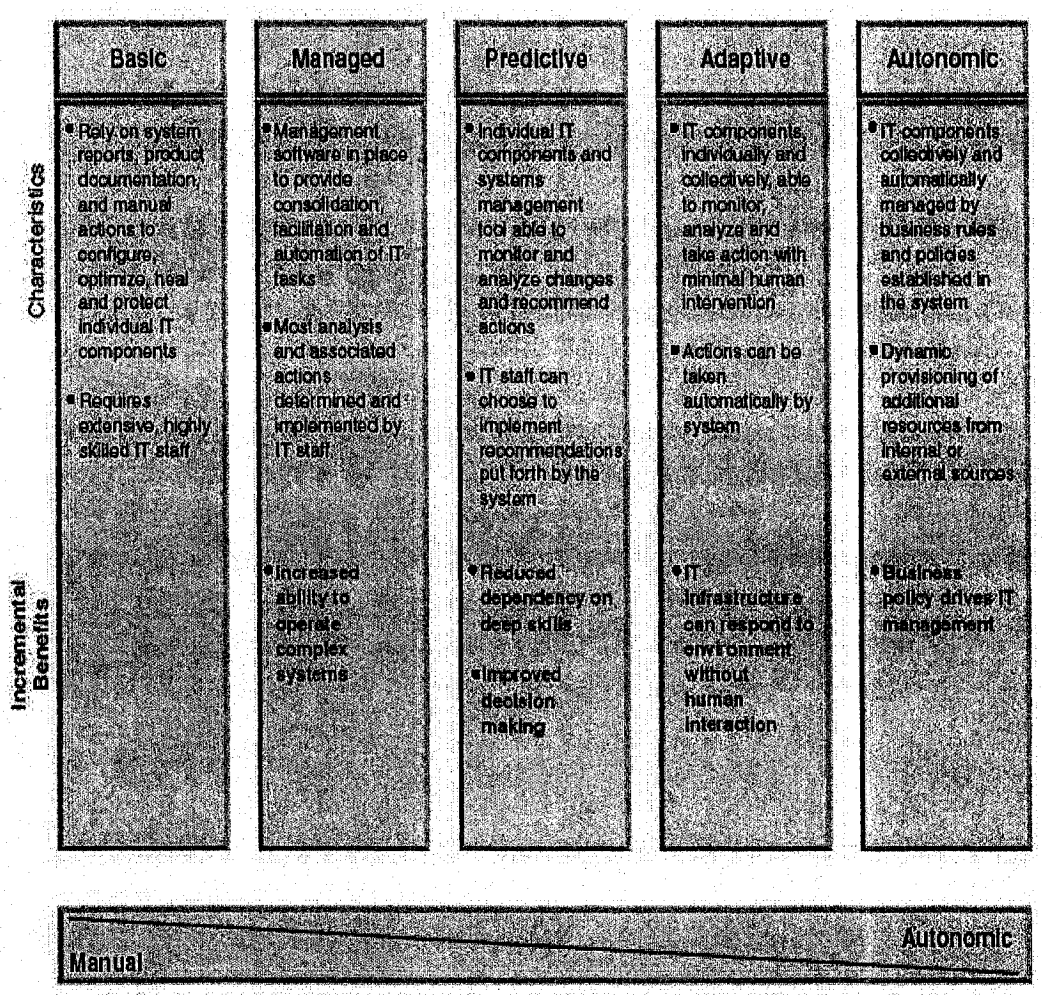

Figure 9: Autonomic Computing Maturity Index [IBM05]

2.5 Autonomic Computing Open Standards

Open standards are defined as interfaces of formats which are openly documented; they have been accepted and freely adopted by industries. Industry standards are necessary to support autonomic computing such that the following are ensured:

- Uniform approach to instrumentation and data collection: 1) enable the intersystem exchange of instrumentation and control information; 2) create the basis for collaboration and autonomic behavior between heterogeneous systems;
- Dynamic configuration;
- Operation.

The following are some proposed standards of interests to autonomic computing, and Figure 10 gives the insights on the mapping of open standards to autonomic core capabilities [Mur04].

Solution Knowledge
No standard existing; to be developed
Common System Administration
JSR168
Problem Determination
CIM, SNMP, JSR3, JMX, BlueFin, OGSA, ARM
Autonomic Monitoring
CIM, SNMP, JSR87, BlueFin, OGSA, ARM

Policy Based Management
RFC3060, JSR3, JMX
Complex Analysis
JSR87, ARM
Transaction Measurement
CIM, JSR3, JMX, JSR87, OGSA, ARM

Figure 10: Open Standards vs. Autonomic Capabilities [Mur04]

- Java Portlet Specification (JSR168): an APIs specification for the Java Enterprise Platform to enable interoperability between Java portlets and Web portals. This specification defines a set of APIs for portal computing that address the areas of aggregation, personalization, presentation, and security.
- Common Information Model (CIM): the object-oriented information model providing the conceptual view of physical and logical system components;
- Distributed Management Task Force (DMTF): the industry organization which is leading the development, adoption, and unification of management standards as well as initiatives for desktop, enterprise and environments;
- Policy Core Information Model (RFC3060): the standard which presents the object-oriented information model for representing policy information developed jointly in the IETF Policy Framework WG and as the extensions to CIM activity in DMTF;

- BlueFin: the proposed standard for data collection;
- Open Group Application Response Measurement (ARM): for application instrumentation;
- Open Grid Service Architecture (OGSA): defines the standard mechanisms for creating, naming, and discovering services as well as specifies various protocols to support accessing services. It is the framework for distributed computing according to Web protocols;
- Java Management Extensions (JSR3, JMX): provide the management architecture, the API and services for building distributed, dynamic, and modular solutions to manage Java-enabled resources;
- Java Agent Services (JSR87): a set of objects and service interfaces which support the deployment and operation of autonomous communicative agents;
- Simple Network Management Protocol (SNMP): enables network administration to manage network performance, find and solve network problems as well as plan for network growth.

2.6 Related Work on Autonomic Computing

2.6.1 Unity

Unity is a multi agent systems approach to autonomic computing [T04]. The aim of Unity is to develop an autonomic distributed computing system based on the interactions among autonomous agents which are called autonomic elements. The components of the Unity system are implemented as autonomic elements:

- Computing resource elements: databases, storage systems, servers, etc.;
- Application manager element: responsible for the internal management of environment, for obtaining the resources needed by environment, and for the communication with other elements depending on the needs of environment;
- Resource arbiter element: calculates the optimum resource based on the estimate received from each application environment;
- Server element: responsible for publishing the address of server and capabilities so that the possible users of the server can use them;
- OSContainer: receives the requests from elements to start up the services or autonomic elements;
- Registry element: enables the elements to locate other elements for their communication. It corresponds to the registries in other multi-agent systems;
- Policy Repository element: provides the human-computer interfaces allowing administrators to enter high-level policies that control the system operations;
- Sentinel element: monitors the functioning of an element on behalf of another element. If the monitored element becomes unresponsive, the sentinel sends a notification to the element which requests the monitoring about the situation.

Each autonomic element is responsible for its own internal autonomic behavior of managing the resources which it controls, and managing its own internal operations, such as self-configuration, self-optimization, self-protection, as well as self-healing. Each element is also responsible for forming and managing the relationships which it enters

into for accomplishing its goals which is the external autonomic behavior that enables the system as a whole to be self-managed.

2.6.2 OceanStore

OceanStore is a "global persistent data storage designed to scale up to billions of users" [OS02]. Researchers at Berkeley University of California are studying systems that perform continuous on-line adaptation called Introspective Computing through continuous optimization to adapt to server failures, denial of service attacks and autonomic computing.

2.6.3 Recovery-Oriented Computing

Recovery Oriented Computing (ROC) is a project within Berkeley University of California that explores autonomic computing techniques for building reliable Internet services [P02]. The researchers investigate recovery from failure techniques. ROC focuses on Mean Time to Repair (MTTR) rather than on Mean Time to Failure (MTTF) in order to provide system availability. The following techniques are mentioned in the report: 1) redundancy; 2) failure containment; 3) fault insertion testing; 4) error diagnosis; 5) non-overwriting storage systems; 6) enhanced availability.

2.6.4 Anthill

Researchers at University of Bologna, Italy are working on Anthill [Anthill01] project. Anthill is a framework that leverages the design, implementation and verification of peer-to-peer application. Those applications can be visualized as Complex Adaptive Systems with inherent emergent behavior and interesting properties as resilience, adaptation and self-organization. Architecturally an Anthill system is composed of

dynamic network of peer nodes, societies of adaptive agents (ants) that can travel over the network interacting with nodes and cooperating with other agents.

2.6.5 J2EEML

In order to make development of autonomic application easier, researchers have developed J2EEML: Applying Model Driven Development to Autonomic Enterprise Java Bean Systems or J3 Process [WSG05]. Their project has the following four components:

- A domain specific language J2EEML, for describing autonomic EJB systems, their goals and their adaptation plans;
- A framework for Java called JFense;
- J2EEML model interpreter called Jadapt to make developing of autonomic systems more feasible.

J2EEML is actually a model-driven development (MDD) tool that can formally capture the design of EJB systems (EJB Structural Model), their quality of service (QoS) requirements (Goal Model), and the autonomic properties that will be applied to the EJBs (Goal-to-EJB Mapping). It supports quick development of autonomic EJB applications via code generation, automatic checking of model correctness, visualization of complex QoS and autonomic properties [WSG05].

2.6.6 Autonomic Computing Infrastructure

Autonomic Computing Infrastructure (MAACE) is a project conducted at Institute of Artificial Intelligence, Zhejiang University, and aims to support the development and deployment of intelligent applications. To date, the team of researchers has implemented a

prototype system that enables self-configuring and self-optimizing of any networked application. The architecture of MAACE is depicted in detail in [HGC04]. The authors note that the architecture is based on previous works: An Infrastructure for Managing and Controlling Agent Cooperation and An Infrastructure for Managing and Controlling the Social Behavior of Agents.

2.7 Issues and Directions in Open Autonomic Computing

In this section, we state a few challenges which the current technologies do not address yet [B03].

- Security: There are many opportunities to apply autonomic computing technologies to security problems such as: 1) the automatic updating of security settings; 2) the secure recovery from software failures; 3) the discovery and remediation of security exposures;
- Connectivity: there is a strong interaction between the autonomic behavior of connectivity and security. Although the technology of secure sharing exists, the policies and information for controlling that sharing are still generally lacking;
- Storage: the challenge in storage is how to abstract and manage both the physical location of the data and the privacy as well as security requirements for the data;
- Peer group collaboration: 1) how to form a peer group; 2) identify the specific collaboration type for the peer group; 3) determine the degree of trust which any member puts in the information obtained from any other member within the peer group;

- Virtualization: enhances isolation, containment, and security. It reduces the domain of an autonomic manager to the contents of a virtual machine; moreover, virtualization provides an effective way for legacy software systems to coexist with current operating environments.

2.8 Metrics and Evaluation of Autonomic Computing

This section describes a set of metrics and measurements by which we can evaluate and compare autonomic systems [MH04].

- Quality of Service: it should reflect the degree to which the system is reaching its primary goal, and it typically consists of metrics. It is a highly important metric in autonomic systems because they are typically designed for improving some aspects of a service. It can be measured as a global goal metric and at the sub-service or component level;
- Granularity (Flexibility): an important issue when comparing autonomic systems and it is important for the systems in which unbinding, loading, and rebinding a component take a few seconds. These few seconds are tolerable in a thick-grained component based architecture, but not in finer-grained architectures where change is either more regular or the components smaller;
- Time to Adapt and Reaction Time: measurements concerning about the system reconfiguration and adaptation. The time to adapt is the measurement of the time that a system takes to adapt to the change in the environment. The reaction time is the time between when an environmental element has changed and the system

recognizes that change, decides on which reconfiguration is necessary to react to the change and get the system ready to adapt it;

- Sensitivity: a measurement of how well the self-adaptive systems fit with their environment;
- Stabilization: the time taken for the system to learn its environment and stabilize its operation.

2.9 Summary

In this chapter, we have briefly reviewed some concepts of Autonomic System, which concepts can be applied to TROMLAB framework.

Autonomic System has the characteristics of self-configuration, self-healing, self-optimization and self-protection. Autonomic computing control loop makes the foundation of autonomic computing architecture. In order to implement those characteristics, autonomic managers should have the capabilities of policy determination, solution knowledge, common system administration, problem determination, autonomic monitoring, complex analysis, and transaction measurement.

Finally, we have briefly described the following aspects on autonomic computing: 1) evolutionary approach; 2) open standards; 3) related work; 4) issues and directions; 5) metrics and evaluation.

Chapter 3: TROMLAB Framework

TROMLAB is a framework where in Time Reactive Object Model (TROM) formalism, language and method, can be practiced in accordance with a process model that integrates formal methods with several phases of the development life cycle. The process model incorporates iterative development, incremental design, validation, and formal verification of design models. The essential contribution of the TROMLAB framework is the integration of the formalism with a practical development methodology. The benefits include an easy-to-use interface for the modular design of the system components, and a basis for rigorous analysis [AAM96].

This chapter introduces the TROM formalism; the conceptual view of TROM identifies the architecture foundation on which we extend.

3.1 TROM Methodology

The TROM formalism is a three-tier formal model illustrated in Figure 11. As a layered model, each lower tier communicates only with its immediate upper tier. The independence between the tiers makes modularity, reuse, encapsulation, and hierarchical decomposition possible. The three-tier structure describes the system configuration, reactive classes, and relative Abstract Data Type. The upper-most tier is the subsystem configuration specification. It specifies the object definition, their collaboration, and the port links, which regulate the communication tunnels between objects. The middle tier is the TROM class, which is a Generic Reactive Class (GRC) and is included in the subsystem. The TROM class is a hierarchical finite state machine augmented with ports,

attributes, logical assertions on the attributes, and time constraints. The lowest tier is the Larch Shared Language (LSL) trait that represents Abstract Data Type used in the TROM classes [Ach95].

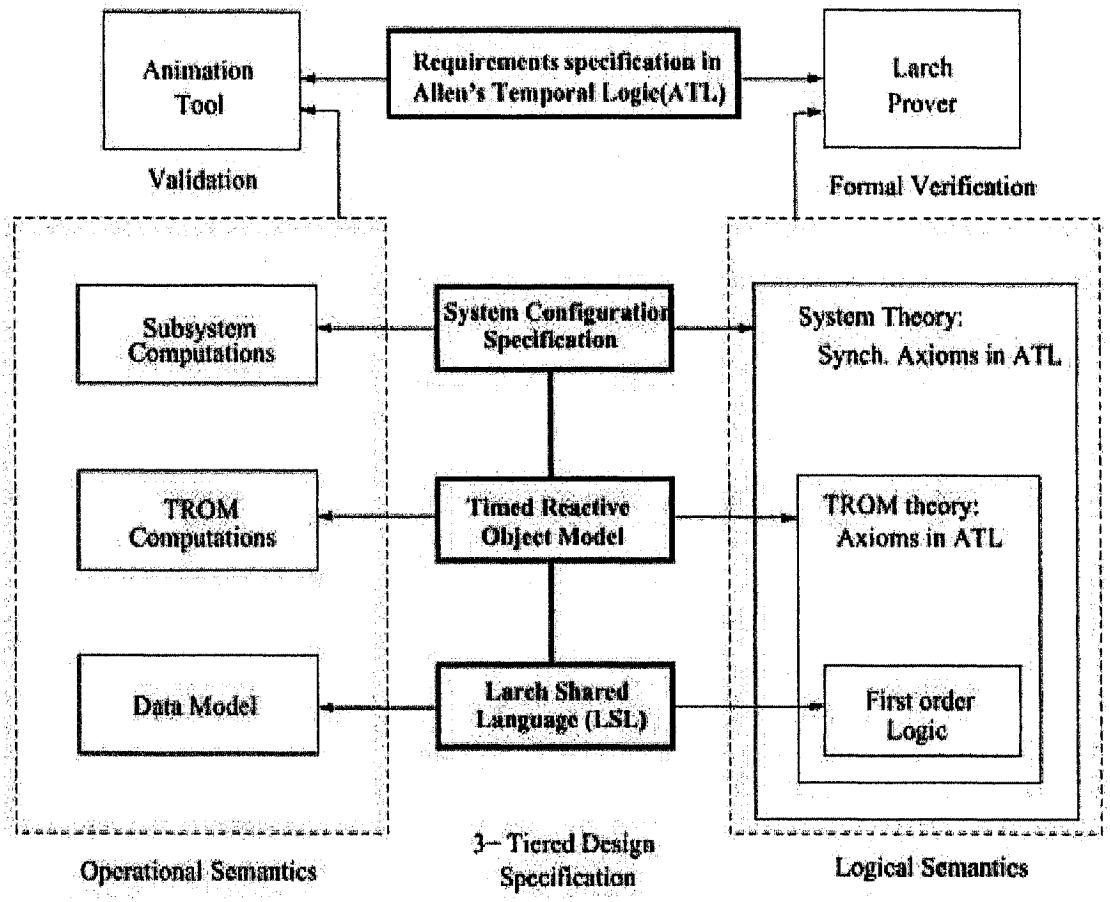

Figure 11: Overview of TROM methodology [Ach95]

3.1.1 First Tier – Larch Formalism

This tier specifies the data abstractions used in the class definitions of the second tier by means of Larch Shared Language (LSL). An abstract data type is defined as LSL trait, and Figure 12 defines a trait that specifies a Set data type.

```
Trait: Set(e, S)
    Include: Integer, Boolean
    Introduce:
        create: ->S;
        insert: e, S->S;
        delete: e, S->S;
        size: S->Int;
        member: e, S->Bool;
        isEmpty: S->Bool;
        belongto: e, S->Bool;
end
```

Figure 12: Set trait [Zha00]

3.1.2 Second Tier – TROM Formalism

A TROM object has a single thread of control. Communication mechanism among TROMs is based on synchronous message passing, also known as *rendezvous*.

- A message passing between a TROM and its environment is represented by an *interaction*;
- An interaction of a TROM with its environment occurs at *port* associated with TROM. Each port has a unique *port-type*;
- A *state* is an abstraction denoting environmental information or a system information during a certain interval of time. A state can be either *simple* or *complex*. A complex state has an initial state and a set of simple as well as complex *substates*. A TROM class has a unique initial state;
- An *event* denotes an instantaneous activity. The events are classified into three types: *Incoming*, *Outgoing*, and *Internal*;

- The *attributes* of a TROM class are of two kinds: abstract data types imported from the first tier and port types.

A formal definition of the different components of a reactive object described above is presented as an 8-tuple (Π, E, Θ, Ξ, Λ, Φ, Λ, Υ). Figure 13 shows the template for a class specification, and Figure 14 shows the example of describing a GRC.

- Π is a finite set of port-type with a finite set of ports associated with each port-type, and the null-type ρ_o whose only port is the null port o;
- E is a finite set of events and includes the silent-event tick. ε_{in} is the set of input events, ε_{out} is the set of output events, and ε_{int} is the set of internal events;
- Θ is a finite set of states. θ_0 is the *initial* state;
- Ξ is finite set of typed attributes. The attributes can be: an abstract data type specification of a data model or a port reference type;
- Λ is a finite set of LSL traits introducing the abstract data type in Ξ;
- Φ is a function-vector (Φ_s, Φ_{at}) where:
  - $\Phi_s : \Theta \rightarrow 2^{\Theta}$ associates with each state θ a set of states, possibly empty, called *substates*.
  - $\Phi_{at} : \Theta \rightarrow 2^{\chi}$ associates with each state θ a set of attributes, possibly empty, called the active attribute set.

- Λ is a finite set of transaction specification including λ_{init}. A transition specification $\lambda \in \Lambda - \{\lambda_{init}\}$, is a three-tuple: $< (\theta, \theta'); e(\varphi_{port}); \varphi_{en} -> \varphi_{post} >$;

  - $\theta, \theta' \in \Theta$ are the source and destination states of the transition.

  - event $e \in E$ labels the transition; φ_{port} is an assertion on the attributes in Ξ and a reserved variable *pid*, which signifies the identifier of the port at which an interaction associated with the transition can occur.

  - φ_{en} is an assertion on the attributes in Ξ specifying the condition under which the transition is enabled. φ_{post} is an assertion on the attributes in Ξ, primed attributes in $\Phi_{at}(\theta')$ as well as the variable *pid* and it implicitly specifies the data computation associated with the transition.

- Υ is a finite set of *time-constraints*. A timing constraint $v_i \in \Upsilon$ is a tuple $(\lambda_i, e_i', [l, u], \Theta_i)$ where:

  - $\lambda_i \neq \lambda_s$ is a transition specification.

  - $e_i' \in (\varepsilon_{out} \cup \varepsilon_{int})$ is the *constrained event.*

  - [l, u] defines the minimum and maximum response times.

  - $\Theta_i \subseteq \Theta$ is the set of states wherein the timing constraint v_i will be ignored.

```
Class < name >
        Events:
        States:
        Attributes:
        Traits:
        Attribute-Function:
        Transition-Specification:
        Time-Constraints:
end
```

Figure 13: Template for Class Specification [AAM96]

```
Class Pump [@P]
    Events: OpenPump?@P, ClosePump?@P, open
    States: *closed, toopen, opened
    Attributes:
    Traits:
    Attribute-Function:
        closed -> {};toopen -> {}; opened -> {};
    Transition-Specifications:
        R1: <closed, toopen>; OpenPump[](true); true => true;
        R2: <closed, closed>; ClosePump[](true); true => true;
        R3: <toopen, opened>; open[](true); true => true;
        R4: <opened, closed>; ClosePump[](true); true => true;
        R5: <opened, opened>; OpenPump[](true); true => true;
    Time-Constraints:
        TCvar1: R1, open, (0, 5), {};
end
```

Figure 14: Generic Reactive Class: Pump [Moh04]

The anatomy of a TROM shown in Figure 15 describes the dynamic behavior of a well-formed generic reactive object (an instance of GRC). A message from an object to another object in the system is called a *signal* and is represented by a tuple $<e, P_i, t>$, denoting that the event e occurs at time t and a port P_i. A computational step of a TROM

is an atomic step which takes TROM from one state to its succeeding state as defined by the transition specifications. Each computational step is associated with an interaction signal, internal signal, or silent signal.

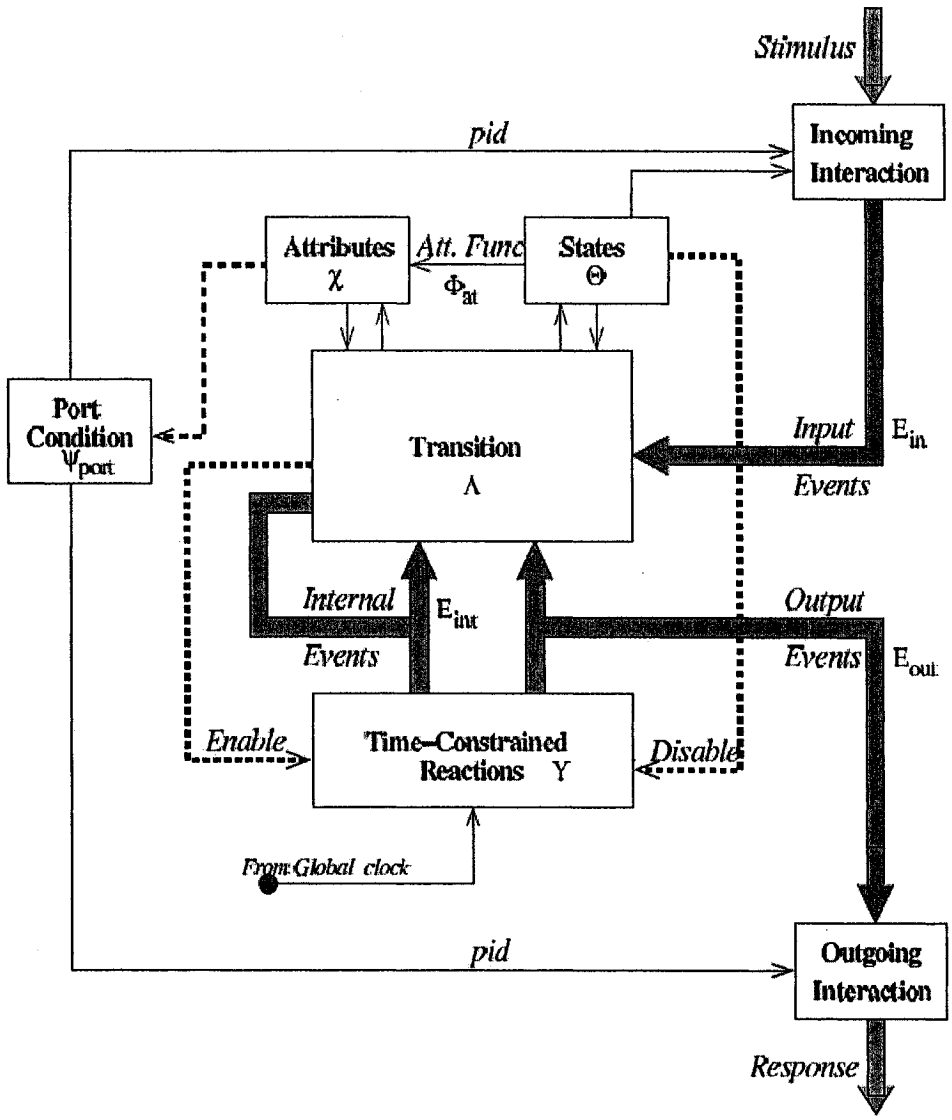

Figure 15: TROM Class Anatomy [Moh04]

3.1.3 Second Tier - Composite Classes

Composite class is introduced to minimize design complexity and to promote modularity at subsystem level. A composite class is a macro-architecture type that may include TROM classes (micro-architectures) as well as other macro-architectures. The composition rule that determines the configuration of components in a macro-architecture is based on port-type compatibility. An object instance of a composite class, called composite object, can have multiple threads of control [AAM96].

TROM classes and composite classes can be composed to obtain a new composite class by gluing the compatible port types. The port types of composite class are available for external communication; the compatible port types which are glued together by connectors become *internal* to the composite class and are not available for external communication. The events that are associated with the glued port types of classes become *shared* events for the objects, and all other external (internal) events become external (internal) events of the composite class. The states, attributes, traits, attribute functions, transition specifications, and timing constraints of the state machines associated with classes are absorbed in the description of the composite class. Figure 16 shows the syntax for describing a composite class.

CompositeClass<*identifier*>[<*listofprot - types*>]
    Incarnations:
    Connectors:
end

Figure 16: Template for Composite Class Construction [AAM96]

The template includes the keyword *CompositeClass* introducing the name of the composite class, and sections labeled with the keyword *Incarnations*, and *Connectors*. An incarnation of a class is the class specification in which the port-type parameters may be renamed. The incarnations' section lists the incarnations of classes which participate as independent components in a composition. The Connectors section lists the connectors that glue compatible ports for internal communication between the components. The protocol for communication along a connector is implicitly available in the signatures of the ports glued by the connector. Figure 18 gives the specification of the composite class shown in Figure 17.

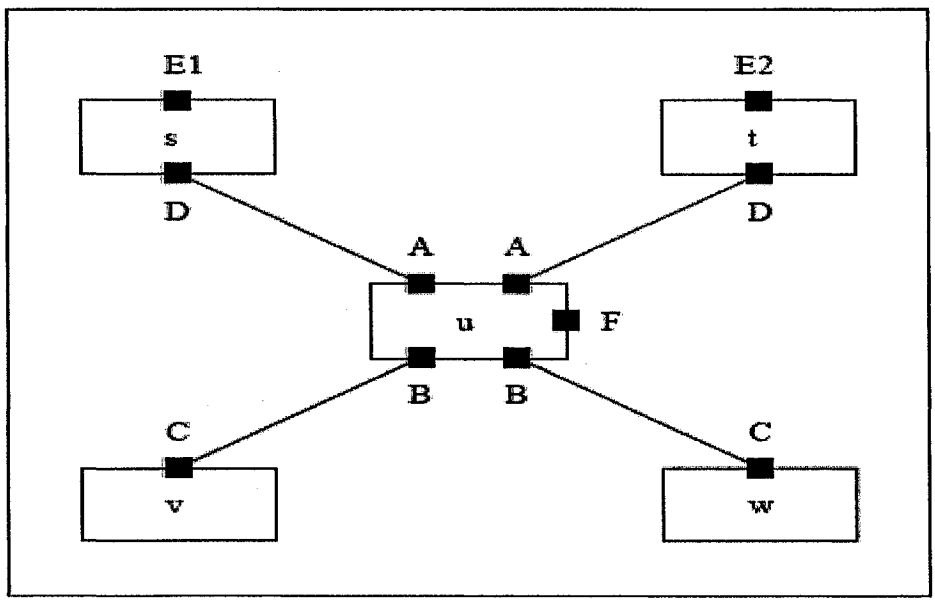

Figure 17: A Composite Class with Five Classes [AAM96]

CompositeClass Class5[@E_1,@E_2,@F]

Incarnations:

s: Class1[@D, @E_1 for @E]

t: Class1[@D, @E_2 for @E]

u : Class2[@A, @B, @F]

v, w : Class3[@C]

Connectors :

s.@ D⊳⊲u.@A

t.@ D⊳⊲u.@ A

v.@ C⊳⊲u.@B

w.@ C⊳⊲u.@B

end

Figure 18: A Composite Class Specification [AAM96]

A composite class is a macro-architecture type that may include TROM classes (micro-architectures) as well as other macro-architectures. The composition rule that determines the configuration of components in a macro-architecture is based on port-type compatibility. An object instance of a composite class, called composite object, can have multiple threads of control, and a composite class type is not a TROM class type.

3.1.4 Third Tier – System Configuration Specification

A *System Configuration Specification* (SCS) describes the system architecture by succinctly specifying the interaction relationship that can exist between the objects in a system. The template in Figure 19 shows the syntax for SCS.

```
Subsystem < name >
    Include:
    Instantiate:
    Configure:
end
```

Figure 19: Template for System Configuration Specification [AAM96]

The syntax includes the keyword *Subsystem* to introduce the identifiers for the system, and sections labeled with the keyword *Include*, *Instantiate*, and *Configure*. The Include clause is for importing other subsystems. The Instantiate clause defines reactive objects by parametric substitutions to cardinality of ports for each port type, and initializing the values of attributes in the initial state of the object. The Configure clause defines a configuration obtained by composing objects specified in the Initiate clause and the subsystem specifications imported through the Include clause.

Figure 20 shows a subsystem configuration for an arbiter system involving two users and one arbiter. The *Arbiter* objects have two ports of type @U, and each *User* object has one port of type @A.

Subsystem *ArbiterSystem*

Include:

Instantiate:

ar_1 :: *Arbiter[@U : 2].Create()*;

us_1, us_2 :: *User[@A : 1].Create()*;

Configure:

$ar_1.@u_1$ <-> $us_1.@a_1$;

$ar_1.@u_2$ <-> $us_2.@a_2$;

end

Figure 20: Arbiter System [AAM96]

Communication links in a subsystem are *external* to objects in the system, whereas connectors in a composite object are *internal* to the objects that are glued. Each composite object behaves as a black-box, and their message exchanges are transparent, but a subsystem behavior is illustrated through a message sequence chart.

3.2 Design Refinement in TROM

A design can be refined by adding more details, which may require adding more states, transitions, and strengthening time constraints. However, the design obtained from an unconstrained inheritance principle does not guarantee the preservation of properties in the derived TROM. Towards remedying this, three forms of constrained inheritances based on subtype are introduced in TROM [AAM96]. Figure 21 shows the syntax for extended class specification:

Class: < *identifier* > [< *porttypes* >]
Inherits:
Events:
States:
Attributes:
Traits:
Attribute-Function:
Transition-Specifications:
Time-Constraints:
end

Figure 21: Template Extension for Inheritance Specification [AAM96]

In the section of *inherits*, the name and port-type parameters of the inherited class is introduced. Port type parameters may be omitted for behavior and extensional inheritance specifications. The other sections (*Events, States, Transition-Specifications,* and *Time-Constraints*) may list details of the inherited class according to the kind of inheritance, and a class may inherit at most one class. Figure 22 shows the TROM class specification of basic telephone, and Figure 23 shows the definition of the class *Answer-Phone* derived from the class *basic-phone*.

Class *Basic-Phone [@P, @Q]*
Events: *OnHook?P, OffHook?P, Ring!Q, Stop!Q*
States: **idle, ringing, revCall, initCall*
Transition-Specifications:

R_1: *<idle, initCall>, <ringing, revCall>; OffHook?(true);*

true -> true;

R_2 : *<initCall, idle>, <revCall, idle>; OnHook?(true);*

true -> true;

R_3 : *<idle, ringing>; Ring!(true); true -> true;*

R_4 : *<ringing, idle>; Stop!(true); true -> true;*

end

Figure 22: A Basic Telephone Device [AAM96]

Class *Answer-Phone [@P, @Q]*
Inherits: *Basic-Phone*
Events: *Start*
States: *ringing(*wait, answer)*
Transition-Specifications:

R_5: *<wait, answer>; Start; true -> true;*

Time-Constraints:

(R_3, *Start, [2,4], {revCall, idle}*)

end

Figure 23: Enhancement with Message Announcing Feature [AAM96]

3.2.1 Behavioral Inheritance

A TROM class T obtained by refining a TROM class T′ according to the refinement mapping stated below is a behavioral subtype of T′. An object A of class T inherits the behavior of an object A′ of class T′.

1. *Attribute redefinition*: the data model of an attribute may be redefined, provided exist coercion functions for each attribute from the refined trait to the original trait.
2. *Transition redefinition*: redefinition of an inherited transition specification may be done such that:
   - The post-condition may be strengthened;
   - The port-condition of a transition involving any event e $\in \varepsilon_{out}$ may be strengthened;
   - The enabling-condition of a transition involving any event e $\in \varepsilon_{out}$ may be strengthened;
   - The initial attribute-constraint φ_{init} may be strengthened.
3. *Time-constraint redefinition:* the minimal time increased or the maximal time delay may be decreased.

3.2.2 Extensional Inheritance

The goal of extensional inheritance is to provide design refinements that preserve behavior. The TROM object A obtained by refining a TROM object A' satisfying the following constraints is an extensional inheritance of A':

1. Possible Redefinitions:
   - Any redefinition as permitted in Behavioral inheritance;
   - The source state θ of a transition may be redefined to any substate of θ;
   - The enabling-condition of any inherited transition may be strengthened;

- The port-condition of any inherited transition may be strengthened.

2. Possible Additions:

   - *Event addition*: new events may be added, enriching inherited port-types;
   - *State addition*: a new state may be added only to make an existing simple state as a complex state;
   - *Attribute addition*: new attributes and traits may be added;
   - *Transition addition*: an added transition can have only new events and new states. The superstate of the source and destination state should be the same;
   - *Time*-constraint addition: an added time-constraint can only constrain a new event.

3.2.3 Polymorphic Inheritance

The TROM object A is a polymorphic inheritance of the TROM object A' if the following conditions are satisfied:

1. Possible Redefinitions: any redefinition as permitted in Behavioral inheritance.
2. Possible Additions:

   - *Port addition:* new port-type may be added. This necessarily introduces new events;
   - *State addition:* new states may be added; refinement of an exiting state is not permitted;
   - *Attribute addition:* new attributes and traits may be added:

     - An added transition can only involve new events.

- An added transition involving an event $e \in \varepsilon_{in}$ can only have newly added states for both source and destination states.

- *Time-constraint addition*: an added time-constraint can only constrain a new event and can only be triggered by a new transition.

3.3 TROMLAB – The Development Framework

TROMLAB is a framework built around the process model in which the rigorous approach based on TROM methodology can be practiced to develop real-time reactive systems [AAM96]. Figure 24 defines the series of software engineering stages which are followed in TROMLAB towards developing real-time reactive systems. The primary distinction between traditional and TORMLAB process models is in the integration of formalism through the different stages of development. Tools have been developed to support formalism at different levels of abstractions.

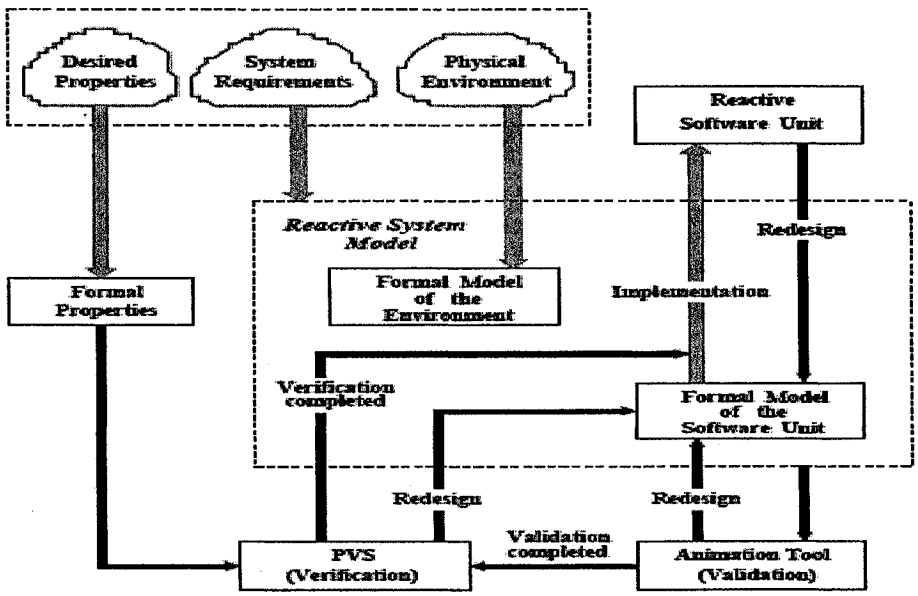

Figure 24: Process Model for Developing Complex Reactive System [AAM96]

3.3.1 Process Model

The process model requires the formal method of the environment to be produced and integrated with system elements [AAM96]. Environmental objects are abstracted and their interfaces to system elements are formally defined. System requirements that include functional and timing requirements are identified and their formal descriptions are produced. A reactive system model is composed of the software unit and the model of the environment. Integration of Rational Rose with the initial states of model building activities enables the construction of a visual model of a reactive system.

The formal model of the reactive unit is implemented by several design iterations. Validation is done by animating the reactive system design. Simulation and reasoning are the two techniques used to debug the design and predict its behavior. The validation and simulation tools use the formal model, so they are independent from any implementation constraints such as resource and process speed. If flaws due to incorrect functionalities or inconsistent timing behavior are noticed during system simulation and reasoning, the process model allows an iterative inner cycle for redefining and validating the formal model of reactive unit.

System verification is the next step of the process model. Time critical properties are formally verified at this step, such as safety properties. The system design is mechanically translated to a set of PVS theories consisting of axioms describing the timed behavior of the system [Pom99]. The desired properties are formalized and are included as lemmas in PVS theories.

3.3.2 TROMLAB Components

The current TROMLAB environment includes the following components:

- Rose-GRC Translator [Pop99]: allows reactive classes to be visually composed, edited, refined, and automatically mapped to the TROM notation;
- Interpreter [Tao96]: parses, syntactically checks a specification and constructs an internal representation;
- Simulator [Mut96, Liu03]: simulates a subsystem behavior at the design phase before the implementation, and enables a systematic validation of the specified system; the results are animated by the visualization tool [Moh04];
- Browser for Reuse [Nag99]: an interface to a library, to help users navigate, query and access various system components for reuse during the system development;
- Graphical User Interface [Sri99]: a visual modeling and interaction facility for a developer using the TROMLAB environment;
- Reasoning System [Hai99]: provides a means for debugging the system during the animation by facilitating interactive queries of hypothetical nature on system behaviors;
- Verification Assistant [Pom99]: an automated tool which enables mechanized axiom extraction from real-time reactive systems;
- Test Case Generator [Zhe02, Che02]: an automated tool for generating test cases from specifications and for optimizing the test suite based on the test adequacy measurement;

- Verification Tool [Mut00]: an automated tool that enables mechanized validation for the safety and liveness properties, and it is based on PVS;
- TROM-SRMS (Software Reliability Measurement System) [Lee03]: a reliability measurement module that predicts the level or reliability from the TROM specifications of the system;
- TROM-SCMS (Software Complexity Measurement System) [Zhu03]: calculates the architectural complexity from TROM specifications and displays the maintenance profile of the system.

3.4 Summary

In this chapter, we have briefly described all the concepts of the TROMLAB environment in which we make the extension. TROMLAB framework is the integration of the TROM formalism with a practical development methodology for developing real-time reactive systems. Our research achievements are introduced in the following chapter.

Chapter 4: Reactive, Distributed and Autonomic Aspects of AS-TRM

This chapter provides the comprehensive conceptual view of the AS-TRM approach; it is intended to capture and convey the significant architectural decisions which serve as a foundation for the further design and implementation. We focus on the structural view, the dynamic view, as well as the specific characteristics of the AS-TRM to discuss the reactive, distributed and autonomic aspects of AS-TRM approach.

4.1 Purpose and Context

Real-time reactive systems are the some of the most complex systems, so the modeling and development of real-time reactive systems becomes a very challenged and difficult work; moreover, the current TROM formalism does not have the appropriate mechanism for specifying autonomic distributed systems. On the other hand, autonomic computing is the new research area which focuses on developing complex computing system smarter and easier to manage. Thus, the goal of our work is to extend the current TROM formalism to AS-TRM to include the specification of distributed reactive autonomic components along with their relationships, and the non-functional properties constraining the behavior of the system.

4.2 Concept of AS-TRM

AS-TRM is the formal framework for autonomic distributed real-time reactive systems which leverages their modeling, development, integration, maintenance, and continuous monitoring of their reliability [VKOP06], and Figure 25 shows the concept of AS-TRM.

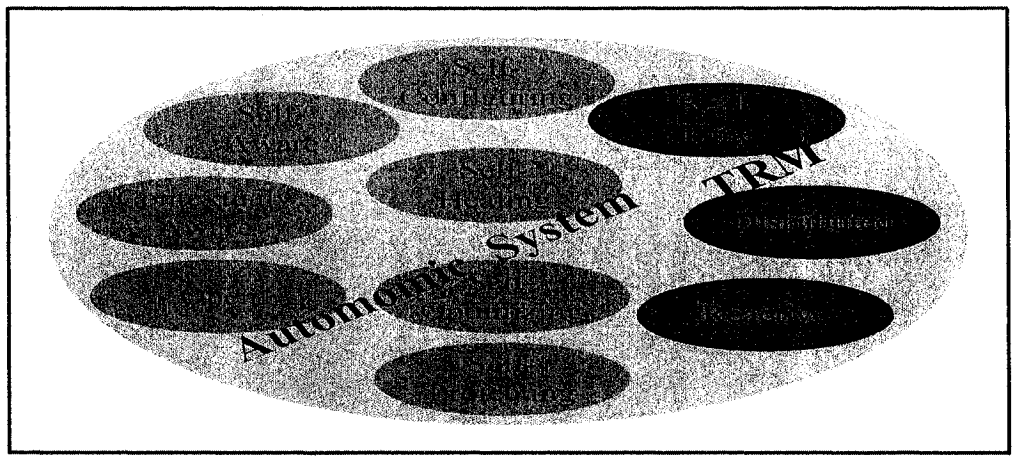

Figure 25: Concept of AS-TRM [Cro06]

4.3 Rationale of AS-TRM

AS-TRM can be considered as TROM with extended autonomic behavior, and the autonomic functionalities are those which create the autonomic behavior. Autonomic functionalities can be implemented locally, using locally maintained measurements and knowledge. The autonomic behavior can be implemented among the members within a peer group through sharing measurements and knowledge of the group. It also can be implemented by using globally available resources in which the measurements and knowledge are maintained for all clients. Generally, autonomic functionalities are implemented as the following categories [B03]:

- Local autonomic functionality: locally, autonomic decisions can be made by using the knowledge which the personal computer stores or can get by itself. Local functionalities include the automatic auditing of software configuration, local backup, the survey of connectivity environment, as well as power management, etc.

- Peer group autonomic functionality: requires the cooperation of a local community, and it includes spontaneous computing service as well as knowledge sharing, etc.
- Global autonomic functionality: enhances and extend the core autonomy of PCs; it includes software updating, backup along with restore, virus update, and mobility support service, etc.

4.4 AS-TRM Formalism

AS-TRM formalism is extending the TROM formalism [AAM96] by adding the following tiers (see also Figure 26):

- A timed reactive autonomic component: Autonomic Component (AC);
- A group of synchronously interacting ACs: AS-TRM Component Group (ACG);
- A collection of asynchronously interacting ACGs: AS-TRM System (AS).

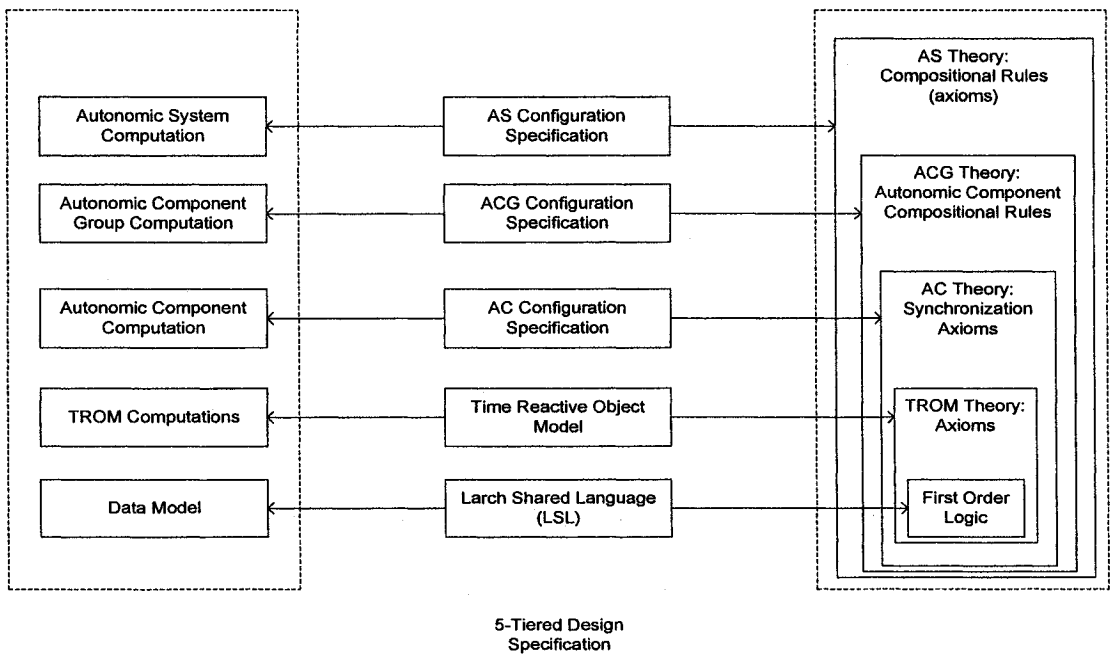

Figure 26: AS-TRM Formalism [VKOP06]

4.4.1 AC Tier

This newly added tier encapsulates the TROM objects (the second tier of the TROM formalism) into the AS-TRM autonomic components. The synchronous interactions among the ACs allow them to perform reactive tasks, and the communication between the AC and its upper tier ACG is asynchronous. An AC is responsible for undertaking a complete or partial real-time reactive task as a worker within the system.

4.4.2 ACG Tier

ACG is a set of synchronously communicating ACs which cooperates in fulfillment of a group task; each ACG can accomplish a complete real-time reactive task independently. The self-monitoring behaviour at the ACG tier as well as the asynchronous interaction between ACG and its ACs is implemented by an ACG Manager (AGM), and the responsibilities of an AGM include the following:

- Monitors the ACG reliability level required by the evolving nature of the peer group;
- Monitors the behavior of the synchronously communicating ACs within the group, and analyzes the correctness of their functionalities according to the policies;
- Receives the diagnostic messages from the ACs within the group, and then sends back corresponding treatment messages to them.
- Removes the broken ACs from ACG and inserts them back when they are ready;

- Automates the initialization and maintenance according to evolving group configuration and changes in the run time;
- Encapsulates any data under the control of the group and manages all the data shared between ACs;

The reactive behavior within AS-TRM is modeled at the AC and ACG tiers; moreover, we model the environmental objects communicating with the system as ACs, and incorporate them into the ACG to fulfill corresponding reactive tasks. In the meantime, the autonomic behavior of self-healing, self-optimizing, self-protecting, and self-configuring can be implemented at the ACG tier which is the peer group level as the following aspects:

- Automating the policy backup for the group;
- Knowledge, resource sharing within the group, and execution time optimizing based on empirical data;
- . Automating event access support from the environment;
- Automating the configuration from the AS Tier and the unplanned changes from the environment.

4.4.3 AS Tier

AS is a set of asynchronously communicating ACGs, and the self-managing behaviour as well as the asynchronous interaction between the AS and the ACGs is implemented by the Global Manager (GM). The responsibilities of the GM include the following:

- Monitors the AS reliability level required to endure the safety of the AS;

- Verifies the user access according to the security policies and different level privileges defined among AS, ACGs, ACs, and environment;
- Monitors the behavior of the ACGs and analyzes whether they work correctly according to the policies;
- Receives the diagnostic messages from the ACGs, and then sends back corresponding treatment messages to them;
- Receives the requests for updating the compositional rules of ACGs as well as the synchronization axioms among ACs from the user, and forwards these requests to the AGMs.
- Encapsulates any data under the control of the AS and manages all the data shared between ACGs;

As a result, the autonomic behaviour of self-protecting, self-configuring, self-optimizing, and self-healing can be implemented as the following aspects at the AS tier, which is the system level: 1) automating the user access support; 2) automating the configuration for users; 3) knowledge and resource sharing within the system; 4) Automating the policy backup for the whole system.

4.5 Characteristics of AS-TRM

From the formalism of AS-TRM, we can find that it has the following characteristics [VKOP06] in addition to real-time and reactive which is inherited from current TROM formalism [AAM96]:

- AS-TRM is self-managed: it can monitor its components (internal knowledge) and its environment (external knowledge) by checking the status from them, so that it can adapt to changes that may occur, which may be known changes or unexpected changes;
- AS-TRM is distributed: the components within AS-TRM can collaborate to complete a common real-time reactive task distributedly;
- AS-TRM is proactive: it can initiate changes to the system;
- AS-TRM is evolving: a) the policies of each AC can be changed in the run time according to the changes of requirements; b) the composition rules of the ACs within corresponding peer group can be changed in the run time; c) the synchronization axioms among the ACs within corresponding peer group can be changed in the run time.

4.6 Summary

In this chapter, we have specified the concept of AS-TRM approach in which we combine the advantages of both the formal representation of reactive components within the TROM formalism, and the autonomy of components under the concept of autonomic computing. The extension from current TROM formalism to AS-TRM formalism is implemented by adding the new tiers of AC, ACG, and AS, so that the AS-TRM is self-managed, distributed, proactive, and evolving in addition to the characteristics of real-time along with reactive which are inherited from TROM.

Chapter 5: Architecture of AS-TRM

The scope of this chapter is to provide the architectural implementation of AS-TRM, and we begin with a brief review of the software architectures for autonomic computing as the rationale of AS-TRM architecture. Our architectural goal is to catch the functional requirements as we stated in the chapter 4, and the most important AS system's non-functional property, namely reliability.

5.1 Related Work on Architectures for Autonomic Computing

At the software architectural level, a system is typically described as a collection of interacting components. Components perform the primary computations within the system, and the interactions between components include high level communication abstractions, such as message passing, event broadcast, pipes, and procedural calls [AAG93].

5.1.1 Multi-Agent Systems

The autonomic systems which are not composed of a single self-managing component can be developed by using intelligent agents, and each agent has its own goals to make its decisions. The agents within an autonomic system are proactive and possess social abilities. Usually, there is a clean separation between the conventional components which perform a task and the autonomic managers which implement self-management. On the other hand, in some systems, the autonomic components are inseparable from the main application logic within corresponding agents [MH04].

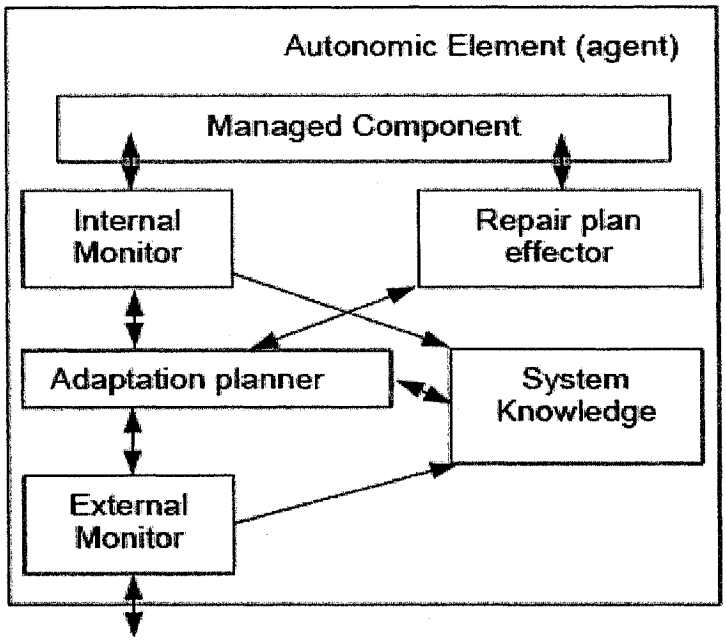

Figure 27: An Autonomic Agent [MH04]

With no centralized monitoring mechanism, agents must monitor themselves (internal monitor) and other agents (external monitor). External monitoring can be proactively achieved by having every agent send its heartbeat or pulse regularly on the autonomic signal channel which other agents send and listen on. The heartbeat provides a summary of the state of an agent to other agents which are responsible for monitoring that state [SB03].

5.1.2 Architecture Design-Based Autonomic Systems

In the architecture-based approach, individual component is not autonomic. Instead, the infrastructure which handles the autonomic behavior within the system uses the architectural description model to monitor, reason about the running system, and determine appropriate adaptive actions [MH04].

An architecture model can be considered as a graph of interacting components, which components are nodes within the graph. The arcs in the graph are called connectors; they represent interaction paths between components. Many systems allow components and connectors to be annotated with a property list as well as constraints; these properties are updated while monitoring the running systems, and the constraints on them are used to decide when an adaptation is necessary [GS02].

Figure 28 shows the diagram of the architecture model-based autonomic system, which describes the monitoring infrastructure:

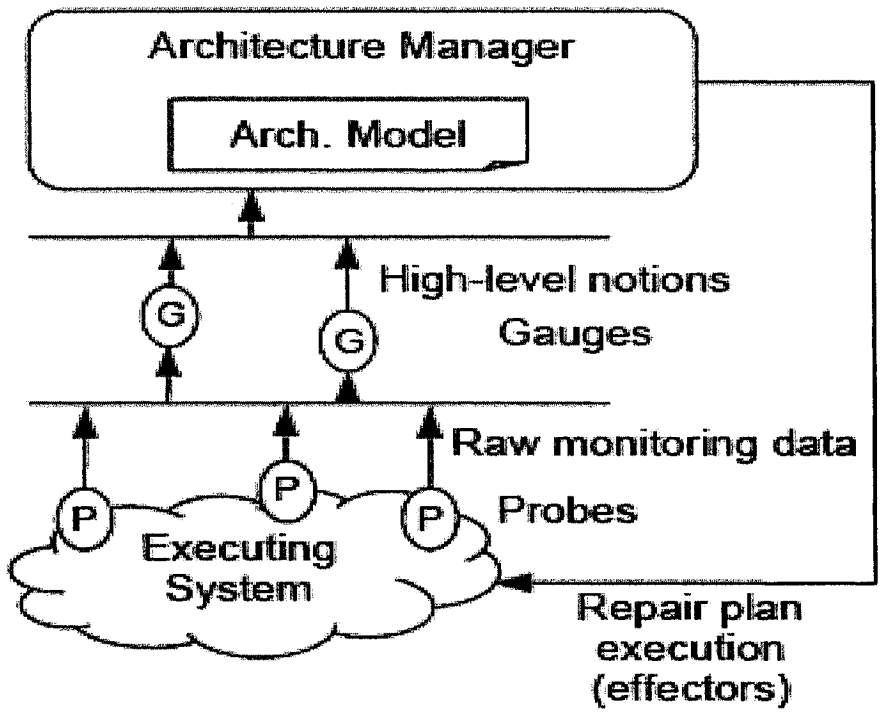

Figure 28: Architecture model-based system [MH04]

- Probes: can be inserted into the running system to monitor it; they are usually localized and deliver system-specific observation;

- Gauges: the intermediary components between the probes in the running system and the architecture manager that controls the adaptation of the system at the architecture model level;
- Repair plan: describes which components or connectors will be removed, adjusted, and inserted; it is created based on predefined repair strategies.

5.2 Rationale for AS-TRM Architecture

Compared to the architecture design-based approach, adaptive multi-agent systems have the inherent distributed architecture, which is one of the most important aspects we need to implement within the AS-TRM architecture. As a result, we choose the Multi-Agent architecture as the architecture of AS-TRM.

As we stated in the chapter 4, AS-TRM is not only a real-time reactive framework, but also an autonomic framework. The key aspects of an autonomic framework are autonomic manager and the element to be managed; moreover, the goal of an autonomic framework is to specify the interfaces as well as protocols for those elements to exchange information and data to implement autonomic behavior [B03].

- Every element needs an element-specific autonomic manager to monitor and control it; the coupling between the element and specific manager represents the lowest level of autonomic behavior;
- Element-specific autonomic managers report to the global autonomic manager which is responsible for achieving end-user goals according to some established policies.

5.3 AS-TRM Architecture

Based on the tiers of the AS-TRM formalism stated in the chapter 4, Figure 29 shows the architecture of AS-TRM which consists of Autonomic Components (ACs), AS-TRM Component Group Manager (AGM), and Global Manager (GM) which are connected to each other at the local, peer group, and system levels.

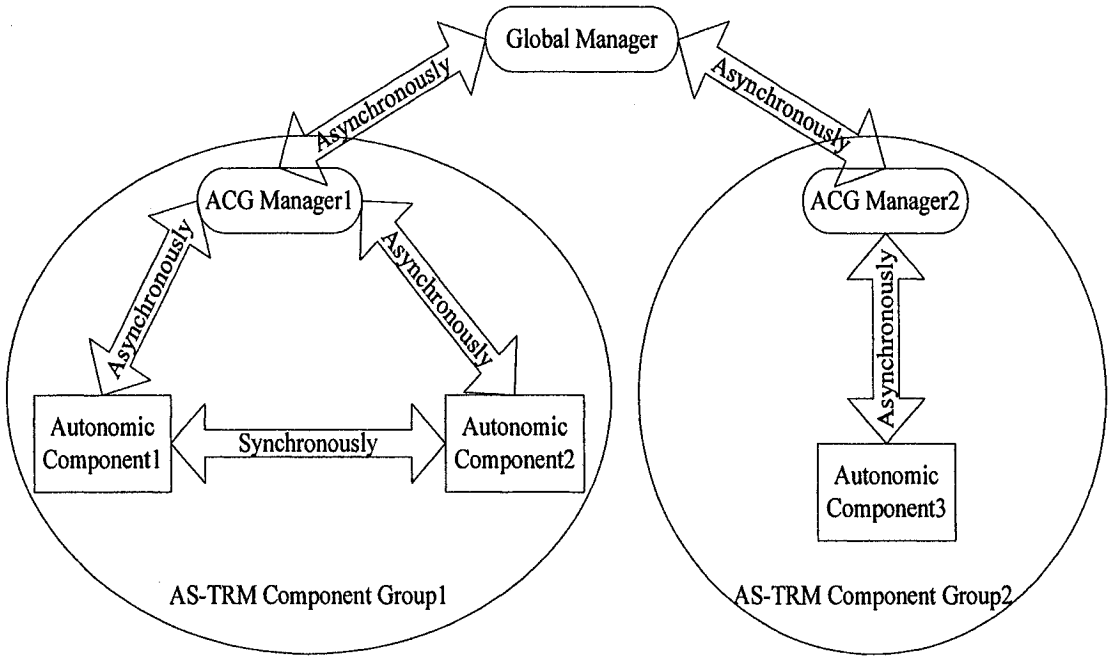

Figure 29: AS-TRM Architecture

At the peer group level, which is also the AS-TRM Component Group (ACG) level, every AGM interacts and shares knowledge as well as information with its ACs; it receives information (policies) from its superior (Global Manager) and implements them with its own resources. The autonomic behavior at this level is a result of peer knowledge-sharing, getting local agreement, and acting locally on that knowledge.

Figure 30 depicts the architecture of an ACG, and Figure 31 illustrates the anatomy of GM and AC.

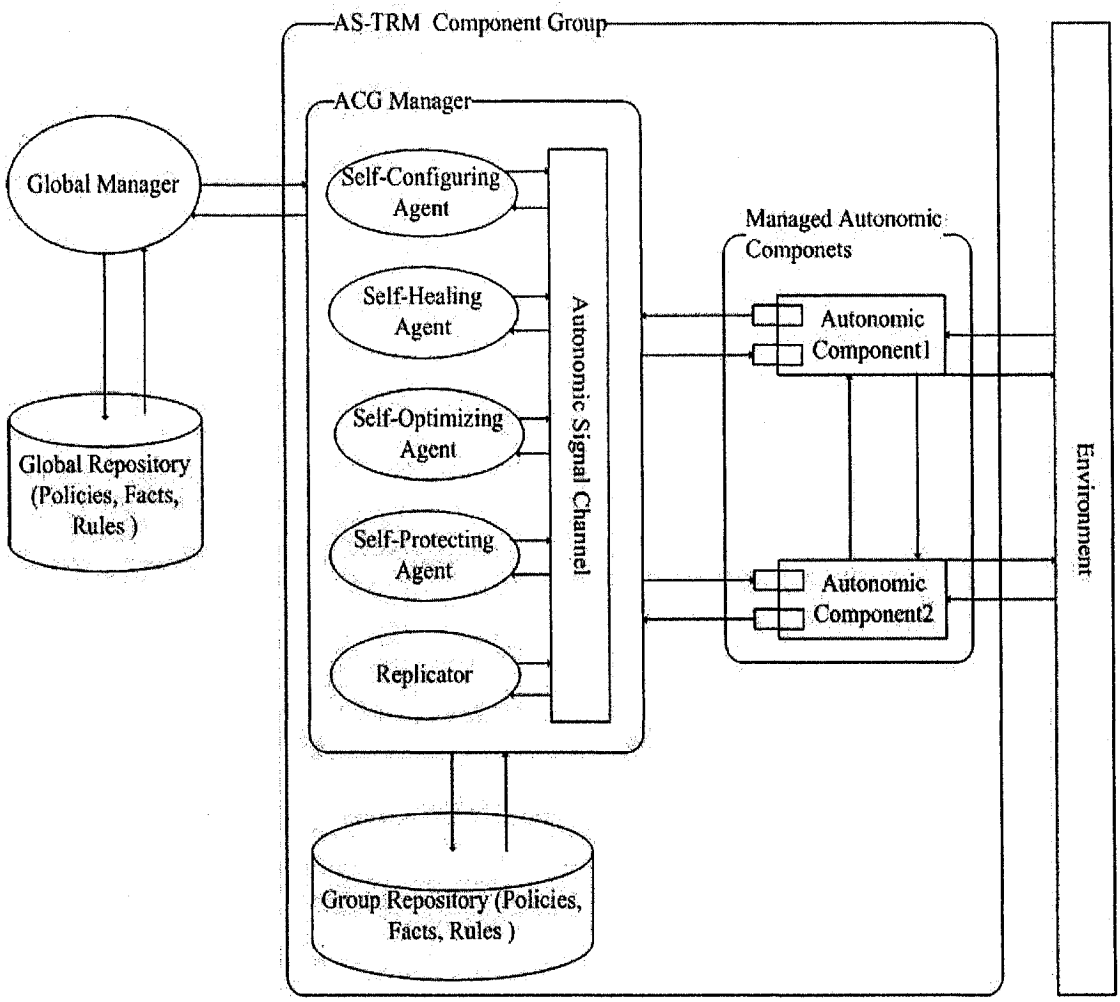

Figure 30: Architecture of AS-TRM Component Group

- Every ACG consists of an AGM and a set of managed ACs;
- An AGM consists of a collection of intelligent agents which is responsible for the autonomic behavior of self-configuring, self-healing, self-optimizing, as well as self-protecting, and a replicator for replicating the states of the ACs within the ACG;
- The intelligent agents in the AGM can communicate one another through the Autonomic Signal Channel;
- Each managed AC communicates its events and other measurements with AGM;

- According to the input received from the ACs, the AGM makes the decisions based on the policies, facts, and rules (stored in the ACG repository) and communicates the instructions with corresponding ACs.

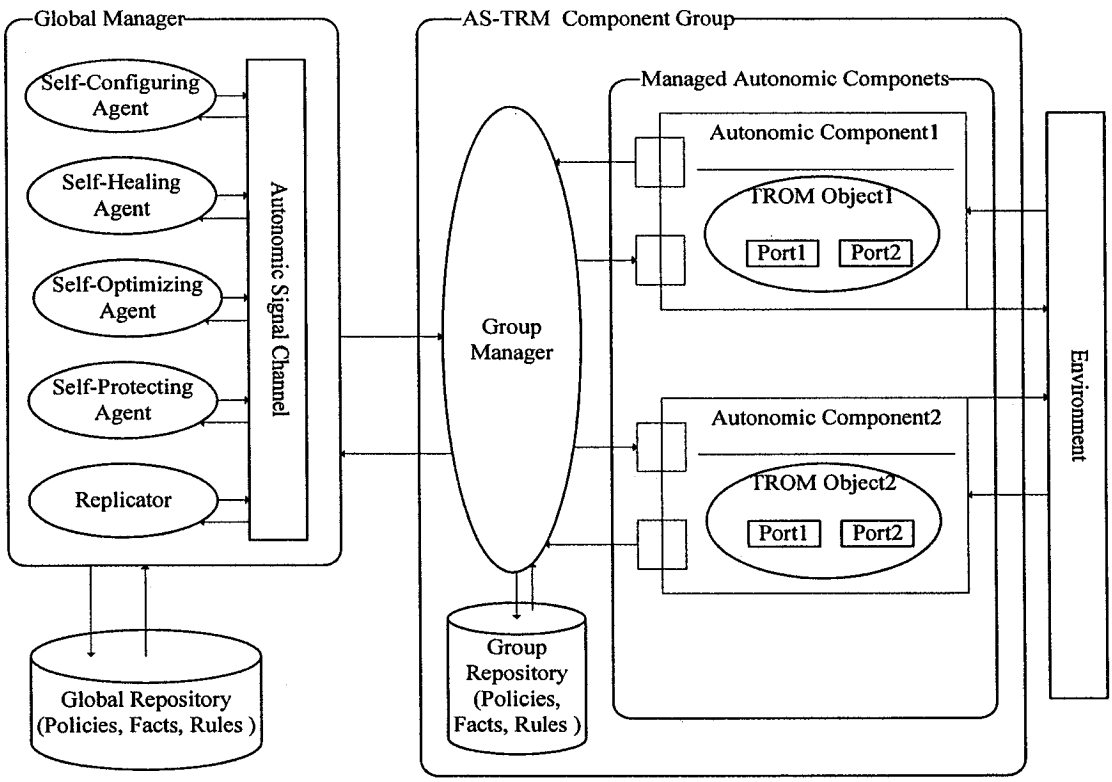

Figure 31: Anatomy of Global Manager and Autonomic Component

- A GM consists of a set of intelligent agents which is responsible for the autonomic behavior of self-configuring, self-healing, self-optimizing, as well as self-protecting, and a replicator for replicating the states of the ACGs within the AS-TRM system;
- The intelligent agents in the GM can communicate each other through the Autonomic Signal Channel;
- Every ACG communicates its events and other measurements with the GM;

- According to the input received from the ACGs, the GM makes the decisions based on the policies, facts, and rules (stored in the AS repository) and communicates the instructions with corresponding ACGs.

Both the interface between an AGM and its managed autonomic components, and the interface between a GM and an AGM are an important part of the architecture, which we state in the following sections.

5.4 Communication Mechanism of AS-TRM

The communication mechanism of the AS-TRM is implemented by the AS-TRM Communication System (ACS). ACS is the autonomic message system within AS-TRM, which provides the interfaces for asynchronous and synchronous message-delivery services [VKOP06]. Particularly, ACS is an application of the Demand Migration Framework (DMF) [VP05], which extends the DMF architecture by adding new features to adapt autonomic behavior in the AS-TRM. The primary objective of this extension is to transform the distributed asynchronous DMF into a distributed autonomic ACS.

5.4.1 Architecture

The architecture of ACS implies asynchronous and synchronous communication among the AS-TRM nodes which are AC, AGM, as well as GM. The asynchronous communication is inherited from DMF centralized message-persistent asynchronous communication [VP05], and the synchronous communication is a variant of peer-to-peer communication [B02]. We define the former as AC <-> AGM <-> GM, and it occurs between ACs and AGMs, as well as between AGMs and GM; the peer-to-peer

communication occurs between ACs, and it is called AC <-> AC communication [VKOP06]. Figure 32 shows the layered architecture of ACS which is derived from DMF.

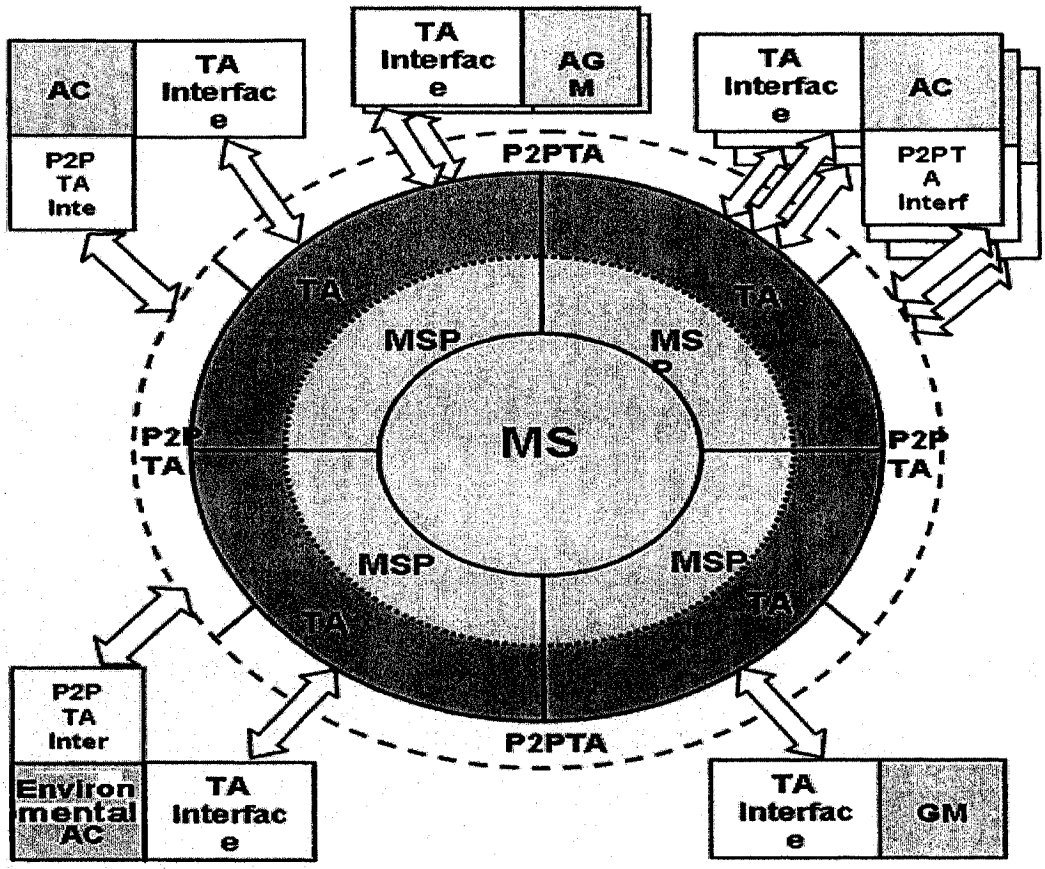

Figure 32: Architecture of ACS [VKOP06]

The architecture of ACS consists of four layers, which are Message Space (MS), Message Space Proxies (MSP), Transport Agents (TA), and Peer-to-Peer Transport Agents (P2PTA). MS, MSP, as well as TA are directly derived from DMF, and P2PTA is DMF's extension which addresses synchronous communication issues.

- MS: incorporates a persistent storage mechanism for all the messages which are exchanged asynchronously within the AS-TRM; moreover, it incorporates an Object Query language (OQL) [Emm00] to query the stored messages;

- MSP: the presentation layer which takes the MS functionality to a more generic level; there are single MS and multiple MSPs, and every MSP is associated with a TA;
- TA: the migration layer that transports messages asynchronously among ACs, ACGs, and GM through the interface; TAs are the independent components which can carry objects over the boundaries of machines; they provide a transparent form of the migration, and every TA works independently as well as concurrently with other TAs [VP05];
- P2PTA: provides an alternative way for communication that is the synchronous point-to-point communication. The ACs can establish a direct connection by using P2PTAs with their interfaces.

5.4.2 Functionalities

There are two kinds of messages communicating via the ACS [VKOP06]:

- Heartbeat message: used for self-monitoring; it provides a summary of the component state. The ACs send their state (heartbeat message) proactively and regularly to corresponding AGM, and the AGMs also send their state to the GM;
- Regular message: the message containing the information of AS-TRM work and configuration.

In addition, each AS-TRM message has its priority which is recognizable by the transport agents of TAs and P2PTAs. The priority mechanism guarantees that the message with higher priority can be delivered first.

According to the functional perspective mentioned above, the ACS should implement the following functionalities, and Figure 33 illustrates the use of ACS by the AS-TRM processing nodes of ACs, AGMs, as well as GM.

- Asynchronously sending and receiving heartbeat as well as regular messages;
- Asynchronously sending regular messages to a specified receiver, and reading regular messages from a specified reader;
- Asynchronously broadcasting regular messages to all ACs and AGMs;
- Synchronously sending and receiving heartbeat as well as regular messages.

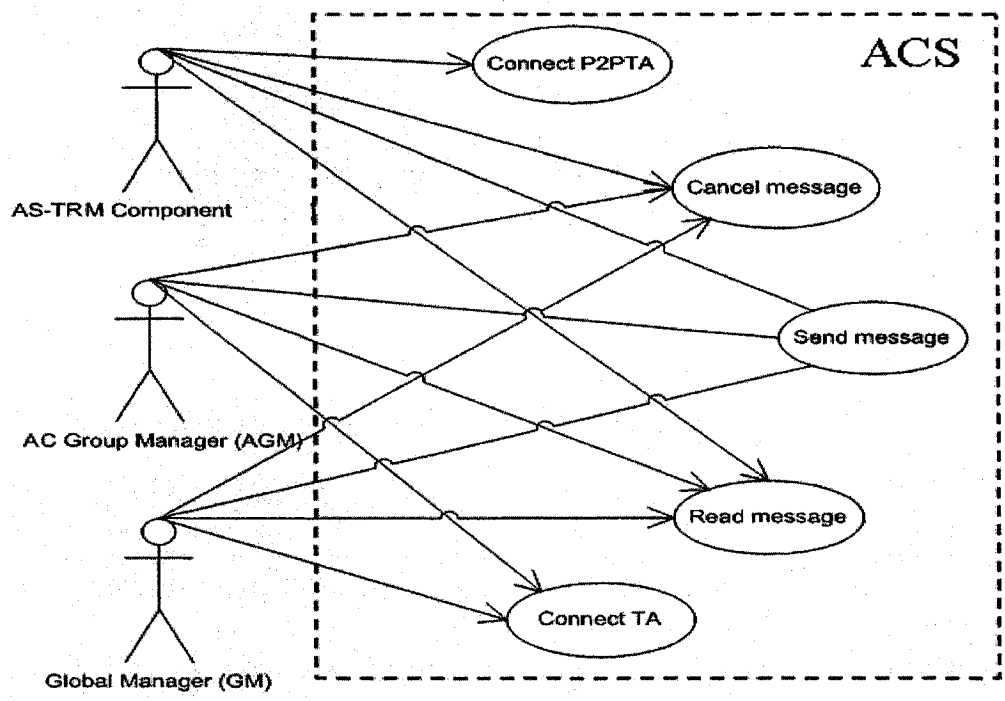

Figure 33: General Use Case of ACS [VKOP06]

5.4.3 Architecture of ACS Component

In order to implement the autonomic component within ACS, we extend the autonomous architecture of DMF [VP05] by adding a management unit to each element in the DMF.

The Management Unit (MU) controls and monitors the corresponding units of ACS, so every component of ACS is a peer of autonomic units which consists of a MU and the work unit of ACS. The former performs the control functions over the work unit of ACS, and the latter performs its work duty and proactively reports its state to the MU that can decide to shutdown or restart the work unit of ACS [VKOP06]. Figure 34 shows the architecture of ACS Component.

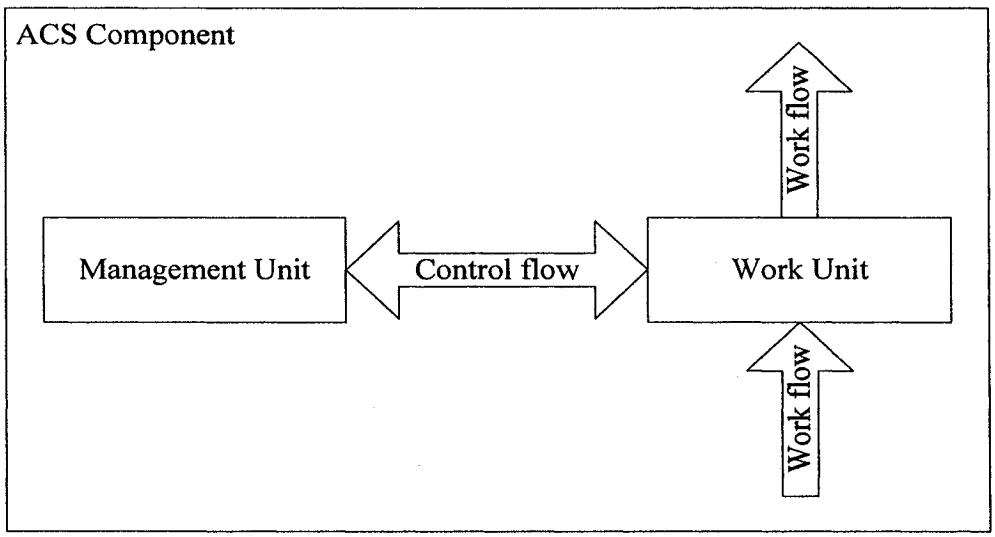

Figure 34: Architecture of ACS Component [VKOP06]

5.4.4 Autonomic Features

The ACS extends the DMF by adding the autonomic features of self-protection, self-optimization, self-configuration, and self-healing; some of them are already addressed by the DMF architecture, such as the core components of MS, TAs which work in the autonomous and independent mode [VP05].

- Self-Protection: only communication-trusted end-points can communicate via ACS, and the ACS provides the integrated security mechanism which can

prevent unauthorized access. This autonomic feature is derived from the DMF architecture [Vas05];

- Self-Configuration: the ACS is the distributed system with hot-plugging features; the TAs is capable of discovering available MS and then plugging into the ACS. This autonomic feature is inherited from the DMF architecture [VP05];
- Self-Healing: the components of ACS are able to restart themselves because of the embedded MU. The ACS inherently has the delivery semantics of at least once that prevents the message lost when it is sent asynchronously [Vas05], and this allows the components of ACS restart from the stop point.

5.5 Reliability Assessment of AS-TRM

The evolving nature of the AS-TRM requires the continuous monitoring of reliability levels to evaluate the risk of deploying a change on the configuration of the AS-TRM system and to identify potential safety hazards for the functionalities of the system.

5.5.1 Rationale of Reliability Assessment

Our reliability assessment methodology is concerned about the uncertainty of the architecture-based model, which is suitable for large complex distributed systems and applicable throughout the life cycle of software. The Markov model has been applied to the model reliability prediction for TROM real-time reactive systems [Orm02]. Because the AS-TRM formalism is an extension of the TROM formalism, the theoretical foundation of our reliability prediction model is the Markov model. The Markov model states that, given the current state of the system, the future evolution of the system is

independent of its history, which is also the main characteristic of reactive autonomic components. Furthermore, the analysis of Markov model yields the results for both the time-dependent evolution of the system, and the steady state properties of the system [VKOP06]. The reliability assessment of AS-TRM is performed at two levels which are peer group (ACGs) and system (AS); it is continuously evaluated by the corresponding AGMs and GM.

5.5.2 Reliability Assessment of ACG

In AS-TRM, the Markov model of an AC is the state diagram, where the states represent the states in the AC which are observable to the environment, and the transactions between states have assigned probabilities [VKOP06]. An algebraic representation of a Markov model is a matrix, namely transition matrix in which the rows and columns correspond to the states, and the entry p_{ij} in the i^{th} row, j^{th} column is the transition probability for being in state j at the stage following state i. The probabilities p_{ij} are calculated by the corresponding AGM as follows:

1. The initial probabilities for all the transitions in the state machine of an AC are calculated. The algorithm for calculating those probabilities for a state is based on the following assumptions:
    - All external events which can occur at the state have the identical as well as independent probability distributions;
    - All internal events which can occur at the state have the same probability;
    - These are in general different.

We assume the most common stochastic queuing model for the arrival time of the external events, namely a Poisson distribution.

2. In case there are more than one transitions with the same type (external/internal) from state i to state j, the transitions mentioned above are substituted by the one whose probability is:

$$P = 1 - (1 - P\{l_1\}) \times \ldots (1 - P\{l_n\})$$

The following property holds for the calculated probabilities:

$$\Sigma_j p_{ij} = 1$$

We create the Markov model of ACG in three steps as the following:

- Create the Markov models for ACs;
- Create the Markov models for every pair of synchronously interacting ACs within the ACG, where the interaction between two ACs is because of the shared (external) events:

  If all the transitions in a state are labelled by internal events, or if all of them are labelled by shared events, the probabilities are obtained by normalizing the probabilities in their respective machines. Nonetheless, if both internal events and shared events occur at the state, the probabilities of shared events are calculated first, and the remaining measure is distributed to the transitions labelled by internal events;
- Create the Markov model for fully configured group of synchronously interacting ACs:

If o_1, ..., o_n are the ACs in an ACG and M_1, ..., M_n are their corresponding transition matrices, the transition matrix M of that ACG is calculated as the following:

a) Calculate:

$$M = M_1 \otimes M_2$$

b) For j = 3 to n calculate:

$$M = M \otimes M_j$$

Where $\otimes$ indicate the direct product of two transition matrices.

The synchronous product machine dynamically changes when ACs join or leave their ACG, so the transition probability matrices also change, and they should be recalculated. The detailed description of the algorithms for creating Markov matrices is given in [Orm02].

We state that the reliability should be calculated from the steady state of the Markov system, and a steady state or equilibrium state is the one in which the probability of being in a state before and after transitions is the same as time progresses. We define the reliability prediction for an ACG configuration which consists of k ACs at the level of certainty quantified by the source excess-entropy as the following:

$$Reliability(Subsystem) = \sum_{i=1}^{k} H_i - H$$

Where

$$H = -\sum_i v_i \sum_j p_{ij} \log p_{ij}$$

Is the level of uncertainty of the Markov model corresponding to an ACG; v_i is the steady state distribution vector for corresponding Markov system, and the values of p_{ij} are the transition probabilities. H_i is the level of uncertainty in the Markov model corresponding to an AC. For a transition matrix P, the steady state distribution vector v satisfies the property of v*P = v.

The level of uncertainty H is exponentially related to the number of paths which are "statistically typical" of the Markov system. Therefore, higher entropy value implies that more sequences must be created for accurately illustrating the asymptotic behaviour of the Markov system.

5.5.3 Reliability Assessment of AS

The reliability prediction of an AS, which is calculated by its Global Manager, is defined as the least reliability measure value among its m ACGs (subsystems) [VKOP06]:

$$Reliability(AS) = min\{Reliability(ACG_i)\}_i^m$$

Higher value of reliability measure implies less uncertainty within the model and thus higher level of software reliability.

The Markov model of a configured AS changes when the AS-TRM architecture evolves; the calculation of the Markov matrix for reconfigured AS allow the comparison of their configurations based on reliability prediction. If the system configuration of AS_{j-1} changes to AS_j, we need to calculate the reliability of the configuration AS_j and compare it with the configuration of AS_{j-1}:

$$Reliability(AS_{j-1}) = min\{Reliability(ACG_i)\}_i^m$$

Where ACG_i is a group of AS_{j-1}, and

$$Reliability(AS_j) = min\{Reliability(ACG'_i)\}_i^m$$

Where ACG'_i is a group of AS_j, If

$$Reliability(AS_j) \geq Reliability(AS_{j-1})$$

As a result, the uncertainty level of reconfigured AS is less than the one of current AS.

The reliability assessment at AS level allows reconfigured system to be deployed by its GM if the minimum required level of reliability is reached.

5.6 Summary

In this chapter, we have specified AS-TRM's architecture based on the architecture of Multi-Agent systems. AS-TRM's architecture consists of Autonomic Component, AS-TRM Component Group Manager, and Global Manager. The intelligent agents within the ACG Manager and Global Manager are responsible for implementing autonomic behavior like self-configuring, self-healing, self-optimizing, and self-protecting. Then, we have built the architecture of AS-TRM Communication System for implementing communication among AS-TRM's elements. The AS-TRM Communication System has been extended from the Demand Migration Framework. Finally, we have established reliability assessment model of AS-TRM based on Markov Model for monitoring AS-TRM's evolution.

Chapter 6: Conclusion and Future Work Directions

This thesis work is our first step towards extending the TROMLAB framework by adding the specification of distributed reactive autonomic components along with their relationships, and the non-functional properties constraining the behavior of the system. Particularly, it addresses: 1) extending TROM to AS-TRM for supporting distributed autonomic behavior; 2) defining AS-TRM's characteristics for determining AS-TRM systems' requirements, design, and implementation; 3) building AS-TRM's architecture and communication mechanism for implementing both autonomic and real-time reactive functionalities; 4) modeling AS-TRM's reliability assessment for monitoring AS-TRM's evolution.

6.1 From TROM to AS-TRM

We have built a five-tier based architecture for the AS-TRM formalism's implementation in the future, which formalism is the formal foundation for integrating distributed autonomic behavior into the TROM methodology:

- The AC tier encapsulates the TROM objects into the AS-TRM autonomic components. for undertaking a complete or partial real-time reactive task as a worker within the system;
- The ACG tier is a set of synchronously communicating ACs, which can support autonomic behaviour at the peer group level and accomplish a complete real-time reactive task independently. The self-monitoring behaviour at this tier is implemented by the AGM which: 1) monitors the reliability level of ACG; 2)

verifies the behavior of ACs; 3) receives the diagnostic messages and sends the treatment messages to ACs; 4) adds and removes the ACs from the ACG; 5) automates the initialization and maintenance; 6) manages all the data shared between ACs;

- The AS tier is the abstracting a set of asynchronously communicating ACGs, which can support autonomic behaviour at the system level. The self-monitoring behaviour at this tier is implemented by the GM which: 1) monitors the reliability level of AS; 2) verifies the behavior of the ACGs; 3) verifies user access; 4) receives the diagnostic messages and sends the treatment messages to ACGs; 5) forwards configuration changes to the AGMs; 6) manages all the data shared between ACGs.

6.2 Characteristics

AS-TRM is the formal framework for autonomic distributed real-time reactive systems which leverages their modeling, development, integration, maintenance, and continuous monitoring of their reliability; AS-TRM is self-managed, distributed, proactive, and evolving.

6.3 Architecture

The architecture of AS-TRM consists of ACs, AGMs, and GM which are connected to each other at the local, peer group, and system levels.

- Every ACG consists of an AGM and a set of managed ACs to implement the autonomic behavior at the peer group level;

- An AGM consists of a collection of intelligent agents for implementing self-configuring, self-healing, self-optimizing, as well as self-protecting, and a replicator for replicating the states of the ACs within the ACG;
- According to the input received from the ACs, the corresponding AGM makes the decisions based on the policies stored in the ACG repository;
- A GM consists of a set of intelligent agents to implement self-configuring, self-healing, self-optimizing, as well as self-protecting at the system level, and a replicator for replicating the states of the ACGs within the AS.
- According to the input received from the ACGs, the GM makes the decisions based on the policies stored in the AS repository;
- The intelligent agents in the AGMs and GM can communicate each other through the Autonomic Signal Channel.

6.4 Communication Mechanism

ACS is the autonomic message system within AS-TRM, which provides the interfaces for asynchronous and synchronous message-delivery services. It is an application of the DMF, which extends the DMF architecture by adding new features to adapt autonomic behavior in the AS-TRM.

The architecture of ACS consists of four layers which are MS, MSPs, TAs, and P2PTAs. The layers of MS, MSPs, as well as TAs are directly derived from the DMF that implement asynchronous communication, and the P2PTAs layer is the extension of DMF that implement synchronous communication.

- MS incorporates a persistent storage mechanism for all the messages which are exchanged asynchronously within the AS-TRM;
- MSP is the presentation layer making the MS functionality more generic;
- TA is the migration layer that transports messages asynchronously among ACs, ACGs, and GM through the interface;
- P2PTA provides an alternative way for communication that is the synchronous point-to-point communication.

The functionalities consist of: 1) asynchronously sending and receiving heartbeat as well as regular messages; 2) asynchronously sending regular messages to a specified receiver, and reading regular messages from a specified reader; 3) asynchronous sending regular messages to all ACs or AGMs within the AS-TRM (broadcasting); 4) synchronously sending and receiving heartbeat as well as regular messages.

The ACS extends the DMF by adding the following autonomic features:

- Self-Protection: ACS provides the integrated security mechanism which can prevent unauthorized access;
- Self-Configuration: ACS is the distributed system with hot-plugging features;
- Self-Healing: the components of ACS are able to restart themselves because of the embedded MU.

6.5 Reliability Assessment

Our reliability assessment methodology is concerned about the uncertainty of the architecture-based model, which is suitable for large complex distributed systems and

applicable throughout the life cycle of software. The reliability assessment of AS-TRM is based on the model reliability prediction for TROM real-time reactive systems, and performed at two levels which are peer group (ACGs) as well as system (AS); it is continuously evaluated by the corresponding AGMs and GM.

- We create the Markov model of ACG for: 1) ACs; 2) every pair of synchronously interacting ACs within the ACG; 3) fully configured group of synchronously interacting ACs;
- The reliability prediction of an AS, which is calculated by its GM, is defined as the least reliability measure value among its m ACGs (subsystems).

Higher value of reliability measure implies less uncertainty within the model and thus higher level of software reliability; moreover, the calculation of the Markov matrix for reconfigured AS allows the comparison of their configurations based on reliability prediction, and allows the reconfigured system to be deployed by its GM if the minimum required level of reliability is reached.

6.6 Future Work Directions

Some of the future extensions to this thesis work include the following aspects:

- Extends the animation tool for the validation of the TROM formalism to adapt the AS-TRM formalism;
- Extends the Larch prover for the formal verification of the TROM formalism to adapt the AS-TRM formalism;

- Extends the operational and logical semantics of the TROM formalism to adapt the AS-TRM formalism;
- Implements the architecture of AS-TRM by using appropriate toolkits and platforms for building multi-agent systems;
- Extends the implementation model and automated code generation of the TROM design model to adapt the AS-TRM design model;
- Migrates the target implementation environment from Real-Time Java [Zha00] to AspectAda [PC05] because: 1) Ada was inherently designed for the concerns of program reliability, maintenance, efficiency, flexibility, extensibility, and additional control over storage management as well as synchronization which are the key requirements for implementing autonomic real-time reactive systems; 2) AspectAda, which is extended from Ada95, provides the powerful language elements to facilitate the aspect oriented programming that can catch the crosscutting concerns within concurrent systems.
- Builds a set of measurement and metrics to evaluate the quality of AS-TRM and its evolution.

References

[AAG93] G. Abowd, R. Allen, and D. Garlan, "Using Style to Understand Descriptions of Software Architecture", In Proceedings of the SIGSOFT '93: Symposium on the Foundations of Software Engineering, pp. 9-20, December, 1993.

[AAM96] V. S. Alagar, R. Achuthan, and D. Muthiayen, "TROMLAB: An Object-Oriented Framework for Real-Time Reactive System Development", Technical Report, Department of Computer Science, Concordia University, Montreal, 1996 (first draft), June 2000 (revised).

[Ach95] R. Achuthan, "A Formal Model for Object-Oriented Development of Real-Time Reactive Systems", PhD thesis, Department of Computer Science, Concordia University, Montreal, Canada, October 1995.

[Anthill01] http://www.cs.unibo.it/projects/anthill

[B02] D. Brookshier et al, "JXTA: Java P2P programming", Indianapolis, Sams Publishing, 2002.

[B03] D. F. Bantz et al, "Autonomic personal computing", IBM Systems Journal, Vol. 42, No 1, pp. 165-176, 2003.

[Che02] Chen M., "Implementation of Specification-Based Testing System for Real-Time Reactive System in the TROMLAB Framework", Master Major Report, Department of Computer Science, Concordia University, 2002.

[Cro06] I. Croitoru, "Autonomic Systems Modeling Development: A Survey",

Master Major Report, Department of Computer Science, Concordia University, April 2006.

[Emm00] W. Emmerich, "Engineering Distributed Objects", Baffins Lane, Chichester, Wiley, 2000.

[GS02] D. Garlan., B. Schmerl, "Exploiting Architectural Design Knowledge to Support Self-Repairing Systems", In Proceedings of the 14th International Conference on Software Engineering and Knowledge Engineering, pp. 241-248, July 2002.

[Hai99] G. Haidar, "Simulated Reasoning and Debugging of TROMLAB Environment", Master Thesis, Department of Computer Science, Concordia University, December 1999.

[HGC04] J. Hu, J. Gao, J. Chen, "Multi-Agent System based Autonomic Computing Environment", In Proceedings of the Third International Conference on Machine Learning and Cybernetics, Shanghai, Volume 1, pp. 105-110, 26-29, August, 2004.

[Hor01] P. Horn, "Autonomic computing: IBM perspective on the state of information technology", IBM T. J. Watson Laboratories, NY, 15th October 2001, Presented at the AGENDA 2001 Conference, Scottsdale AR.

[IBM03] IBM Corporation, "An architectural blueprint for autonomic computing", IBM and autonomic computing, 2003.

[IBM04] IBM Corporation, "An architectural blueprint for autonomic computing", IBM

and autonomic computing, 2004.

[IBM05] IBM Corporation, "An architectural blueprint for autonomic computing", IBM and autonomic computing, 2005.

[IEEE98] IEEE Std 1061-1998, "Software Quality Factor-Criteria-Metrics Framework", 1998.

[Lee03] Lee F. A., "Reliability Measurement Based on the Markov Model for Real-Time Reactive Systems: Design and Implementation", Master Major Report, Department of Computer Science, Concordia University, 2003.

[Liu03] Liu S. H., "Simulated Validation of Real-Time Reactive Systems with Parameterized Events", Master Thesis, Department of Computer Science, Concordia University, August 2003.

[LML05] Paul Lin, Alexander MacArthur, John Leaney, "Defining Autonomic Computing: A Software Engineering Perspective", Proceedings of the 2005 Australian Software Engineering Conference (ASWEC'05), pp. 88-97, 2005.

[LPPC03] G. Lafranchi, P. Della Peruta, A Perrone, and D. Calvanese, "Toward a new landscape of systems management in an autonomic computing environment", IBM Systems Journal, Volume 42, No.1, pp. 119-128, 2003.

[MH04] J. A. McCann, M. C. Huebscher, "Evaluation issues in autonomic computing", In Proceedings of the Grid and Cooperative Computing Workshops (GCC), pp. 597-608, 2004.

[Moh04] M. Mohammad, "Visualization Animation for Real-Time Reactive Systems Simulation", Master thesis, Department of Computer Science, Concordia University, Montreal, Canada, August 2004.

[Mur04] R. Murch, "Autonomic Computing", Prentice Hall Professional Technical Reference, IBM Press, pp. 119-132, 2004.

[Mut96] D. Muthiayen, "Animation and Formal Verification of Real-Time Reactive Systems in An Object-Oriented Environment", Master thesis, Department of Computer Science, Concordia University, Montreal, Canada, October 1996.

[Mut00] D. Muthiayen, "Real-Time Reactive System Development – A Formal Approach Based on UML and PVS", Ph.D. Thesis, Department of Computer Science, Concordia University, January 2000.

[Nag99] R. Nagarajan, "Vista – A Visual Interface for Software Reuse in TROMLAB Environment", Master Thesis, Department of Computer Science, Concordia University, April 1999.

[OS02] http://oceanstore.cs.berkeley.edu/

[Orm02] O. Ormandjieva, "Deriving New Measurements for Real-Time Reactive Systems", Ph.D. Thesis, Department of Computer Science, Concordia University, April 2002.

[PK00] J. Paquet, P. G. Kropf, "The GIPSY Architecture", LNCS, Vol. 1830, pp. 144-153, 2000.

[P02] D. Patterson et al, "Recovery Oriented Computing (ROC): Motivation, Definition, Techniques, and Case Studies", Technical Report CSD-02-1175, University of California-Berkeley, March 2002.

[Pom99] F. Pompeo, "A Verification Assistant for TROMLAB Environment", Master Thesis, Department of Computer Science, Concordia University, September 1999.

[Pop99] O. Popistas, "Rose-GRC Translator: Mapping UML Visual Models onto Formal Specifications", Master Thesis, Department of Computer Science, Concordia University, April 1999.

[SB03] R. Sterritt, David W. Bustard, "Towards an Autonomic Computing Environment", DEXA Workshops, pp. 699-703, 2003.

[Sri99] V. Srinivasan, "An Intelligent Graphical Interface System for TROMLAB", Master Thesis, Department of Computer Science, Concordia University, December 1999.

[T03] R. Telford et al, "Usability and design considerations for an autonomic relational database management system", IBM Systems Journal, Vol. 42, No 1, pp. 568-581, 2003.

[T04] G. Tesauro et al, "A Multi-Agent Systems Approach to Autonomic Computing", In Proceeding of the 3rd International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS 2004), pp. 464-471, August 2004.

[Tao96] H. Tao, "Static Analyzer: A Design Tool for TROM", Master Thesis, Department of Computer Science, Concordia University, August 1996.

[Vas05] E. Vassev, "General Architecture for Demand Migration in the GIPSY Demand-Driven Execution Engine", Masters Thesis, Department of Computer Science, Concordia University, June 2005.

[VKOP06] E. Vassev, H. Kuang, O. Ormandjieva, J. Paquet, "Reactive, Distributed and Autonomic Computing Aspects of AS-TRM: A Means of Achieving Reliability", Technical Report, Department of Computer Science, Concordia University, Montreal, February 2006.

[VP05] E. Vassev, J. Paquet, "A Generic Framework for Migrating Demands in the GIPSY' Demand-Driven Execution Engine", In Proceedings of the 2005 International Conference on Programming Languages and Compilers (PLC 2005), Las Vegas, USA, pp. 29-35, June 2005.

[WMK03] H. Watherspoon, T. Moscovitz, J. Kubiatowicz, "Introspective Failure Analysis: Avoiding Correlated Failures in Peer-to-Peer Systems", In Proceeding of the 21st. Symposium on Reliable Distributed System (SRDS 2002), pp. 362-, 2002.

[WPT03] R. Want, T. Pering, D. Tennenhouse, "Comparing autonomic and proactive computing", IBM Systems Journal, Volume 42, No.1, pp. 129-135, 2003.

[WSG05] J. White, D. Schmidt, A. Gokhale, "Simplifying the Development of Autonomic Enterprise Java Bean Applications via Model Driven

Development", Department of Electrical Engineering and Computer Science, Vanderbilt University, Nashville, 2005.

[Zha00] L. Zhang, "Implementing Real-Time Reactive Systems from Object-Oriented Design Specifications", Master Thesis, Department of Computer Science, Concordia University, 2000.

[Zhe02] M. Zheng, "Automated Generation of Test Suits from Formal Specification of Real-Time Reactive Systems", Ph.D. Thesis, Department of Computer Science, Concordia University, 2002.

[Zhu03] M. Zhuo, "Real-Time Reactive System Measurement Tool TROM-QM: Design and Implementation", Master Major Report, Department of Computer Science, Concordia University, 2003.