ISSN(Online):2320-9801
ISSN(Print): 2320- 9798

International Journal of Innovative Research in Computer and Communication Engineering

(An ISO 3297: 2007 Certified Organization)

Vol. 2, Issue 4, April 2014

Self-organization and Self-healing: Rationale and Strategies for Designing and Developing a Dependable Software System

Okon S. C.[1] **and Asagba P. O.**[2]

Lecturer, Department of Computer Science, Akwa Ibom State University, Ikot Akpaden, Mkpat Enin L. G. A. Akwa Ibom State, Nigeria[1]

Senior Lecturer, Department of Computer Science, University of Port Harcourt, Choba, Rivers State, Nigeria[2]

ABSTRACT: The focus of this work is to overview a set of fundamental principles that underlie software system dependability and therefore suggest a different approach to the development, assessment and implementation of dependable software. Due to a lack of sufficient data to support or contradict any particular approach, a software system may not be declared "dependable" based on the method by which it was constructed. Rather, it should be regarded as dependable-certifiably, dependable-only when adequate evidence has been gathered and tested in support of an argument for dependability that can be independently assessed. The goal of certifiably dependable software cannot therefore be achieved by mandating particular processes and approaches, regardless of their effectiveness in certain situations. Instead, software developers should marshal evidence to justify an explicit dependability claim that makes clear which properties in the real world the system is intended to establish. Such evidence forms a dependability case as found in nature via self-organization and self-healing in natural artifacts. Consequentially, our creating and designing of a self-organized and a self-healing software system whose dependability case is the cornerstone of the approach to developing certifiably dependable software systems (DSS) of our interest.

KEYWORDS: Dependable Software, Self-organization, Self-healing, Configuration, Monitor.

I. INTRODUCTION

A system is dependable when it can be depended on to produce the consequences for which it were designed and no adverse effects, while operating inside its intended environment. This means, that the term dependability has no useful meaning for a given system until these consequences and the intended environment are made explicit by a clear prioritization of the requirements of the system and an articulation of environmental assumptions. The effects of software are felt in the physical, human, and organizational environment in which it operates, so dependability should be understood in that context and cannot be reduced easily to local properties, such as resilience to crashing or conformance to a protocol. Humans who interact with the software should be viewed not as external and beyond the boundary of the software engineer's concerns but as an integral part of the system. Failures involving human operators should not automatically be assumed to be the result of errors of usage; rather, the role of design flaws should be considered as well as the role of the human operator. As a consequence, a systems engineering approach - which views the software as one engineered artifact in a larger system of many software components, some engineered and some given, and the pursuit of dependability as a balancing of costs and benefits and a prioritization of risks is vital. Thus, the field of software engineering suffers from a pervasive lack of concrete evidence about the incidence and severity of software failures; about the dependability of existing software systems; about the efficacy of existing and proposed development methods; about the benefits of certification schemes; and so on. We are hoping and working towards a breakthrough.

II. DEFINITION OF TERMS

Configurator: A software module that re-assemble the flow of control/module traversals in a dependable software system (DSS) in other to repair or overcome a given problem.
Monitor: A software alert system (SAS) or driver that raise exceptions (issues of concern) that will be fed into the DSS to trigger self-organization or self-healing strategies thus ensuring dependability.
Self-organization: A spontaneous process where some form of global order (software goal) or coordination arises out of the local interactions between the software modules (a set of, some of or all the modules) of a software system to automatically evolve towards a state of equilibrium and or attraction.
Self-Healing: The ability to perceive that a system is not operating correctly and without human intervention, the system makes the necessary adjustments to restore itself to normal operation.
Software Dependability: Trustworthiness of a software system such that reliance can justifiable be placed on the service it delivers

III. RELATED WORKS

Self-Organization and Self-healing have been researched into in many disciplines, like Biological Science, Chemistry, Physics and Hardware systems. However, research is ongoing on Self-organization and healing in Software Engineering. We looked at self-organization and self-healing in some of these areas mentioned above and brings the concept into practice in Software Engineering. In [1] authors worked on protein folding, where the shape and structure of protein is self-organized into a well defined three-dimensional structure was reviewed. Our review also highlights how the human body self-organized in face of external conditions to maintain stability [5]. The beautiful pattern formed during colony growth based on self-organization was also reviewed and applied [19].

Natural complex systems are highly sophisticated yet they organized themselves to adapt to situations. They are incredibly robust, flexible and adaptive, tackling problems far more complex than any computer system [16]. Self organizing systems are intrinsically robust; they can withstand a variety of errors, perturbations, or even partial destruction. They will repair or correct most error themselves, getting back to their initial state. When the damage becomes too great, their function will start to deteriorate, but “gracefully”, without sudden breakdown. They will adapt their organization to any changes in the environment, learning new “tricks” to cope with unforeseen problems. Out of chaos they will generate order.

The concept of self organization as applied in this research work is an abstraction of that of biological systems “*Homeostasis*”. The field of self-organization seeks general rules about the growth and evolution of systematic structure, the forms it might take, and finally methods that predict the future organization that will result from changes made to the underlying components, [24]. According to [7] the future of robust and dependable autonomous communication networks depends on self-organization. [20], opined that natural intelligence draw its powers and sophistication from self-organization, as a corollary to this, authors in [25, 26, 27] have shown that software system’s intelligence can be improved by applying self-organization and self-healing.

In Hardware Engineering, Self-organization is an evolutionary trend from Fault Tolerant, adaptive and autonomous computing. Self-organization abounds in systems of networks with massively distributed hardware with self-organized control. In [8] the authors used ant- inspired technique to design a storage area network. According to [25] the author likens microprocessor devices to human and concluded that both are self-organizing system. [27] uses self-organization principles to proffer solutions to mobile network problems. [3], presented an autonomous distributed self-organizing and self-healing electronic DNA (eDNA) hardware architecture, which is capable of responding to multiple injected faults by autonomously reconfiguring itself to accommodate the fault and keep the application running. In [10], the authors reveal that there exists a relationship between self-organization in hardware devices and principles of pattern formation in natural biological systems.
The study of Self-organization in natural complex systems and hardware systems has given us enough to design and develop dependable software via self-organized/healing level of abstraction and that is exactly the intended purpose of this research work.

ISSN(Online):2320-9801
ISSN(Print): 2320- 9798

International Journal of Innovative Research in Computer and Communication Engineering

(An ISO 3297: 2007 Certified Organization)

Vol. 2, Issue 4, April 2014

IV. STRATEGIES FOR DEVELOPING DEPENDABLE SOFTWARE.

Dependability has several aspects:

- With respect to the readiness for usage, dependability means availability;
- With respect to the continuity of service, dependability means reliability;
- With respect to the avoidance of catastrophic consequences on self and environment, dependability means safety;
- With respect to the prevention of unauthorized access, dependability means security.

Our strategies revolves around the fact that self-organization naturally depends on employing self-similar entities in an architecture. The self-similar nature of the architectural component could be in structure or in function or both e.g. neurons. We hereby deploy these in designing software architectures based on functional and/or structural self-similarities to realized dependability [26]. We also exploit the fact that each element or component in our proposed software architecture could replace at least one other component in case of any failure, thus reconfiguring the entire system state to a safe and dependable state.

V. SELF-ORGANIZING CHARACTERISTICS

- Self-organization essentially relates to the capacity of the system to spontaneously produce new organization/structures in case of environmental changes or fluctuations.
- Software components dynamically bind together under user requests or for pursuing their own goals; software components dynamically adapt or conform to non-functional requirements, and policies of different nature (execution flows, resources description and access constraints).

In other words, the system taken as a whole re-organizes itself as the result of software components' own actions (without human or external assistance) under environmental changes: user requests, other software components availability, changing policies, resources needs, etc.

Four necessary requirements have been identified for systems exhibiting a self-organizing behaviour in thermodynamics environments [17]. In order to give a more precise description, we will identify for each point the corresponding behaviour in the software architecture model.

Mutual Causality: "At least two components of the system have a circular relationship, each influencing the other" [9]. One of the aims of the architecture is to support semantic interaction of software components that may not know each other in advance. This interaction necessarily implies a mutual causality in the sense that: on the one hand the service requester is satisfied or not by the service provider, thus affecting the requester's subsequent behaviour; and on the other hand the service provider has been activated under the service request. This may affect the provider in several ways: need to make some computation (resource consumption), possible modifications of local state due to the service request, etc. In our software architecture, to achieve mutual causality, the state at run time depends on input data and its nature [26]. The software modules also depend on services of each other.

Autocatalysis: "At least one of the components is causally influenced by another component, resulting in its own increase" [9].

Interacting software components may have an autocatalytic effect on the system, in the sense that a well satisfied request will cause the corresponding service to be solicited more than another service which may be less available or efficient. On the contrary, too much solicitation may lead to a degraded service satisfaction, and through dynamic regulation, according to specified reconfiguration schemas, more performing/efficient components (software modules) will be selected to satisfy the request. In our proposed system, each selection/reconfiguration is dependent on choosing a correspondingly efficient module with respect to the systems need (internal input/signal and/or external input/signal) to achieve dependability [26].

Far-from equilibrium condition: "The system imports a large amount of energy from outside the system, uses the energy to help renew its own structures (autopoeisis), and dissipates rather than accumulates, the accruing disorder (entropy) back into the environment" [9]. Our proposed software system is capable of learning from external input signals which is the environmental impact/constraints on the software system [26].

Our proposed software system must be part of a highly changing environment. They need power supplies, network links, memory, etc. Software Components continually join and leave the system (inserted or removed by the environment,

or no longer available because of lack of resources). The system then permanently and seamlessly integrates new or updated software components, as well as environmental elements, such as additional CPU, power supply, etc.
Malfunctioning or non-responding software components see their accesses or interactions denied from other software components or from the system. In that sense, they leave the set of interacting software components, maybe temporarily, and are then part of the environment, since they still consume some resources (entropy is pushed back into environment).
Morphogenetic changes: "At least one of the software components of the system must be open to external random variations from outside the system. A system exhibits morphogenetic change when the components of the system are themselves changed" [9].

Run-time evolution of software components contributes to the continuous morphogenetic changes of interacting entities/modules: software components leave and join (appear/disappear); entities/modules are upgraded (undertake software evolution).

A given system may last for several years (old and recent portions of it working together) in a manner akin to the regeneration of animal cells.
In addition to changes in the software components themselves, their executing environment may change: computing, networking, or storing resources may be removed, added, or changed.

VI. POTENTIAL OF SELF-ORGANIZING SOFTWARE ARCHITECTURES FOR SELF-DEPENDABLE SYSTEMS

This section shows the interest of the above described architectural model for self depending systems. More precisely, it describes the potential of such an approach for realizing the four self-depending concepts defining the autonomic computing view as explained in [22]. Essentially this potential relies on: dependability metadata for expressing functional behaviour of software components, as well as their non-functional requirements; reconfiguration schemas for expressing different kinds of policies; and on the underlying run-time infrastructure (component management environment) supporting seamless interactions of software components, dynamic binding, etc.
Self-Configuration: "Automated configuration of software components and systems follows high-level policies. Rest of system adjusts automatically and seamlessly", [22].

- The notion of reconfiguration schema naturally serves to express high-level goals as could be done by a (human) administrator. They constitute initial service requests triggering the rest of the system. At a lower level, service requests expressing high-level configuration policies may serve for automatic distribution of entities (e.g. placement on a Grid) of software components participating in some computation. This leads to new reconfiguration schemas inserted into the system. Finally, at a local level, each component of an autonomic computing system may describe, through its non-functional dependability metadata, its own installation needs (e.g. CPU, Memory, Data files or Network).
- High-level policies acting as high-level requests (goals) from human administrator (system installation needs) are expressed through reconfiguration schemas; High-level requests for configuration policies (Grid distribution) are expressed as a service requests in the form of reconfiguration schemas;
- Local-level configuration policies of individual software components, express individual software components' installation needs (CPU, memory, etc.). They are described through non-functional metadata.

Self-Optimization: "Software components and systems continually seek opportunities to improve their own performance and efficiency", [22].

The combined use of non-functional dependability metadata related to optimization needs, and recon-figuration schemas for expressing optimization policies serves self-optimization purposes. Parameters to optimize may be described through metadata, and optimization of parameters depending on the context may be expressed through reconfiguration schemas.

The underlying run-time infrastructure is then responsible for executing software components' requests, while actually satisfying their requested optimization parameters descriptions and optimization policies. Since, at each service request, the underlying run-time infrastructure checks optimization parameters and policies, the system then permanently improves its own performance given the current environmental context.

ISSN(Online):2320-9801
ISSN(Print): 2320- 9798

International Journal of Innovative Research in Computer and Communication Engineering

(An ISO 3297: 2007 Certified Organization)

Vol. 2, Issue 4, April 2014

For example, it uses updated software components or more efficient available software components for satisfying the ongoing requests; or it takes into account current environmental resources for providing efficient software components executions. In summary:

- Non-functional metadata describe optimization parameters;
- Reconfiguration schema describe optimization of parameters depending on the context;
- Run-time infrastructure ensures the use of optimized service for satisfying the service requests.

Self-Healing: Our proposed dependable software system automatically detects, diagnoses, and repairs localized software and hardware problems [22, 26, 27].

Generally speaking, to make a system self-healing we need to apply appropriate fault tolerance techniques [23, 27]. The choice here depends on the requirements, the resources available and the fault assumptions.
Making evolvable and open systems self-healing requires advanced fault tolerance mechanisms which employ redundant resources in a dynamic and adaptable fashion. This typically involves dynamic system reconfiguration with deployment of new software components.

The architectural model introduced in Section VIII for supporting dynamically resilient software systems is a specific approach which enables building of self-healing systems capable of tolerating a wide range of faults, damaging events and changes in the systems, infrastructures and system environments. A dedicated architecture allows choices about the use of the specific fault tolerance mechanism and of the redundant resources to be made dynamically using metadata/driver available in run-time.

Moreover, functional metadata/driver serve for both detecting erroneous code (checking of code against functional specification), and for potentially replacing such erroneous code by an equivalent one (automatic generation of code). Indeed, from the one hand, the underlying run-time infrastructure may verify the adequacy of a code from its corresponding functional metadata (e.g. through proof-carrying code techniques).

If a software component is recognized as erroneous by the run-time infrastructure, its functional and non-functional metadata is removed from the repository of available modules/services (that module/service is no longer available at that instance). The whole system then automatically works with the remaining available modules/services, maybe in a gracefully degraded manner, until a new code is inserted into the system and replaces the erroneous one. Alternatively, the explained/envisaged architecture can be extended to incorporate automatic generation of a code (recognized as erroneous) from its corresponding functional metadata description. Consequentially;

- The architectural model contains built-in support for dealing with a wide range of faults;
- The model can be extended for: 1. checking code/module against its functional metadata/driver, thus detecting potential erroneous code; 2. using functional metadata/driver as a basis for replacing erroneous code with another code having an equivalent functional metadata/driver; 3. automatically generating correct code for a replacement module from functional metadata information of an erroneous code.

Self-Protection: "Our proposed System must automatically defend itself against malicious attacks or cascading failures. It uses early warnings, to anticipate and prevent system-wide failures", [22, 27].

The proposed architectural model arises from fault-tolerance, resilient and dependability concerns. Then, it inherently supports system resilience despite software component or environmental failures, essentially through the notions of dependability metadata such as conditions regulating services delivery, and recon-figuration schemas, which naturally describe high-level security policies that have to be realized in the whole system. Regarding malicious attacks, reconfiguration schemas may serve for expressing signatures of security attacks and response schemas to attacks, thus allowing the software system to recognize and react to run-time attacks.

Additionally, self-regulating schema for failures and attacks can also be considered based on non-functional metadata. For instance, combining trust and reputation information within non-functional metadata may prove to be an efficient tool for self-protection [11]. Thus;

- The architectural model contains built-in support for system failures through fault-tolerance related issues;
- Malicious attacks can be addressed through the use of reconfiguration schemas;
- Additional self-protection schemas, such as those based on trust and reputation can be incorporated through adequate descriptions in non-functional metadata and reconfiguration schemas.

VII. ADAPTATION AND DEPENDABILITY

Software-based systems are designed and implemented to operate under a given set of environmental and operational circumstances.

Our software system design is based on assumptions about the state and various forms of state manipulations and possible changes of the system, the platforms on which it will execute, and the possible form of communication network between modules and environment over which it will communicate. These assumptions may be either explicit (highly desirable), or implicit (undesirable, but common). Even well-designed systems tested against an explicit set of expected conditions often experience faults or failures when unforeseen circumstances violate one or more expectations. While robust architectures and good system design practices have always led to systems that respond relatively gracefully to system or platform faults, even if that means little more than to say that well-designed systems fail relatively gracefully, for most applications and platforms, highly dependable systems must also be self-adaptive–that is, they must incorporate functionality that enables them to reconfigure themselves in response to unforeseen conditions – to actually achieve high levels of dependability.

The increase in dependability from self-healing is proportional to the probability that a given explicit or implicit assumption will be violated, and the probability of a system failure given such a violation, and the probability that self-healing will prevent the failure, divided by the overall probability of system failure for any reason.

It is obvious that the potential for adaptive self-healing systems can enhance dependability where any of the assumptions underlying the system implementation may be violated. And unless the system is being built on a completely deterministic execution platform, using machine language or a real-time operating system, and all possible combinations of data input and system state are able to be generated and tested before the system is deployed, which thou possible with small scale software system always becomes a complex and complicated task with large scale software systems. Self-healing is a critical strategy for enhancing system dependability for any applications where the cost of system failure would be prohibitive (mission critical systems).

VIII. ADAPTIVE SYSTEM REQUIREMENTS

Since self-adaptive systems must detect runtime conditions for which some kind of adaptation should be done, and then perform some kind of adaptive configuration in response, it follows that adaptive system architectures have three basic requirements: a *reflection* mechanism to detect internal or external conditions to which the system should respond; a *reasoning* mechanism to determine what actions should be performed in response to input from the reflection mechanism; and a *configuration* mechanism to perform the necessary changes to repair or optimize the system as directed by the reasoning mechanism. Adaptation implementations can range from simple, ad-hoc solutions consisting of hard-coded inline program statements that check for and respond to specific runtime conditions, to comprehensive hierarchical agent-based monitor-configurator frameworks. *Figure 1* shows a conceptual model for self-healing software architectures.

The adaptation mechanism consists of REF-module that performs *reflection*, RES-module that performs *reasoning* and CON-module that perform state/modules/software components *configuration*. Each adaptation component either monitors or configures one or more aspects of the dependable system or its environment.

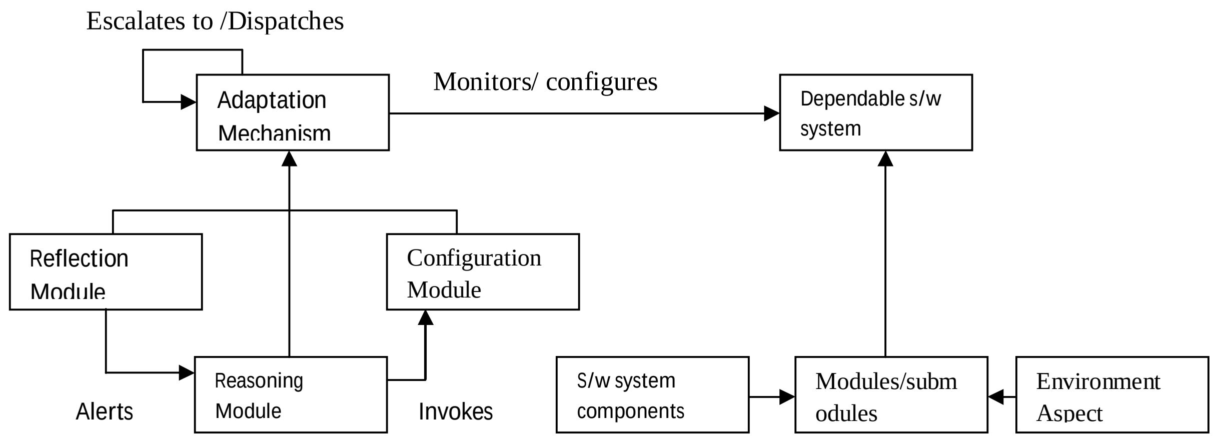

Fig. 1: Conceptual model for self-healing software architectures

IX. DESIGN ISSUES

There exist various aspects of self-healing software architectural design, this section focuses on four major design issues related to the organization and communication among the major adaptive modules in the architecture (reflection module, reasoning module and configuration module). Architectural aspects explored include the degree of inter-module coupling, the direction and level of inter-module communications, configuration dispatch styles, and ways to specify configuration functionality. In this paper, runtime reflection software modules are called *monitors/drivers* which coordinates the drivers which are sub-class of monitors, and configuration software modules are called module *configurators* which coordinate the software modules as desired by a particular request. Configurators may also encapsulate reasoning functionality; separate system-level reasoning software modules are called *configuration managers*.

Tight vs. Loose Coupling

The first major architectural design issue for adaptive systems is the level of inter-component coupling. In a tightly coupled adaptation system, the reflection, reasoning and configuration software components have direct, explicit interdependencies. Examples of tight coupling include systems where the monitoring, configuration and configuration management implementation entities (software components, objects, or methods) explicitly invoke one another without an intervening layer of logical abstraction.

Tightly coupled systems may be somewhat simpler to implement, but this apparent simplicity comes at the cost of flexibility and scalability. Tight coupling is a particularly risky architectural style for a self-healing system because certain types of tight coupling can make the self-healing mechanism itself vulnerable to being disabled when system components fail, leading to the paradoxical situation where the self-healing system itself may need healing. And since any system where hard-coded references must be recorded in order to change implementation objects is probably too inflexible to use for serious large-scale system development anyway, loosely coupled adaptation frameworks are far more practical for large, complex systems. Loosely coupled systems can also exploit the full abstractive and adaptive power of the component-connector abstraction, which allows connectors to handle and encapsulate inter-component connection and communication concerns, freeing system components to focus on application-related functionality.

Instance vs. Intent-based Selection

How the adaptation system or its software components specify one another when they need to request a service (e.g., reconfiguration) is another important aspect of a self-healing architecture, and is closely related to the degree of inter-component coupling.

Specifying a particular implementation of a monitor or configurator leads to logical dependencies, or *logical tight coupling*, among the adaptation mechanism entities. A better design is for the adaptation mechanism to be designed so that its software components identify one another by functional *intent* [11], that is, by their logical or functional role in the system. This requires all adaptation entities to be identifiable by functional role, and it requires the system to include a directory service, service provider request mechanism, or similar functionality that the system or its software components can use to locate and establish a connection with the appropriate component or software components that will fulfill the required service. Note that intent or functionality-based specification only results in looser coupling if the supporting directory or service request mechanism is flexible (that is, functional role to component instance mappings are not hard-coded, the system supports adding and removing component instances at runtime, etc.).

Peer-to-Peer vs. Hierarchical Organization

Peer-to-peer approaches allocate self-adaptive functionality to symmetrical sets of monitor and configurator peer software components.

Each monitor interacts with a peer-level configurator, and vice versa. The configurator peer may delegate configuration duties, or escalate to a higher-level configurator if necessary (see the aggregator-escalator-peer style below). Similarly, the monitor peer may pass alerts and other messages on to a higher-level aggregator monitor, which communicates with its higher-level configurator peer.

Peer-to-peer styles are simple and symmetrical, and are similar in many ways to common network protocol architectures. However, more hierarchical approaches allow monitor messages to reach higher-level configurator or configuration manager components, which are able to make more comprehensive subsystem- or system-level reconfiguration decisions if needed.

Single vs. Multiple Dispatch Configuration

It may be desirable to use a “chain of commands” style of configurators; if one configurator cannot handle a given configuration request, or if a configurator unsuccessfully attempts to handle a request, the configurator or the configuration manager can pass the request to the next configurator in the chain.
Alternately, the configuration functionality can be allocated to a sequential or branching chain of configurators; each configurator performs its part of the configuration and then passes the request to the next configurator until the request has traversed the entire chain or tree, and all the configuration actions have been performed.

X. ARCHITECTURAL APPROACHES

In this section, we present and discuss several reference architectural styles that illustrate the architectural design issues from the previous section. Each architecture, include a diagram, a brief explanation of how the architectural style works, and a short discussion of some of its strengths and weaknesses as they relate to the previously discussed design issues, and any other issues that may affect the utility of the style for practical system design. Figure 2 shows Peer-to-Peer architectural style.

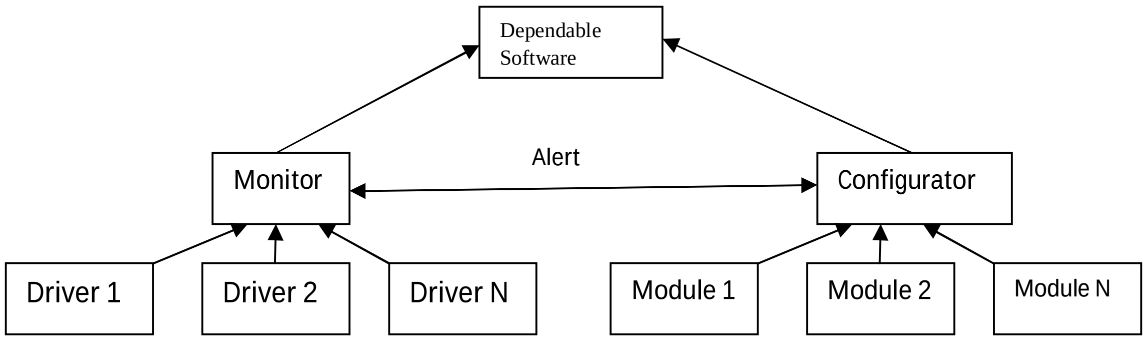

Fig. 2: Drivers/Monitors Peer-to-Peer architectural style

ISSN(Online):2320-9801
ISSN(Print): 2320- 9798

International Journal of Innovative Research in Computer and Communication Engineering

(An ISO 3297: 2007 Certified Organization)

Vol. 2, Issue 4, April 2014

Peer-to-Peer

The peer-to-peer architectural style is a simple approach that allocates a monitor component to monitor each driver of the system or environment that needs to be monitored, and a peer configurator component to reconfigure the system module (or delegate configuration; see, e.g., the aggregator-escalator-peer style below). This style is similar to network protocol architectures where each level in the protocol stack has a peer-level counterpart at the other end of the communication link. Such a strict peer-to-peer approach, while conceptually simple, has serious drawbacks for most self-healing applications, because an actual configurator or configuration manager may require output from more than one monitor to make a decision about the optimal reconfiguration actions to take. As a result, the peer-to-peer approach needs to be combined with one or more additional strategies (e.g., aggregator-escalator-peer or chain-of-configurators, described below) to be useful.

Aggregator-Escalator-Peer

The aggregator-escalator-peer adaptation style overcomes some of the limitations of a strictly peer-to-peer monitor-configurator approach by allowing monitors to pass their outputs to higher-level aggregator monitors, which then package the combined output from the lower-level monitors into a coherent composite package. This composite output is then passed to a peer configurator, which benefits from getting a single consistent picture of related monitor outputs, instead of receiving all the outputs separately, and having to either cache the data until it has received enough, or worse, having to respond to each low-level monitor alert separately.

Figure 3 shows a simple example that demonstrates how a self-healing system can use an aggregator-escalator-peer adaptation architecture to aggregate output from multiple environmental monitors, and pass a comprehensive set of monitor output data to a higher-level configurator that will be able to make better configuration decisions and operate more efficiently than it would if it received and responded to each low-level monitor alert separately.

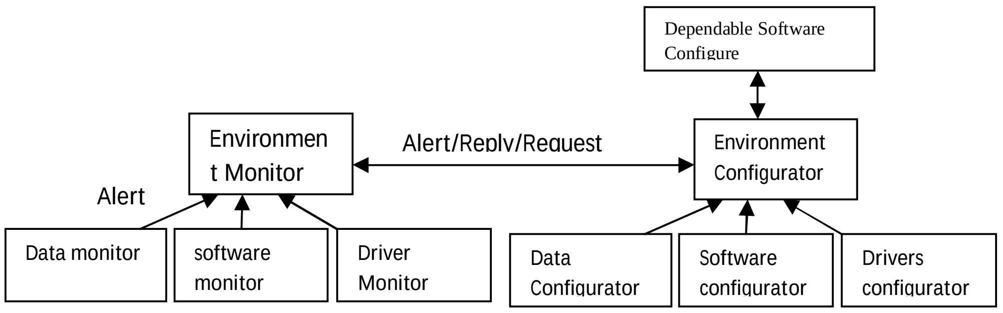

Fig. 3: Aggregator-Escalator-Peer architectural style

Chain-of-Configurators

The chain-of-configurators architectural style is really two different styles. In the first variation, similar to the Chain-of- Responsibility design pattern, multiple configurators are chained together in a linear or other traversable structure (e.g., tree). The configuration request is passed along the configurator chain until a configurator is able to successfully handle the request. This enhances loose coupling, and also makes it easy to implement runtime addition and removal of configurator instances. The chain-of-configurators style allows self-healing systems to try all available configuration strategies for repairing a given problem.
The system can then promote successful strategies and demote or prune less-successful strategies while the system is running, just by manipulating the list.

The second chain-of-configurators variant is similar to the first, except that it utilizes a Visitor pattern, in which configurator chaining is used to compose higher-order configuration functionality using a group of lower-order configurators. This visitor-style variant enhances the power and flexibility of the adaptability mechanism by allowing the

ISSN(Online):2320-9801
ISSN(Print): 2320- 9798

International Journal of Innovative Research in Computer and Communication Engineering

(An ISO 3297: 2007 Certified Organization)

Vol. 2, Issue 4, April 2014

adaptation reasoning or configuration management mechanism to construct new configuration solutions at runtime from existing lower-level solutions. While the visitor variant does not enhance loose coupling by itself, combining the two approaches can lead to powerful and flexible configuration solutions.

The visitor-style variant of the chain-of-configurators can also be used to implement configuration functionality similar to the aggregator-escalator-peer style in peer-to-peer monitor-configurator systems; in this case, the configurator chain aggregates the output from all the necessary peer-level monitors, and passes it on to the next higher-level configurator. Since both variants of the chain-of-configurators style lack any intrinsic organization, beyond the traversal order of the list or tree data structures used, the configuration mechanism or human system engineer must enforce any ordering constraints, etc., on the set of configurator software components in the chain to achieve software dependability. *Figure 4* depicts the chain-of-configurator-modules architectural styles.

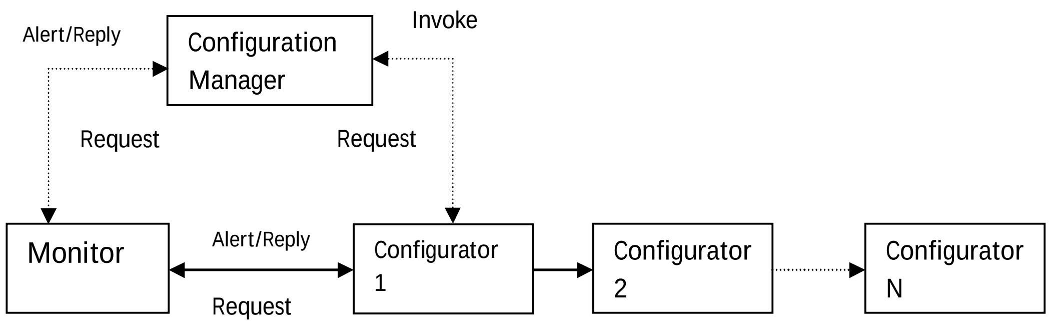

Fig. 4: Chain-of-Configurator-modules architectural style

Configuration Manager

The configuration manager architectural style utilizes a configuration manager component to centralize and encapsulate the logic involved with reasoning about configuration changes. In addition, the configuration manager is typically also the central communication clearinghouse for the other adaptation entities (e.g., monitors and configurators), since the configuration manager must ultimately make any reconfiguration decisions and communicate those decisions to the configurator software components that will be involved in making the required changes. Figure 5 shows the basic configuration manager style for dependable software system.

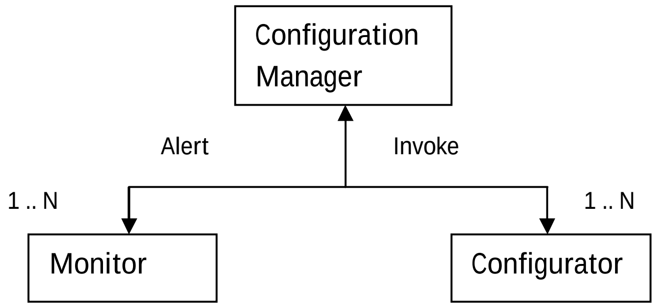

Fig. 5: Configuration Manager architectural style

Configuration manager architectures can be combined with other architectures, e.g., aggregator-escalator or chain-of-configurators), but regardless of other architectural styles used, the centralized control imposed by a configuration manager strongly affects the fundamental nature of the self-configuration mechanism. While the peer-to-peer approaches are intended to support more decentralized and distributed modes of inter-component communication and configuration reasoning, a configuration manager centralizes at least the software system's self-healing logic, and closely related supporting functionality such as learning and planning. A configuration manager may still utilize loose coupling if it delegates monitoring and reconfiguration, and the system includes a mechanism that supports functional intent-based component specification or a similar technique.

XI. CONCLUSIONS

Since dependable systems must often are implemented using less than - dependable platforms and networks, and even dependable hardware and software still fails occasionally, especially in the presence of unexpected user input or other data and state-related issues, self-organizing-self-healing systems are an important strategy for software engineers who develop dependable systems. We have presented the potential of self-organizing-healing mechanisms to enhance system dependability; then, starting from the basic requirements of self-organizing and healing systems, we present a general strategy for dependable systems, discuss several important architectural design issues, briefly analyze several architectural styles for self-healing systems, and use that analysis to recommend several specific architectural style directions that show the most promise for implementing dependable self organizing and a self-healing systems.

Finally, to enable system architects to more easily select and incorporate appropriate dependable architectures, and to promote separation of concerns between application and dependable architectures, we proposed the following enhancements to current architectural description languages (ADLs) and system design environments to incorporate explicit support for self-organizing and self-healing architectural frameworks:

• *Self-organizing ADLs*: Add new syntactic constructs to specify, define and combine architectural strategies for self-organizing systems, along with component-level interfaces for runtime self-organization.
• *Self-healing ADLs*: Add new syntactic constructs to specify, define and combine architectural strategies for self-healing systems, along with component-level interfaces for runtime monitoring and configuration, etc.
• *System design toolsets*: Enhance with explicit self-organizing, self-healing system support, including self-adaptive architectural styles.
• *Visualization*: Add self-organizing, self-healing design views such system *aspects views*; structural views; drivers or *monitor views*; *alert condition editors*; modules-*configurator views*; and *alert type views*.
• *Additional requirements*: Enhance *meta-data* support, e.g., for system- and component-level properties that may be monitored.

Adding capabilities such as these to ADLs and system design tools promises to contribute significantly to the productivity of system designers who need to design architectures for self-organizing-healing dependable software systems (SOHDSS).

REFERENCES

[1]. Alberts, B., Alexander, J., and Peters, W., "The shape and structure of proteins, molecular biology of cells", New York, Garland Science, ISBN 0-8153-3218-1, 2002.
[2]. Blair, G., Coulson, G., Blair, L., Duran-Limon, H., Grace, P., Moreira, R., and Parlavantzas, N., "Reflection, Self-Awareness, and Self-Healing", WOSS'02, 2002.
[3]. Boesen, M. R., Madsen, J., and Keymeulen, D., "Autonomous Distributed Self-organizing and Self-healing Hardware Architecture, The eDNA concept", Aero-space Conference, IEEE, 2011.
[4]. Brandozzi, M., and Perry, D., "Architectural Prescriptions for Dependable Systems", WADS, 2002.
[5]. Camazine, S., Deneubourg, J., and Bonabeau, E., "Self-Organization in Biological Systems", Princeton University Press, ISBN 0-691-11624-5, 2003.
[6]. Cheng, S., Huang, A., Garlan, D., Schmerl, B., and Steenkiste, P., "An Architecture for Coordinating Multiple Self-Management Systems". WICSA-4,2004.
[7]. Christian, P., "Self-organization in Communication Networks: Principles and Design Paradigms", IEEE Communication Magazine, 2005.
[8]. Dicke, E., Byde, A., Cliff, D., and Layzell, P., "An Ant-inspired technique for storage area network design", Proceedings of Biologically inspired approaches to advanced Information technology LNCS 3141 pp 364-379, (2004),.
[9]. Douglass, B. P., "Doing Hard Time: Developing Real-Time Systems with UML, Objects, Frameworks and Patterns". Object Technology Series.Addison-Wesley, Reading, MA, 1999.
[10]. Doursat, R. and Ulieru, M., "Emergent engineering for the management of complex situations", 2nd Int'l Conf. Autonom. Comput. Commun. Syst.(Autonomics), 2008.

ISSN(Online):2320-9801
ISSN(Print): 2320- 9798

International Journal of Innovative Research in Computer and Communication Engineering

(An ISO 3297: 2007 Certified Organization)

Vol. 2, Issue 4, April 2014

[11]. Garlan, D., Cheng, S., and Schmerl, B., "Increasing System Dependability through Architecture-based Self-Repair, Architecting Dependable Systems", LNCS 2677, Springer-Verlag, 2003.
[12]. Garlan, D., and Schmerl, B., "Model-Based Adaptation for Self-Healing Systems", WOSS'02, 2002.
[13]. Hansson, H., Akerholm, M., Crnkovic, I., and Torngren, M., "SaveCCM – a Component Model for Safety-Critical Real-Time Systems". In Proceedings of 30 Euromicro Conference, Special Session Component Models for Dependable Systems, 2004.
[14]. Hawthorne, M., and Perry, D., "Applying Design Diversity to Aspects of System Architectures and Deployment Configurations to Enhance System Dependability". WADS'04, 2004a.
[15]. Hawthorne, M., and Perry, D., "Exploiting Architectural Prescriptions for Self-Managing, Self-Adaptive Systems: A Position Paper", WOSS'04 2004b.
[16]. Kauffman, S. A., "The Origins of Order: Self-Organization and Selection in Evolution", Oxford University Press, 1993.
[17]. Koopman, P., "Elements of the Self-Healing System Problem Space", WADS'03, 2003.
[18]. Lemos, R., and Fiadeiro, J., "An Architectural Support for Self- Adaptive Software for Treating Faults", WOSS'02, 2003.
[19]. Melinda, J. T., and David, A. C., "Cellular pattern formation during retinal regeneration: a role for homotypic control of cell fate acquisition", Vision Research 47(4) 501-511, 2007.
[20] Myrna, E., "Self-organizing Natural Intelligence: Issues of Knowing, Meaning, and Complexity", Springer-Verlag, 2006.
[21]. Shaw, M., and Garlan, D., "Software Architecture: Perspectives on an Emerging Discipline". Prentice-Hall. 242p. ISBN 0-13-182957-2, 1999.
[22]. Magge, J., Georgiadis, I., and Kramer, J., "Self-organizing software architectures for distributed systems", In WOSS'02, 2002.
[23]. Meystel, A., "Measuring Performance of Systems with Autonomy: Metrics for Intelligence of Constructed Systems", White Paper for the Workshop on Performance Metrics for Intelligent Systems. NIST, Gaithesburg, Maryland, August 14-16, 2000.
[24] Okon, S. C., "A Fail-safe Strategy for Scientific/Engineering Project: A Tool for Sustainable Development", Journal of Sciences and Technology Research Vol. 5 No. 2, pp 6-9, 2006a.
[25] Okon, S. C., "Microprocessor Devices (Computer) like Man: A Self-organizing System", Journal of Research in Physical Sciences Vol. 2 No. 4 pp1-7, 2006b.
[26]. Okon, S. C., and Asagba, P. O., "Deploying Self-organizing-healing Techniques for Software Development of Iterative Linear Solver". International Journal of Computational Engineering Research, vol 3 no 2, 2013a.
[27]. Okon, S. C., Asagba, P. O., and Aloysius, A., "Self-organizing-healing paradigm for Mobile Networks", International Journal of Advanced Research in Computer Science, vol 4 no 4, 2013b.