Self-adaptive systems: Overview and Approaches

Lorena Castañeda Bueno, Norha M. Villegas, Gabriel Tamura

Icesi University, Cali, Valle del Cauca, Colombia
{lcastane, nvillega, gtamura}@icesi.edu.co

Abstract. Nowadays, businesses objectives are highly affected by complex and dynamic environments. Thus, as IT strategies are defined to support the achievement of organizational goals, new IT practices have to be implemented to address the dynamic evolution of businesses. Within this dynamics, computing and software systems have became more complex as they are required to support business strategies under changing requirements, heterogeneous and distributed platforms, and uncertain environments. In this endeavor, autonomic computing is a promising direction where systems are able to manage themselves by adapting their behavior according policies given by administrators. This survey article presents an overview of foundational elements and different approaches for implementing autonomic software systems.

1 Introduction

For over the past ten years, unexpected changes in the business requirements have been affecting the organizations. Continuous changes in customer preferences and technology, are affecting the business objectives, forcing the organizations to take decisions over the company's present and future. Those organizations that have survived to this phenomenon are known as emergent organizations. However, along with these organizations, the software systems which purpose was to support the organization's stability now is required to be flexible enough to support the changes and therefore, to be able to adapt in order to meet the new business goals [1].

Over time, software intensive systems inside the emergent organizations had rapidly grown and even evolved. As a consequence, *Ultra-Large Scale Systems (ULS-S)* appeared as very big systems not only because their data capacity or the number of lines of code, but also, because of every activity in the dimensions of construction, implementation and maintenance. Therefore, an ULS-S is a very high scale and complex software system. Indeed, its size and nature characterize ULS-S by the decentralization of its development process and its runtime environment. In addition, an ULS-S is the sum of many independent elements that make it grow as new elements are added [2].

As it has been noted, ULS-S must be in a continuous evolution and deployment in order to integrate new capabilities. Moreover, the ULS-S requires experts to manage it, that is to install, maintain, configure and optimize the system. Those managing requirements had led organizations, to high costs and long time activities. In fact, IT professionals spend most of their time fixing problems, installing and integrating new components and understanding the complexity of the system. Therefore, as the context is unpredictable, engineers will be also monitoring and analyzing the system every time searching for possible faults [3].

A fundamental need for organizations is a low cost solution to manage these complex systems. In order to do that, some routine managing tasks over the system must be automated. This automation requires for the system the ability to *manage itself*. Therefore, systems must coordinate configuration, healing, optimization and protections tasks without human intervention and guided by policies. These kind of systems able to perform self-management are called *autonomic systems* [4, 3].

Of course, self-management implies not only for the system to be *context-aware* but also to perform some adaptation tasks. Following this, is necessary the performance of four tasks: **monitoring** the internal and context, **analyzing** the data gathered to identify symptoms, **planning** a set of adaptive actions, and **executing** those actions over the system [5, 4, 3, 6].

One of the most important challenges in self-adaptation is to create the ability in the system to reason about itself and the context. Control engineering uses feedback to measure and act upon the behavior of the system. Moreover, *feedback loops* become important in the process that the system must follow to read an output property of the system and then evaluate if the desired properties are being reached [7]. Software engineers have made significant efforts to propose self-adaptive systems. These proposals addresses different approaches based on feedback loops, components and even hybrid architectures, and a set of standards to communicate these systems[8, 9, 7, 5, 10–15]. However, all these approaches are not sufficient for the different systems, data types, and configurations that are available. Often, this proposals merge self-adaptation with the system making very difficult to analyze the adaptation capability as an independent system, and feedback loops are not explicit in must of these proposals.

This survey article, presents a description of self-adaptive system proposals, identifying for each one, the type of approach, and the elements to meet context-aware and self-adaptation. As a consequence it will help to identify the advantages of the most relevant approaches for self-adaptive systems.

Furthermore, this article will study the state of art in self-adaptive system proposals. Section 2 introduces the concepts of Ultra Large Scale Software Systems. Section 3 describes what it is Autonomic Computing and its elements and properties. Also describes the fundamentals of adaptive software. Section 4 introduces the importance of feedback loops into self-adaptive systems. Section 5 presents the most representative approaches into self-adaptive systems, a brief description and the exhibition of the adaptation mechanism for each one. Section 6 describe the three reference architectures studied for self-adaptive systems. Finally, Section **??** presents some challenges for autonomic computing, and Section 7 presents the conclusions in this article.

2 Ultra Large Scale Software Systems

Each organization expects software systems to support the production processes and also help the management with useful information, which increasingly has large volumes of data storage and processing. The analysis of this data must somehow help management areas to make decisions about the company, and so, the management expects some quality aspects, like efficiency, accuracy and availability.

As time passes, the organizations, which are no longer stable and static, on the other hand, are constantly changing and evolving, expect their software systems do so and at the same time to be able integrate with the entire organization, who no longer is geographically placed in one location, because most of the time the information is also needed outside the company; employees working from outside, external meetings, franchises, and many other where the organization information is needed. The systems cease to be centralized and also begin to integrate with other systems. The company expects that all systems of different areas to communicate and share information and also expected to be integrated into future areas throughout the company including software in their systems with some quality service expectations.

The Ultra-Large Scale System is a very high scale and complex software. Its big size is in every aspect: number of lines of code, hardware, peoples using the software, amount of data being stored, processed and analyzed, number of connections from other software components, and the number of processes from the company that gives support [2]. Because of its size and nature, an ULS-S has to be decentralized not only in runtime but also in development.

3 Autonomic computing

The context is unpredictable and organizations are forced to change the business goals and therefore the software systems must meet those requirements in a very short time. Under these conditions, it is nearly impossible for software engineers to design a non error prone system. Therefore, engineers are also monitoring and analyzing the system every time searching for possible faults. Subsequently, most the IT budget is spent in prevention and recovery from failures [4]. As a result, organization need a low cost solution for the management of these complex systems and then free it administrators of this managing and routine tasks. It is necessary to build systems that can be more self-manageable. In order to do that, the systems must have characteristics stated as follows [4]:

- The system needs to *know itself*, therefore, the components in the system must have an identity.
- The system must know the context surrounding its activity.
- The system must coexist with foreign components, that is, operating with open standards.
- The system should know its purpose and the business goals that is pursuing.

An autonomic system is a system that can manage itself given some kind of decision making mechanism, such as policies, from administrators [3]. An autonomic system is composed of many self-managed components that interact with each other autonomously. The relationship among these components must be controlled by themselves and with no need for human interaction besides giving the policies that ruled their behavior. Furthermore, an autonomic system must be able to maintain and adjust their operations according to changing external or internal conditions, and guarantee the alignment with the business goals.

The fundamental characteristic of an autonomic system, is self-management, in other words, the capability to run routine tasks and maintenance without the intervention of the system administrator. The system must be continuously monitoring itself to be aware of changes in the system that might require either reconfiguration or optimization of the components, protecting itself against suspected behavior or recovering from failures [3]. In order to accomplish this self-managing behavior, systems must implement four fundamental features: **self-configuring**, the ability to adapt to the shifting environment itself in accordance with high-level policies align to the business goals and given by system administrators, **self-healing**, the ability to recover itself after a disruption in the system and minimize outages to keep the software system up and available, **self-protecting**, the capability to predict, detect, recognize and protect itself against malicious attacks and unplanned cascading failures, and **self-optimizing**, as the way system improves its operations[4, 3].

3.1 Autonomic computing properties

The fundamental characteristic of an autonomic system, is self-management, in other words, the capability to run routine tasks and maintenance without the intervention of the system administrator. The system must be continuously monitoring itself to be aware of changes in the system that might require either reconfiguration or optimization of the components, protecting itself against suspected behavior or recovering from failures [3].

In order to accomplish this self-managing behavior, systems must implement four fundamental features: self-configuring, self-healing, self-protecting and self-optimizing [4, 3].

1) Self-configuring

It responds to the ability to adapt to the shifting environment itself in accordance with high-level policies align to the business goals and given by system administrators. The self-configuring property must be online all the time, that means, the required changes must be implemented during runtime. Moreover, new features, software or hardware, could be added to the system without disruption in the services given by the system. Each time a new component is added, it will learn about the system and register itself. Furthermore, the component will know some specifications required and configure itself in accordance. In those cases, the system will be aware of the new component and the rest of the system would take the necessary actions to reconfigure itself to meet the new services.

2) Self-healing

It refers to the ability to recover itself after a disruption in the system and minimize outages to keep the software system up and available. First, the system must detect the failure, isolate it and replacing it for a fixed component, this could be either an older version of the component or a new one. In order to know a failure, the systems must use the knowledge base of the component and look for policies to apply. This knowledge base must be updated after the recovery for a future purpose. The intention of self-healing is also proactive, that is to say, that the system should be able to predict problems and take actions to prevent a failure based on the component behavior and the information stored in the knowledge base.

3) Self-protecting

The self-protection capacity is to predict, detect, recognise and protect itself against malicious attacks and unplanned cascading failures. The system must be able to provide secure capabilities to user access.

4) Self-optimizing

It refers to seek the way of improve its operations. The system would monitor, experiment and tune its parameters to learn about appropriate choices. That is, efficiently maximize the utilization of the resources without the intervention of system administrators. This kind of capabilities, should lead the system to make itself more efficient in performance or cost.

3.2 Adaptive software

he adaptation refers to the capability of changing the software structure and behavior according to the alterations in the environment. This recomposition must occur dynamically and often during runtime. The adaptive property can be seen on mobile devices, which must adapt to different connectivity conditions but keeping quality attributes like battery and signal. Furthermore, this adaptive capability is necessary on the systems that are vital survive to hardware failures and security attacks [5].

Software adaptation can be implemented either by parameter or by composition among others. Parameter adaptation alters the software variables responsible for the system behavior. However, does not allow new components or algorithms to be added after deployment. On the other hand, compositional adaptation exchange algorithms or components with other elements in the system to fit the current environment [5].

The dynamically adaptation provides flexibility to the system because can adjust itself during runtime. However, this flexibility is very difficult to design and program for software engineers. Mckinley et al. proposal includes three techniques to implement reconfigurable software: *separa-*

tion of concerns, computational reflection and component-based design[16, 5]. The **Separation of concerns** refers to the separation in development of an application's functional behavior from the non-functional requirements. It simplifies development and maintenance and promotes the reuse software components. The **Computational reflection** refers to the software's ability to reason about its behavior and possibly to alter it. There are two activities for reflection: *introspection* that allows a system to observe itself, and *intercession* that allows the system to modify its own behavior. Furthermore, it supports two kinds of reflection: *structural*, where relates to data types, interconnections and class hierarchy, and *behavioral*, where relates to the computational semantic. The **Component-based design**, relates to software units that are developed, deploy and compose by third parties independently. This technique supports two types of composition: *static*, where many components are combined to produce an application, and *dynamic*, where the components can be manipulated during runtime. Despite the techniques named before, it is important to know the how, when and where, composition takes place in order to design a proper adaptive solution[5].

As it has been noted, autonomic systems require a generic basic architecture able to provide the enough level of abstraction and generality to deal with the challenges posed for autonomic systems. An architectural approach is necessary because it offers generality, level of abstraction, potential scalability and integration [9].

3.3 Adaptation elements

Self-adaptive systems are characterized for the ability to reason about their state and the environment. This reasoning involves four key activities: collect, analyze, decide and act. Figure No.1 depict these activities as a closed loop[6].

1. The loop starts with the *collection* of relevant information taken from the environment and the system itself. Not all the information is important to be collected, so is necessary a mechanism to discriminate the important data to collect. The collecting takes place through *sensors* connected to the system and the external environment. This activity is also called the *Monitor*
2. The second step is to *analyze* the collected data and reasoning about it. This analysis provide some results about the current state of the system. This activity is also called the *Analyzer*
3. Next, according to those results, the system *decides* whether to adapt the system in order to achieve the desire state. This activity is also called the *Planner*
4. Finally, the system *acts* according the decisions made via *effectors* connected to the system. It is important in this step, to decide the right moment to execute those actions and the risks taken during this execution. This activity is also called the *Executer*

4 Feedback loops: a reference architecture for self-adaptation

As it has been noted, self-adaptive systems decide autonomously about the changes the system needs to perform. In order to take those decisions, the system must be aware of the context and of itself. Hence, the system must know the output of its actions and use it as feedback to prone relevant information for the decision to be made[17]. Control theory is about knowing the characteristics of the system. It deals with understanding how the desired input is affected by disturbance and noise reflected over the measured output. In computing systems, control theory is necessary knowing that systems have desired output characteristics all over its operation[7].

Fig. 2 shows a SISO[1] feedback control system, whose elements are described as follow:

[1] SISO: Single-Input, Single-Output

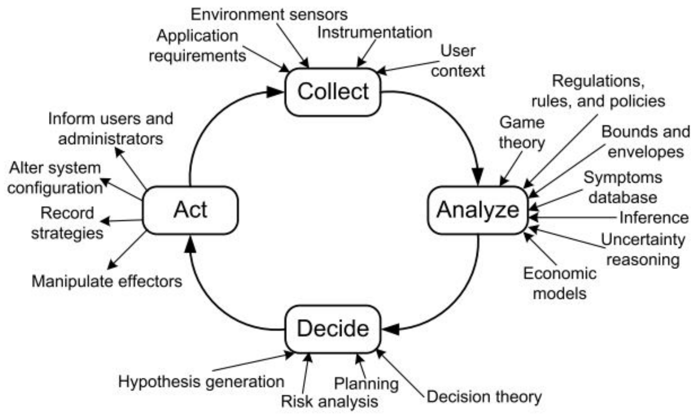

Fig. 1. Activities of the control loop [6]

- *Reference input*: This is the desired control objectives to be achieved. Is a reference that is used by the control system to validate if the objectives are being reached.
- *Control error*: It is the result of the comparison between measured output and the reference input. In other words, it expresses how differs if the output is not the desired.
- *Target System*: It is the system to be controlled.
- *Controller*: This element is responsible computing the control input for the target system to achieve the reference input.
- *Control input*: It is the information dynamically produced by the controller to affect the behavior of the target system.
- *Disturbance and Noise*: These two elements correspond to context. The disturbance is any changes in the external environment that affect the target system. The noise is any effect inside the target system that affects it.
- *Measured Output*:The measures of characteristics of the system that is required to control.
- *Transducer*: This is an optional element. This transducer transforms the measured output to the same measurement units of the reference input, so they can be compared.

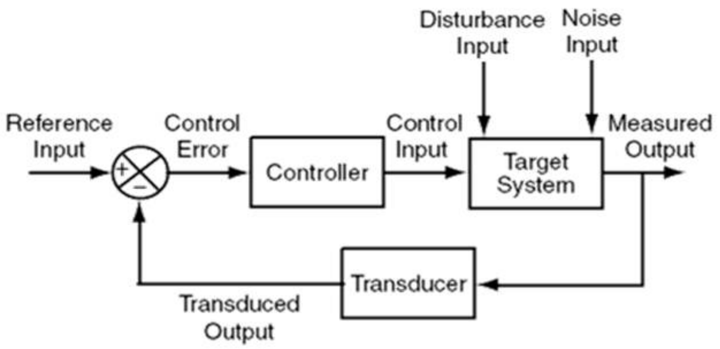

Fig. 2. Block diagram of a feedback control system [7]

Feedback loops are important for this project because provide the generic mechanism required for self-adaptation. Feedback loops are important in software processes to manage and support software evolution. This generic feedback loop represents four main activities: First the monitor probes *collects* the relevant data through sensors to reflect the current state of the system. Second, it *analyzes* the collected searching for symptoms about the desired behavior and the current state. Third, it *decides* whether it is necessary to adapt the system to reach the control objectives. And last, *implements* the decisions through effectors[18]. The steps in the feedback loop cycle are very similar to the tasks of autonomic computing.

The main reason to implement feedback loops into computing systems is to reduce the effects of uncertainty that appear in different forms as a consequence of the shifting context and the changes in business goals. Therefore, is necessary to make explicit feedback loops in the design of autonomic systems in order to reason about its behavior and to analyze the adaptation mechanism separately from the target system operation.

One of the most common models of feedback loops, and relevant for this project is the model MRAC[2] shown in Fig.3. The main achievements from this model is the flexibility and leverage in the feedback mechanism by defining a *reference model[19]* and an *adaptive algorithm*. This separation of concerns makes it ideal for self-adaptive systems.

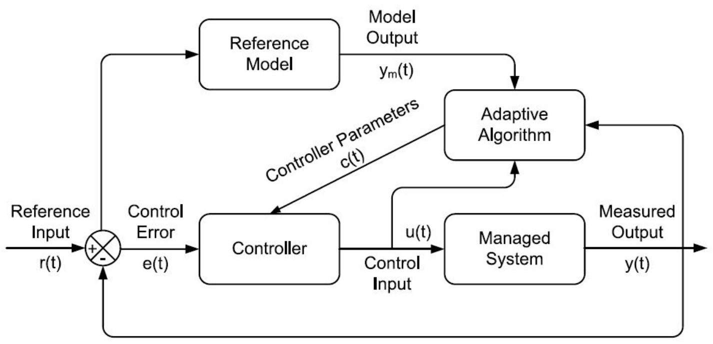

Fig. 3. Basic structure of model reference adaptive control (MRAC)[20]

5 Approaches

5.1 Autonomic computing reference architecture (ACRA)

IBM proposed an architectural blueprint for autonomic computing. According to this proposal, the management of a system involves four common activities: *collect information, analyze it to determine actions that need to be taken, create a plan with those actions*, and finally, *execute the plan.* In light of this, for the system to manage itself this process, the following conditions must exist[8]:

– The management tasks known as: configuring, healing, optimizing and protecting, must be automated.

[2] MRAC: Model-Reference Adaptive Control[18].

- These processes must be possibly initialized based on situations observed in the context.
- The system must possess enough knowledge to take the automated tasks.

The IBM's proposal presents a five layer scheme, shown on Fig. 4, to control self-management systems. However, the most relevant aspects in IBM's proposal for my project are in layers 2 and 3[8]:

- *Level 2 - Touchpoints* These are the components responsible to interact with the managed resources. The interaction refers to be aware of the state and management of the resource trough two components: sensor and effector. The sensor is in charge of knowing the state of the resource either by acknowledging the properties and looking for changes, or by the events triggered over the changes in the resource. Meanwhile, the effector is in charge to notify changes over the resource either by the properties or by operations that are implemented in the managed element.
- *Level 3 - Touchpoint autonomic managers* These components implement the *intelligent control loop* over the managed resources, arranged typically in four scopes: single resource, homogeneous, heterogeneous and business system.
  This control loop is also known as the MAPE-loop: Monitor, Analyzer, Planner, and Executer (See Fig. 5). These four parts communicate and collaborate to each other and exchange appropriate knowledge and information. The Monitor collects the information from the managed resources. The Analyzer observes the reported situation, and determines if any change needs to be made. The Planner creates the plan to achieve the changes, and the Executer provides the mechanism to perform the changes over the managed resource.
  The common part of the MAPE loop, is the knowledge source, which is shared data among the autonomic managers in the system. In order to do that, the knowledge must by expressed using common syntax and semantic to the system. There are three ways to specify knowledge: through policies, activities inside the autonomic system or the actions inside the autonomic control loop.

5.2 Garlan's Rainbow Framework

The rainbow framework represents a control loop for self-adaptation (See Fig.6. This framework uses and abstract model to monitor the system properties during runtime and evaluate if any constraint violation is being made, such case, the system performs self-adaptation to reestablish the property as is desired. This framework supports runtime adaptation by capturing dynamically the system's attributes[10].

The Rainbow's control loop is divided in two layers where the self-adaptive elements can be found described as follows:

- In the *System Layer* are the target system, the sensors and effectors. This layer provides the system to be managed and the connector needed to sense the state of the system and the effector to perform the adaptation changes.
- In the *Architecture Layer* are the control loop elements. The Model Manager (Monitor) monitors the sensed information according to a set of types and properties. Next, The Constraint Evaluator (Analyzer) validates the monitored information according to a set of rules, and decides whether to adapt. If adaptation is needed, the Adaptation Engine (Planner) uses strategies and tactics to plan a set of adaptation actions. Finally, The Adaptation Executor (Executor) uses operators to apply the adaptation plan over the system.

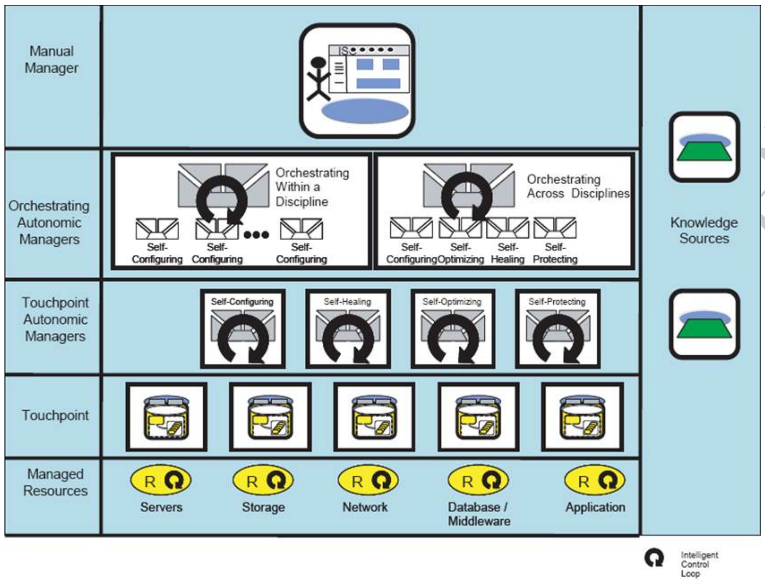

Fig. 4. Autonomic computing reference architecture (ACRA) [8]

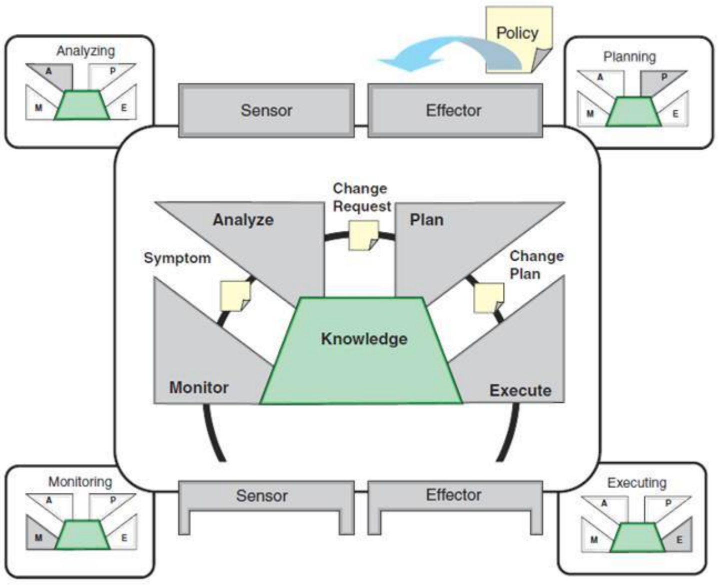

Fig. 5. Functional details of the Autonomic manager (ACRA) [8]

5.3 Parra's Architecture

The Parra's Architecture is focused to context-aware systems, so the adaptation need appears as a response of changes in the context in order to dynamically integrate the assets in a running system[11]. As shown in Fig.(7), Parra proposed a Component-based platform architecture. In his proposal, the self-adaptive elements can be identified as follows:

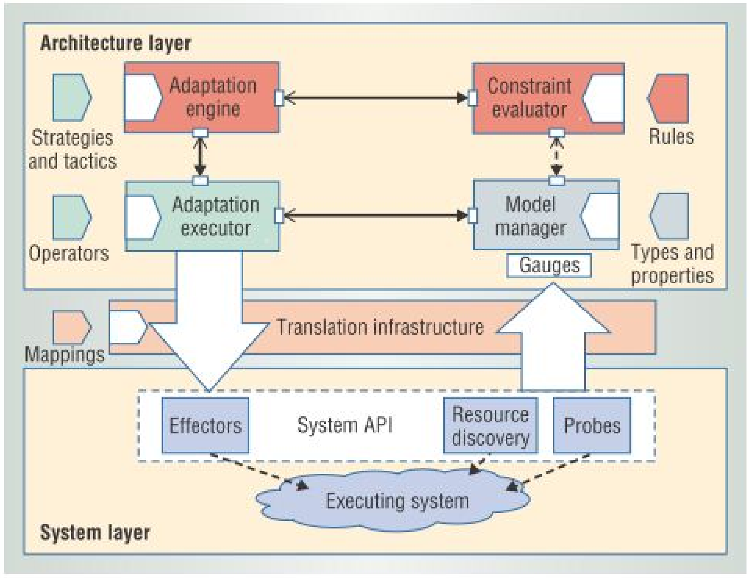

Fig. 6. Garlan's Rainbow Framework [10]

- The Monitor in his proposal is composed by two elements: Three nodes (Sensors) connected to internal context (Runtime platform), external context (Sensor Layer) and the policies given by administrators (User Preferences). These nodes report the sensed information to the second element: the Context Manager. This element process the data and presents as a single context information.
- The Analyzer, Planner and Executor are represented by the Decision Maker, this component evaluates the context, and decides whether is necessary to adapt or not using the Rules repository. This component has an interface to perform the reconfiguration operations over the system (Effectors).

5.4 Solomon's autonomic system

Solomon's proposal presents an adaptive control loop architecture (See Fig8) separated in three layers[12]. In his proposal, Solomon includes two types of feedback loops. The first feedback loop is to adapt the system and the second loop is to manage the adaptation mechanism itself. Also, these layers are made of some other components that represent the elements in self-adaptation as follows:

- The adaptation Layer is responsible for modifying the control loop in change of adaptation. For this loop, The Autonomic System Model (Monitor) monitors the autonomic system adaptation results. Next, the Constraint Evaluator (Analyzer) evaluates if the adaptation rules are valid bounds according the system behavior. The Adaptation Controller (Planner) receives notification if those rules are not valid and chooses the actions to be taken in order to guarantee the control loop's adaptation behavior. Finally, the Adaptation Executor (Executor) changes the filters in the autonomic system to apply the correct adaptation.
- The Autonomic System Layer is responsible to effect the adaptation over the system. In this layer, the Sensor (Monitor) examinees the the System according the upper layer information

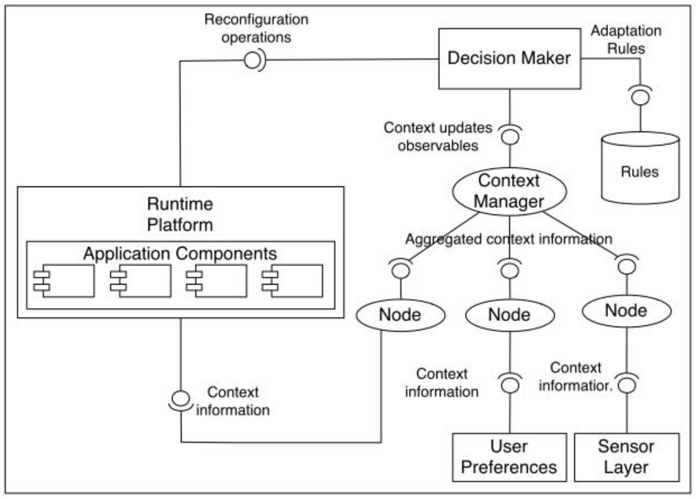

Fig. 7. Parra's platform architecture [11]

needs. The Filters (Analyzer) component analyzes the previous monitored information and decides whether is necessary to adapt or not. Next, the Controller (Planner) is in charge of create the adaptation plan over the system and passes it to the Actuator (Executor) who is responsible to perform the changes in the system.

In Solomon's proposal, the sensors and effectors are not explicit but the connections between the respective target systems.

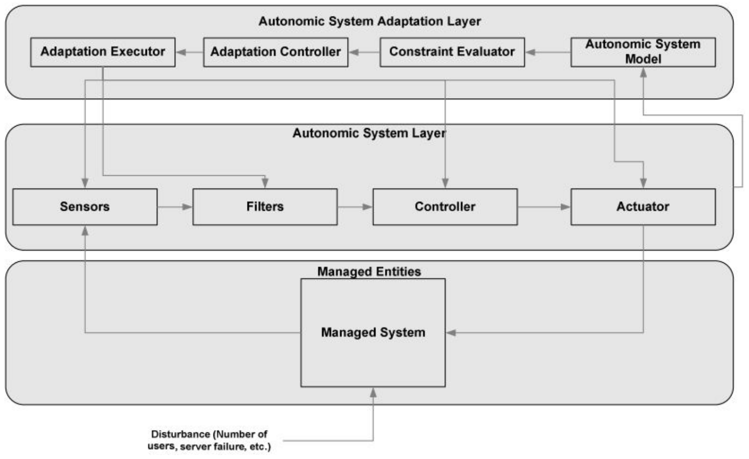

Fig. 8. Solomon's adaptive autonomic system [12]

6 Reference architectures

6.1 Muller et al.'s reference model

Undoubtedly, in order to implement adaptation through feedback loops, is necessary to make explicit the management of self-adaptive properties as the control reference goals, and the separation of concerns by using different feedback loops to achieve the reference goals[13].

The reference model for context-based self-adaptation by Villegas et al[13] shown in Fig.9 is based on the SISO feedback control (Fig. 2) with explicit functional elements and corresponding interactions to control dynamic adaptation. These are the MAPE[3] loop elements.

The separation of concerns is a key aspect in this model. In order for a system to become self-adaptive are needed three subsystems[13]:

1. *Control objective manager:* Manages the target system purpose in terms of its control objectives, according to the policies given by administrators. These control objectives can change at runtime according to user-level negotiations. However, this management implies to express quality attributes quantitatively and have them updated at runtime.
2. *Context manager:* Responsible for maintaining the pertinence and relevance of the target system under changing conditions of execution. In order to do that, the context manager must be able to decide about the context facts, past, present and future states.
3. *Adaptation mechanism:* Responsible for the adaptive actions over the target system according to the evaluation of its behavior. That means, to guarantee the accomplishment of the target system purpose to satisfy quality attributes.

In this reference model is important the interaction between the three loops. Even though they are design of it independently they require to operate together to achieve the system objectives.

6.2 Kramer and Magee's layers architecture

Kramer and Magee proposed a three level architectural model for self-managed systems [9]. The three layers are: *Component control, Change Management* and *Goal Management* (See Fig. 10).

– *Component control layer*
  This layer is responsible for the interconnection among components to accomplish the function of the system. It facilitates the information about the status of the components. Furthermore, in this layer is where occurs the deployment, undeployment and interconnection of the system components.
– *Change Management*
  The second layer in this model is responsible for the execution plans, that is, the actions to handle new unexpected situations. The actions taken in this layer might include the introduction of new components, changes in the interconnection or even in the operational parameters.
– *Goal Management*
  At the highest level, the plans needed from the layer below are produced. Also, in this layer the introduction of new goals take place. These new goals are being introduced in the system a consequence of the changing environment that could require a different behavior of the system to achieve its goals or even changing its goals.

[3] MAPE: Monitor, Analyzer, Planner and Executor[8]

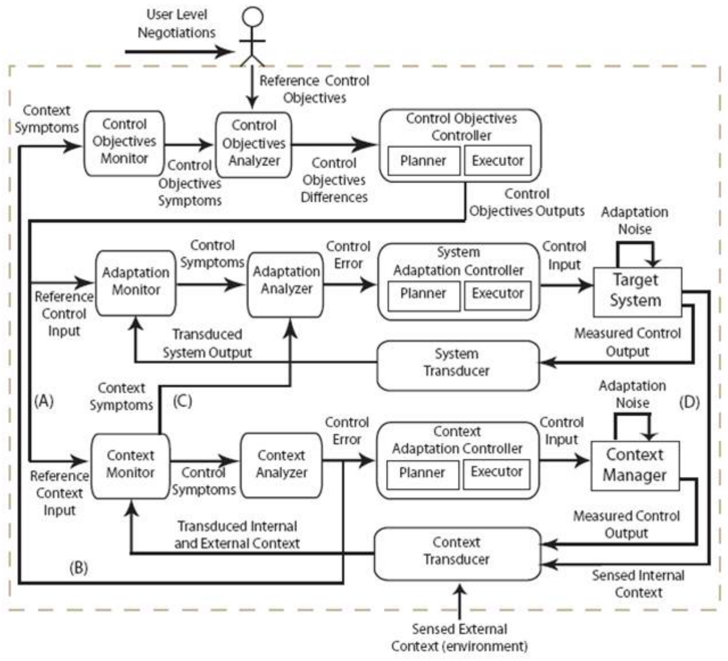

Fig. 9. Reference model for context-based self-adaptive systems [13]

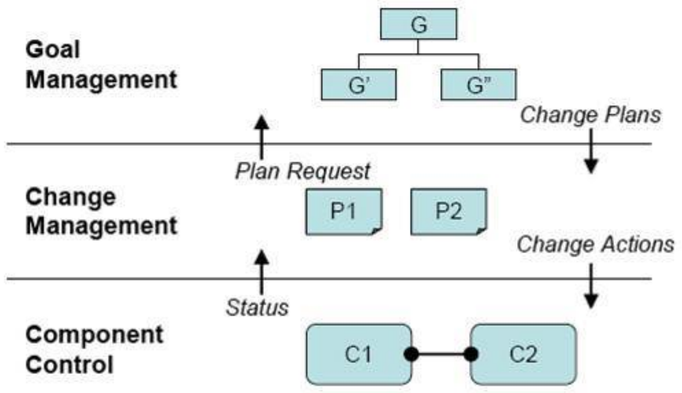

Fig. 10. Three Layer Architecture Model for Self-management [9]

6.3 A reference architecture for adaptive SOA governance

Villegas proposes an architecture where governance feedback loops co-operate with each other to monitor the dynamic relevant context to keep pertinence between SOA governance[4] objectives and context information to be monitored. SOA governance implies the appropriate management of the SOA infrastructure to accomplish the business objectives. Thus, the SOA governance goals

[4] SOA governance: defines the policies and enforcement mechanisms for implementing, executing and evolving service-oriented systems[14]

are to control, monitor and adapt the components in that infrastructure by keeping track of the environmental changes[14].

As depicted in Fig.11, two domains are required to monitor context in SOA environments that are interconnected:

– **SOA Governance Domain** defines the elements to regulate the accomplishment of the SOA governance objectives. The *SOA Governance Monitor* receives from the context monitor domain the context symptoms and with these identifies governance symptoms to make decisions about the necessity of requesting a governance plan. The *SOA Governance Controller* receives the request to define a plan according former policies, depending on whether it is required to adapt the context monitoring infrastructure.
– **Context Monitor Domain** defines the elements required to gather the relevant context information that can affect the SOA governance objectives accomplishment. The *Sensing Infrastructure* defines the sensors and endpoints required to acquire raw context information from the environment. This information is monitored by the *Context Monitor* that based of the Context Model Controller filters the relevant context information to process and delivers context symptoms to the SOA Governance Domain

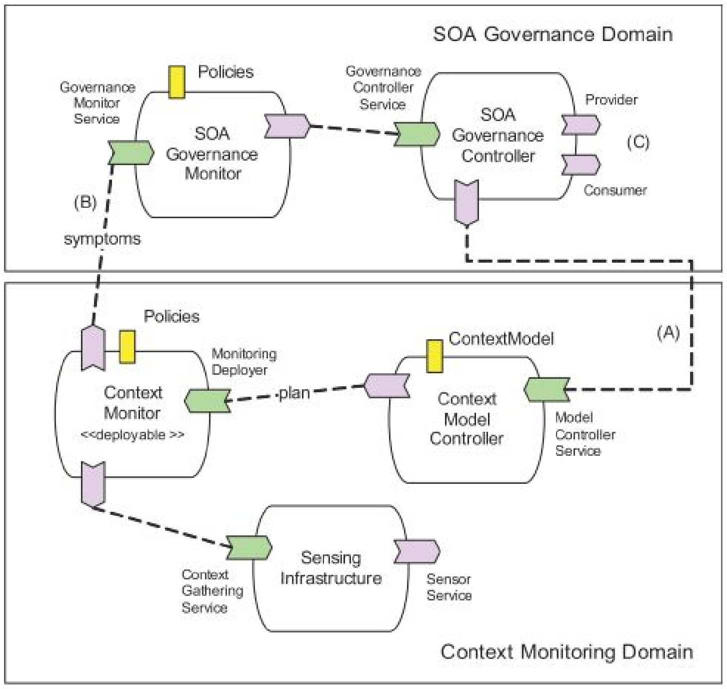

Fig. 11. Villegas's Control-based service component reference architecture for monitoring context in SOA governance[14]

7 Conclusions

Undeniably organizations no longer are stable, on the contrary, are in a constant change because of the dynamic behavior of context, thus, software engineers must design software able to fit those organizations through its evolves and adaptation without badly interfere. Therefore software production industry must design ULSS adaptable to highly complex, unpredictable and uncertain environment. Most importantly, this adaptation must be efficient in effort and human intervention.

ULSS are composed of people, technology and data; so it is inadequate to build ULSS without considering all these aspects and the interactions among them. Therefore, software engineers must interact with people from all the other disciplines related to every aspect in the engineering of ULSS. The ULSS challenges reflect the urgent necessity of innovative and applicable research initiatives related to the instrumentation of ULSS with context-aware self-adaptive capabilities, without sacrificing their quality and requirements

As it has been noted during this survey, there is a need for systems to self-manage in order to free IT professionals from routine maintenance tasks over today's high complex systems. Autonomic systems are those that can self-configure, self-heal, self-optimize and self-protect according to policies given by administrators. This type of systems are required to self-adapt at runtime in response to changes in the context, and the adaptation is required most of the times of runtime.

As new business objectives are required, the system must be able to meet those objectives generally by adapting some part of it-self. This self-adaptation requires to perform four tasks: **monitoring** the new business requirements, as the internal and the external context, **analyzing** the data gathered to identify symptoms and need for adaptation, **planning** a set of adaptive actions, and **executing** those actions over the system.

One of the most important challenges in self-adaptation is to create the ability in the system to reason about itself and its context. Control engineering uses feedback to measure and act upon the behavior of the system. Moreover, *feedback loops* become important in the process that the system must follow to read an output property of the system and then evaluate if the desired properties are being reached [7]. Software engineers have made significant efforts to propose self-adaptive systems. The proposals reviewed in this survey have used different approaches based on feedback loops, components and even hybrid architectures, and a set of standards to communicate these systems. However, all these approaches are not sufficient for the different systems, data types, and configurations that are available. Often, this proposals merge self-adaptation with the system making very difficult to analyze the adaptation capability as an independent system, and feedback loops are not explicit in most of these proposals.

References

1. Truex, D., Baskerville, R., Klein, H.: Growing systems in emergent organizations. Communications of the ACM **42 No. 8** (1999)
2. Northrop, L., Feiler, P., Gabriel, R., Goodenough, J., Linger, R., Longstaff, T., Kazman, R., Klein, M., Schmidt, D., Sullivan, K., Wallnau, K.: Ultra-Large-Scale Systems. The Software Challenge of the Future. Carnegie Mellon University (2006)
3. Kephart, J., Chess, D.: The vision of autonomic computing. IEEE Computer **36 No.1** (2003) pp. 41 – 50
4. Ganek, A., Corbi, T.: The dawning of the autonomic computing era. IBM Systems journal **42 No.1** (2003) pp. 5–18
5. McKinley, P.K., Masoud Sadjadi, S., Cheng, B.H.: Composing adaptive software. IEEE Computer (2004)
6. Cheng, B.H., Lemos, R., Giese, H., Inverardi, P., Magee, J., Andersson, J., Becker, B., Bencomo, N., Brun, Y., Cukic, B., Serugendo, G.M., Dustdar, S., Finkelstein, A., Gacek, C., Geihs, K., Grassi, V., Karsai, G., Kienle, H.M., Kramer, J., Litoiu, M., Malek, S., Mirandola, R., Mller, H.A., Park, S., Shaw, M., Tichy, M., Tivoli, M., Weyns, D., , Whittle, J.: Software engineering for self-adaptive systems: A research roadmap. Springer-Verlag **LNCS 5525** (2009) pp. 1 – 26
7. Hellerstein, J.L., Diao, Y., Parekh, S., , Tilbury, D.M.: Feedback Control of Computing Systems. John Wiley and Sons (2004)
8. IBM: An architectural blueprint for autonomic computing. Autonomic computing (2005)
9. Kramer, J., Magee, J.: Self-managed systems: an architectural challenge. IEEE Computer (2007) pp. 259268
10. Garlan, D., Chen, S.W., Huang, A.C., Schmerl, B., Steekiste, P.: Rainbow: Architecture-based self-adaptation with reusable infraestructure. IEEE Computer (2004)
11. Parra, C., Blanc, X., Duchien, L.: Context awareness for dynamic service-oriented product lines. Proc. of 13th Intl. Software Product Line Conference (2009) pp. 131140
12. Solomon, B., Ionescu, D., Litoiu, M., Mihaescu, M.: A real-time adaptive control of autonomic computing environments. CASCON (2007) pp. 124136
13. N.M., V., H.A, M., G., T., L., D., R, C.: A control engineered reference model for context-based self-adaptation. To appear **xx** (2010)
14. Villegas, N., Müller, H.: Context-driven adaptive monitoring for supporting soa governance. Proceedings of the 4th International Workshop on a Research Agenda for Maintenance and Evolution of Service-Oriented Systems (MESOA 2010) **Carnegie Mellon University Software Engineering Institute** (2010. To appear)
15. Tewari, V., Milenkovic, M.: Standards for autonomic computing. Intel technology journal **10 No.4** (2006) pp. 275–284
16. Dawson, R., Desmarais, R., Kienle, H., Müller, H.: Monitoring in adaptive systems using reflection. ACM/IEEE **SEAMS 2008** (2008) pp. 81–88
17. Müller, H., Pezz, M., , Shaw, M.: Visibility of control in adaptive systems. Proceedings of the 2nd international workshop on Ultra-large-scale software-intensive systems **ULSSIS 2008** (2008) pp. 23–26
18. Giese, H., Brun, Y., Serugendo, J.D.M., Gacek, C., Kienle, H., Mller, H., Pezz'e, M., , Shaw, M.: Engineering self-adaptive systems through feedback loops. Springer-Verlag **LNCS 5525** (2009) pp. 47 – 69
19. Bass, L., Clements, P., Kazman, R.: Software Architecture In Practice. Addison-Wesley (2003)
20. Müller, H..A., Kienle, H.M., , Stege, U.: Autonomic computing: Now you see it, now you dontdesign and evolution of autonomic software systems. Lecture Notes in Computer Science **5413** (2009) pp. 32–54