An Adaptable and Generic Fault-Tolerant System for Distributed Applications

Ouanes AISSAOUI
Department of Computer Science
Badji Mokhtar University
Annaba, Algeria
aissaoui.ouenes@gmail.com

Abdelkrim AMIRAT
Department of Computer Science
Med Cherif Messaadia University
Souk Ahras, Algeria
amirat_karim@yahoo.fr

Fadila ATIL
Department of Computer Science
Badji Mokhtar University
Annaba, Algeria
atil_fadila@yahoo.fr

Abstract— The work presented in this paper is inscribed within the framework of the autonomic computing which is an initiative started with IBM in 2001. Its final goal is to develop self-managed computing systems and to overcome the rapidly growing complexity problem. In this work, we treated the property self-healing of the autonomous systems on account of its importance in the development of the critical systems and with high availability requiring a self-protection against the software failures for ensuring the service continuity. This paper presents GAFTOS (Generic and Adaptable Fault-TOlerant System), a fault-tolerant system adaptable and generic for the distributed and dynamic component-based applications. We illustrate the expressive power of GAFTOS by encapsulating the following fault tolerance techniques in the sub-components of GAFTOS: primary-backup, message store, distributed backward-recovery and distributed checkpointing.

Keywords— Adaptable and distributed component-based applications; Fault tolerance; Reliability; Software architecture

I. INTRODUCTION

An autonomous system [1] is a system which makes only decisions, by using policies on a high level. It will check and optimize constantly its statute and will adapt automatically to the changing conditions. Its objective is to detect, diagnose, and repair the localized problems resulting from the bugs or the failures, in the software and the hardware. For this reason, researchers in this axis study each property separately. The more studied property is the self-adaptation due to the need of development of critical applications with high availability and which require an update at runtime. In parallel to the need for the applications adaptation, one problem occurs concerning their reliabilities which is an important attribute of running safety [2]. We say that a system is reliable if it's in conformity with a given specification. Although the importance of this attribute, it was not well studied in the proposed components models and adaptable architectures models. This can be justified by the concentration of researchers on how to make the applications adaptable during their execution.

One of the technological challenges that must be solved before such critical applications can be used in exploitation is the adoption of fault-tolerance techniques. Unfortunately, the fault-tolerance protocols are widely regarded as complex and implementing them correctly is likely to overwhelm all except the best programmers.

The works which studied reliability in the component-based applications, and more particularly in the dynamic applications, do not ensure the desired level of reliability. It's only a simple mechanisms integrated in the application components or only in the components adapters. Moreover, the code charged to adapt the application is mixed with that charged to make it reliable [2], [3], [4], [5]. Note that this mixture prevents the reliability mechanism evolution and this should not be fixed. Also, the techniques implemented in the reliability mechanisms are limited to the backward recovery or the transaction system.

After having identified this problem and this lack, and in order to provide a certain degree of autonomy to the applications, we propose the utilization of mechanisms which take into account some of the enumerated issues concerning reliability. Such techniques are put together in a same system which we call GAFTOS (Generic and Adaptable Fault-TOlerant System). Its aim is to increase the reliability of the distributed and dynamic component-based applications. The objective is not to introduce fault-tolerance in components, but rather to make the component platform more fault-tolerant with the ability to automatically recover from errors.

This paper is organized as follows: the second section describes the architecture of our system GAFTOS; the third section exposes the fault-tolerance techniques which we have adopted for the definition of our system; the fourth section outlines in detail the fault-tolerance manager of GAFTOS; in section 5 is presented the current state of the GAFTOS's implementation. Section 6 analyses related proposals found in the literature. Lastly, section 7 concludes this paper.

II. ARCHITECTURE OF OUR SYSTEM (GAFTOS)

For the definition of our system GAFTOS, we consider a set of constraints which are: (i) independence of the existing standards and component models, (ii) flexibility and extensibility of the system, (iii) modularity and adaptability of the fault-tolerance management mechanisms, (iv) taking into account the distributed nature of the software to make it reliable and of the heterogeneity of their execution environments.

Our approach solves the reliability problem already discussed by viewing the applications insensitive to failures as a set of four parts: the application, the translation infrastructure, the fault-tolerance manager and coordinators. The application programmers should concentrate only on the functional code of the application rather than on the code that make it reliable. GAFTOS is a fault-tolerant system which we propose for all distributed component-based applications and, in particular, dynamic applications, in order to make these applications more sure. It is an adaptable system because it benefits from the flexibility which results from the adoption of the component-based development. The figure 1 represents this structure.

978-0-7695-4959-0/13 $25.00 © 2013 IEEE
DOI 10.1109/ACSAT.2012.63

CPS Conference Publishing Services

We treat in this work the distributed and dynamic component-based applications where the application components are distributed on several sites. For that, and according to our approach, application can contain more than one fault-tolerant system which can be distributed on several sites, plus a set of components representing the business logic of the application and which can be distributed on several sites. At one of these sites, we must find a component <<Infrastructure translation>>, a component <<Coordinators>> as well as a file representing the application log plus an XML file which describes the architecture of this last (figure 1).

The part of Translation Infrastructure. It allows establishing a causal connection between the application architecture description and the system at execution. This connection facilitates the application control and the management by the administrators or other software entities. This part is used by the recovery manager <<*RecoveryManager*>> (see section IV) each time that an operation of recovery is carried out to cancel the effect of the executed actions of the adaptation operation or to replace the state of each modified component by its state saved before the execution of the adaptation operation (see section IV). This part is represented by a component apart, charged to accomplish this task.

In order to facilitate this transition, we adopted the XML format for describing the applications architecture. The designer can use an architecture description language ADL such as the language xADL 2.0 [6].

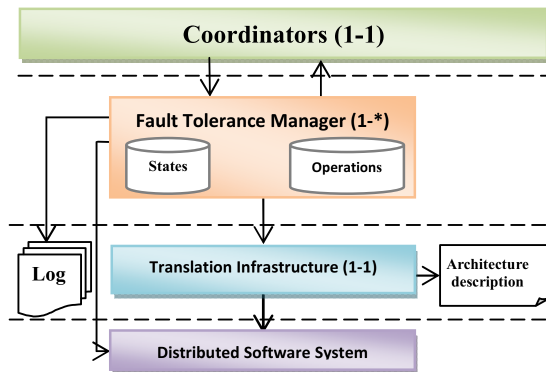

Figure 1. Structure of the applications integrating GAFTOS

The part of Fault tolerance manager. It represents the heart of our fault-tolerant system GAFTOS. It ensures the application service continuity. We also propose a component-based structure for this manager i.e. that it's a composite; or each his sub-component provides a precise function. In order that this manager achieves his goal, it contains a fault-detection component, a management component of service quality, a recovery component, an application structure checking component and components-behavior checking component plus an executor component. See figure 2.

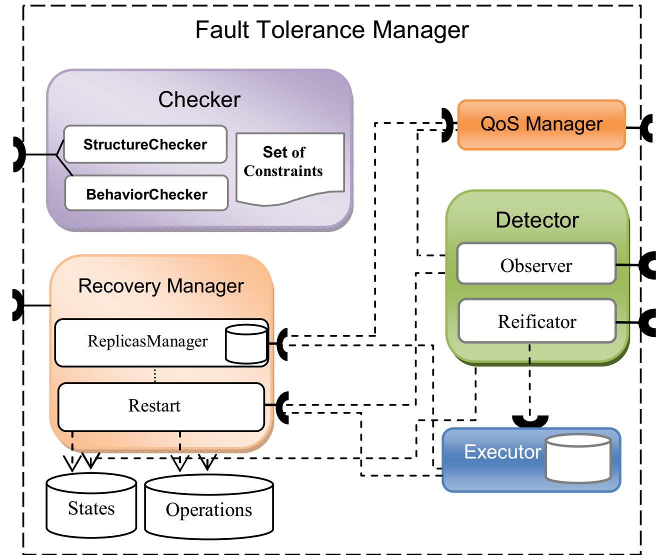

Figure 2. Fault-tolerance Manager Structure

The part of Coordinators. We speak in our context about the distributed applications. Therefore, the coordination for the recovery is a very important point. For that, we have introduced a composite component composed of two sub-components <<CheckepointingCoordinator>> allows the coordination for the distributed checkpointing and <<RecoveryCoordinator>> allows the coordination in the case of the necessity of the rollback. The fault tolerance techniques used

III. The Fault Tolerance Techniques Used

In this section, we will present the techniques which we have implemented in our fault-tolerant system which are: distributed checkpointing, primary-backup, message store and the distributed backward recovery. Our goal is to use these techniques for providing a general solution in order to benefit from the proposed mechanisms.

A. Distributed checkpointing

As we speak in our context about the distributed applications, the coordination for the distributed checkpointing is very important. The basic idea behind this safeguard is to ensure that either all objects in an application checkpoint or none do. The set of local checkpoints taken must form a consistent global state—all methods received by an object must be recorded as having been sent [7].

B. primary-backup

The highly available services can be achieved by replicating the servers and there by introducing redundancy [8]. If one server fails, the service is still available since there are other servers that are able to process the incoming requests. There are two main techniques for achieving such software-based redundancy: active replication and primary-backup [9]. In this work we are interested in the primary-backup technique for its advantage in comparison with the active replication technique.

In the primary-backup (1) a primary object receives and responds to all messages, (2) When the primary object finishes the service a method of update sends the state updates to the backup service before answering the client,(3) finally, the primary object sends the answer to the clients. It also sends regular heartbeats so that the backups are able to detect when the primary server fail.When a primary fails, the backup execute the following invocations of methods.

The primary-backup has as consequence a faster recovery of the failed objects because the backups are already in activity and ready to serve the calls of methods. With such significant reduction of the recovery average duration, a failure recovery can be perceived as delays. The implementation of this technique is represented by the sub-component <<ReplicasManager>> of the component <<RecoveryManager>> (see section IV).

When the replication technique does not guarantee the masking of faults for the reason of the software crash (for example a problem in a component requires the search of a coherent state to continue processing) or for the reason of material problems, the recovery will be the best solution.

C. *Backward recovery*

In the backward recovery, the applications must be rolled back to a previously consistent state in order that it continues processing normally [7]. For that, a set of check-points must be saved each time that it's necessary. The major problem of this technique is that the recursive execution of the recovery process on an object can lead to the problem DOMINO i.e. that the component could be in its initial state [9]. For the mapping of the backward recovery technique on our system we have adopted this technique in our fault-tolerant system, and we have avoided the problem DOMINO by the use of the reification technique which consists to intercept the input and output calls of each component for extracting the useful informations representing the state of the server and client objects in order to record them before serving each request.

One of the problems which can be posed in the distributed applications is the assurance of the message transmission of a process to another. To overcome this problem, we use the message store technique described in the next paragraph.

D. *Message Store*

Message store systems deal with the problem of reliably delivering messages from one process to another. The basic principle is that a local message queue handler can store messages on non-volatile storage [8]. Once the client delivers the message to the message queue handler, the client is relieved from any additional concerns of delivering the message. The message queue handler delivers the message to the server. If the server is down at the time the client sends the message, the message queue handler simply waits until the server comes up. If the message queue handler crashes, the message remains in the storage. The message will be delivered to the final (or next) destination when the message queue has recovered.

We integrated this technique in the component <<Executor>> of the fault-tolerant manager and also in the replication manager (the component <<ReplicasManager>>) (see the section IV).

The use of these four techniques (primary-backup, message store, distributed backward recovery and distributed checkpointing) allowed us to develop a more powerful fault-tolerant system than the majority of the proposed mechanisms in the literature and adaptable because we adopted the component-based development for the implementation of this system.

IV. Description of the fault-tolerance manager of GAFTOS

In this section we describe in detail the sub-components of the fault-tolerance manager.

A. *The component << Checker>>*

This component plays a very important role in the preservation of the application coherence because it allows the checking of the applications components conformity to their component model. Moreover, it allows making a structural checking of the application and a behavioral checking of the components. For that, it has two sub-components <<CheckerStructure>> and <<BehaviourChecker>>; the first realize the structure checking, whereas the other allows the components behavior checking. Note, that this operation of checking takes place at the architecture description (an XML file). To attain this objective, each sub-component has a special algorithm which will check a set of constraints described in a file apart (For the format of constraints, we propose the use of an XSD file representing the meta-model). For the checking of the behavior, we consider only the checking of the component attributes.

The component < < checker > > can be used by the probes which control the application components or by the adapters - as we speak here about the component-based dynamic applications- for the checking of the respect of the architectural and behavioral constraints which must be respected.

B. *The component <<Detector>>*

The role of this component is the monitoring of the application and hardware components and also the reification of the components methods calls (i.e. the service request) for detecting the faults which can appear in the application and more precisely the components' crashes. For that, this component is composed of two types of sub-component; components of the type <<Observer>> charged of the monitoring of the application components. They ping periodically the components which they supervise for detecting the failed components and for treating these faults thereafter with the component <<RecoveryManager>>.

The component << Reificator>>, its role is to reify the calls of methods. For that, its implementation must be based on the aspects which intercept each call of service of any component in the sub-system controlled by its fault tolerance manager. The objective of this reification is firstly, the safeguard of the state of the caller and callee (source and target object) before the realization of the service, and also the safeguard of the service request (operation) in the operations warehouse for a possible use (case of backward recovery). Therefore, at the interception of a method call, the <<Reificator>> extracts the target and source objects and effects the recording of their states in the

warehouse of states via the sub-component <<GAFTOS-checkpoint>> of the component <<RecoveryManager>> (the section d). Then, it safeguards the service request in the operations warehouse. Finally, it passes the request to the component <<Executor>> which supervises and manages the implementation of the request (figure 3). The following section describes this component.

C. The component Executor

This component undertakes the management and the control of the requests execution sent by the component <<Reificator>>. For that, it contains a message storage system which manages a queue used for the safeguard of the requests sent by the component <<Reificator>>. At the reception of a request, this component records the latter in its messages warehouse. Then, it seeks if the component "provider" of the required service is a duplicated component or not. If it's the first case, it sends this request to the sub-component <<ReplicasManager>> of the component <<RecoveryManager>>, and the latter manages the request treatment by the replicas group (figure 2). Else, if the component "provider" of the required service is not duplicated, the executor itself sends the request to this provider and awaits the reception of the result around a certain time indicated by the component <<QoS Manager>>. If the time passes and the executor does not receive a result, it detects therefore that the component (provider of service) is failed, and in this case, it launches a recovery request to the sub-component <<Restart>> of the component <<RecoveryManager>> for tolerating the application to this faults.

The figure 3 presents a sequence diagram illustrating the running of an execution without failure of a request sent by a component C to a component P. This diagram summarizes much more the operation of the <<Reificator>> and the <<executor>> through the running of this execution. This diagram contains an operator "alt" indicating an alternative. It represents two possible behaviors: if the component P (server) is duplicated, there is a precise operation to carry out, else there is others operations to effect.

Note that, the <<Reificator>>, when it intercepts the replies of the servers sent by the executor, it doesn't make any thing, only it passes them to the clients.

D. The component Recovery Manager

This component plays a very important role in the application. It treats the faults detected by the component <<Detector>> and also those detected by the adapter via the component <<Checker>> (for the applications that contains such component). The two fault-tolerance techniques (Primary-backup and backward recovery) are implemented in this component.

In order that the manager of recovery reaches these envisaged objectives, we have decomposed it on two sub-components; the first is called <<ReplicasManager>> implementing the primary-backup technique, and the second <<Restart>> implementing the backward recovery technique.

The component <<ReplicasManager>>. It Implements the primary-backup technique described previously. It receives the requests of the component <<Executor>> and then launches, according to the primary-backup strategy, the execution of the called method on the primary server (component). When the primary replica crashes, the component <<ReplicasManager>> removes this component and replaces it by a backup server. Since the old primary can be failed before sending the return value to the clients, the new primary server returns the last return value. Thus, the clients can receive the replica return value. We suppose that the clients can handle the double values.

The component <<Restart>>. This component performs the backward recovery if a component crashes or an adaptation operation violates the structural or behavioural constraints defined by the component model used to develop the application components. Moreover, it performs the recording of the check-points in the states warehouse via its sub-component << GAFTOS-Checkpoint>>.

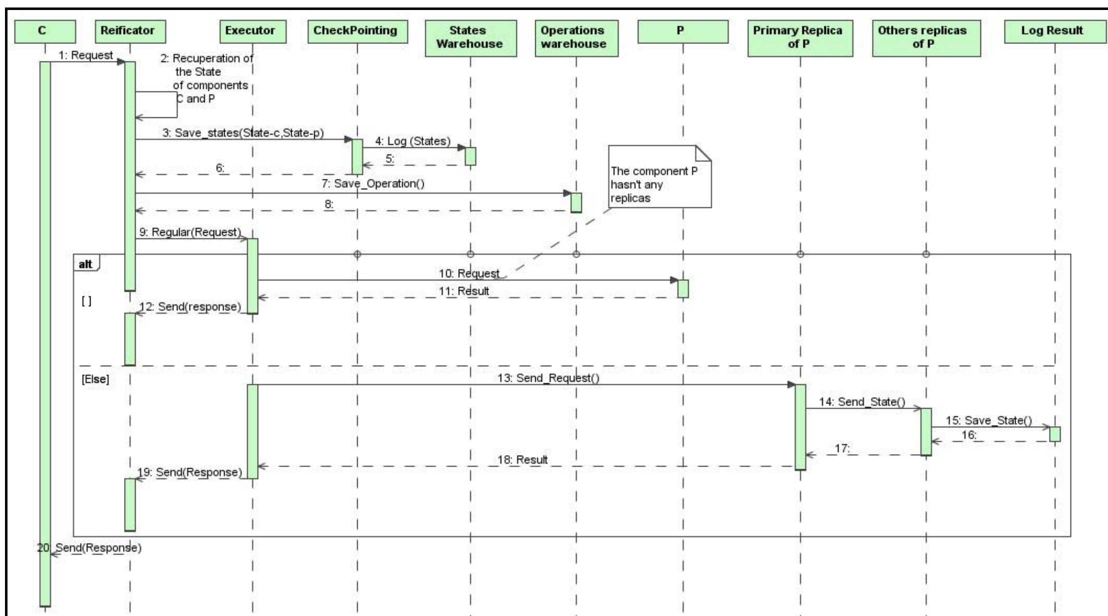

Figure 3. An execution without breakdown of a component C request by a component P.

When the sub-component <<Observer>> of the component <<Detector>> or the replicas manager detects that one component is failed, they call the recovery function of the sub-component <<Restart>> which will carry out the recovery or the restart of the component as the components which depend on this last and which exist as well in the site of the failed component or, in other sites if the recovery does not solve the problem. Also, if the application adapter detects via the component <<Checker>> that an adaptation operation cannot finish correctly or it violates the constraints defines by the component model, it calls the recovery function of the component <<Restart>> which execute the opposite operations of the adaptation operation actions to cancel their effects. Next, it replaces the states of the modified components and those which depend on the latter by their states saved in the last check-point.

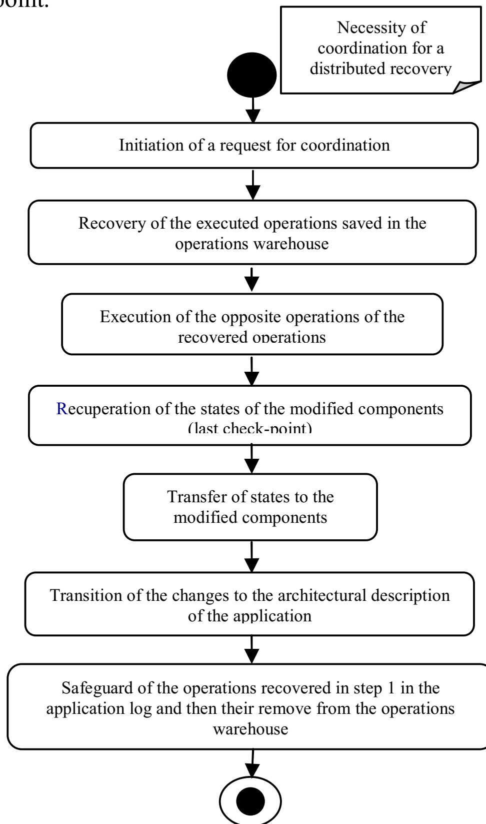

Figure 4. A recovery diagram in the case of the violation of the application architectural constraints.

Note, that we speak here about applications, distributed on several sites, so, there is dependence between the distributed components which requires a recovery of all these components, therefore coordination for distributed recovery, and before this coordination for distributed checkpointing are necessary. After the realization of the recovery, the recovery manager calls the translation function of the component <<Translation Infrastructure>> to transfer the changes on the system architecture description. (Figure 4).

E. The component <<QoS Manager>>

It is a component used for controlling the service quality level. It represents a graphical interface from which the user can increase or decrease the level of the quality of the service. We have the possibility to define the waiting time of answer by the executor <<Executor>>, the waiting time of request confirmation, the interval of time during which an observer ping the component (s) which it supervises, the replica max number... This component is used by the component <<Executor>>, sub-component <<ReplicasManager>> of the component <<RecoveryManager>>, sub-components <<Observer>> of the component <<Detector>>.

V. CURRENT STATE AND EXPERIMENTATION PERSPECTIVE

The architecture of our fault-tolerant system model GAFTOS which we propose here is independent on the component models i.e. it can be implemented by any model provided that it is reflexive or adaptable via a set of controllers that it comprises, because a fault-tolerant system should not be fixed. A prototype of our system is in the process of implementation. We propose the use of the component model Fractal [10], an open model, so, easily extensible. Also, it's reflexive and recursive (hierarchical at several levels) with concept of division. Fractal authorizes a definition, a configuration and a dynamic reconfiguration of a component-based architecture as well as a separation of the functional and non-functional concerns. The reification of the architecture at runtime and the capacities of introspection and intercession in particular allow building adaptive systems. Another advantage of the model is that it covers completely the life cycle of the software: since the design of its architecture, its development and until its administration.

VI. RELATED WORK

The problems treated in this paper accost the domain of research around the fault-tolerance of the computing systems and in particular, the distributed component-based systems. The fault-tolerance in the distributed applications is largely studied [11],[12],[13],[14]. For the component-based applications, we can cite for example the work of Jean-Charles Fabre et al. [15], which propose fault-tolerant architecture adaptable to the execution conditions of a system. This architecture is composed of three levels including two meta-levels. Level 0, also called basic level, contains the functional application. Its meta-level meta-1 regroups the algorithms of fault-tolerance which are applied to the functional application. Lastly, the meta-level meta-2 treats the management problem of the modification at runtime of the fault-tolerance. The choice of strategies of fault-tolerance and their implementations in components is left to the responsibility of the designer. Moreover, it doesn't fix a clear decomposition of the fault-tolerance.

AFT-CCM (Adaptive Fault-Tolerance one the CORBA Component Model) [5] is a model which adds the support of the fault-tolerance to the component model CORBA by using a set of non-functional components which control and supervise the requirements of the quality of service QoS defined by the user and effects the adaptation all times which is necessary. The only technique of fault tolerance used here is the replication which implies that the faults model of this framework is limited to the crash of components. Moreover, this solution strongly depends on the component model CORBA.

In [16], the authors present a hierarchically-structured fault-tolerant architecture for component-based robot systems. The framework integrates widely-used, representative fault-tolerant measures for fault detection, isolation, and recovery. The system integrators can construct fault-tolerant applications from non-fault-aware components, by declaring fault handling rules in configuration descriptors or/and adding simple helper components, considering the constraints of components and the operating environment. This framework integrates several techniques of fault tolerance which allows it to tolerate several types of faults. From a viewpoint of adaptation it is not adaptable. Also it is not generic because it is conceived specifically for the systems of robot.

The works which studied reliability in the dynamic component-based application did not hold the desired level of reliability. It's only a simple mechanisms integrated in the application components or only in the components adapters. For example, in [2] the authors propose a definition of consistency for configurations and reconfigurations in the Fractal component model with a model based on integrity constraints like for example structural invariants. Reliability of reconfigurations is ensured thanks to a transactional approach which allows both to deal with error recovery and to manage reconfiguration concurrency in systems. The use of the transaction system for making the applications reliable is a widely adopted method [17], [18], [19] but only the work presented in [2] takes into account the four properties ACID (Atomicity, Consistency, Isolation, Durability) of the transactions.

In contrast to the other approaches, our system GAFTOS allows to make the platform of components more fault-tolerant with the ability to automatically recover from errors. Our solution is for universal use, but it can be specialized according to the needs of a particular solution. Also, it is independent on the standards and the component models. Moreover, it proposes a clear decomposition of the fault tolerance in software components.

VII. Conclusion and prospects

A various motivations were presented in this paper, aiming the construction of surer fault tolerance system mechanisms. This article had developed a fault-tolerant system that we call GAFTOS (Generic and Adaptable Fault-TOlerant System) for the dynamic and distributed component-based applications. We have supplied a generic solution for the management of the reliability of these applications in order to make them surer.

The System GAFTOS has several advantages. Firstly, it is for universal use; however, it can be specialized according to the needs of a particular solution. Secondly, it is adaptable because it benefits from the flexibility resulting from the adoption of the component-based development. Thirdly, it puts a causal connection between the application architecture description and the system at execution which facilitates the management and the control of these applications and consequently, to make them surer. Fourthly, it allows the verification of the structural and behavioral constraints at the application architecture description. Fifthly, it is independent on the component models and standards. Currently, our system GAFTOS is not able to tolerate the failures in values. For that, in our future work, we plan to make GAFTOS able to tolerate this type of failures.

References.

[1] IBM. An architectural blueprint for autonomic computing. Autonomic computing whitepaper, 4th edition. 2006.

[2] Marc LEGER, " Fiabilité des Reconfigurations dynamiques dans les architectures à composants", thèse de doctorat ; école nationale supérieure des Mines PARIS; 19 mais 2009.

[3] Djalel Cherfour et Françoise André. "Auto-adaptation de composants ACEEL coopérants", dans CFSE'3, Conférence Française sur les Systèmes d'Exploitation, France, Octobre 2003.

[4] Wen-Ke Chen, Matti A. Hiltunen & Richard D. Schlichting. "Constructing Adaptive Software in Distributed Systems". In 21st International Conference en distributed computing system (ICDCS-21), page 635-643, Mesa, AZ, Avril 2001.

[5] joni Fraga, Frank Siqueira, and Fàbio Favarim. "An adaptative fault-tolerant component model". In IEEE Component Society, editor, the Ninth IEEE International WorkShop on Object-Oriented Real-Time Dependable System, Pages 179-186, Capri Island, Italy, 2003.

[6] Eric M. Dashofy, André van der Hoek, and Richard N. Taylor. "A highly-extensible, XML-based architecture description language". In Proceedings of the Working IEEE/IFIP Conference on Software Architectures (WICSA 2001), Amsterdam, Netherlands, 2001.

[7] Chikofsky, E. and Cross II, J. 1990. "Reverse Engineering and Design Recovery: A Taxonomy". IEEE Softw. 7, 1 (January 1990), pp 13-17.

[8] Jari Koistinen, "Dimensions for Reliability Contracts in Distributed Object Systems", Software Technology Laboratory, Hewlett-Packard Laboratories, Technical Report HPL-97-119, October 3, 1997.

[9] Michael R. Lyu, "*Software Reliability Engineering: A Roadmap*", In FOSE'07, Future of Software Engineering, IEEE Computer Society Washington, DC, USA, 2007.

[10] Eric Bruneton, Thierry Coupaye, Matthieu Leclercq, Vivien Quéma, and Jean-Bernard Stefani. "The Fractal Component Model and its Support in Java", Software Practice and Experience, special issue on Experiences with Auto-adaptive and Reconfigurable Systems, 36(11-12) :12571284, 2006.

[11] Anish Arora, Sandeep S. Kulkarni, "Detectors and Correctors : A Theory of Fault-Tolerance Components". 18th International Conference On distributed Computing System (ICDCS'98), Amesterdam, Pays-Bas, mai 1998.

[12] Edsger W. Dijkstra. "Self-Stabilizing Systems in Spite of Distributed Control". Communications of the ACM, 17(11), nov. 1974.

[13] Pankaj Jalote. Fault-Tolerance in Distributed Systems. Prentice Hall, avr. 1994.

[14] Fred B. Schneider. "Abstractions for Fault-Tolerance in distributed Systems". IFIP 10th World Computer Congress, Pages 727–733, Dublin, Irlande, sep, 1986.

[15] Jean-Charles Fabre, Marc-Olivier Killijian, Thomas Pareaud."Towards On-line Adaptation of Fault Tolerance Mechanisms".EDCC 2010: 45-54

[16] Heejune Ahn, Dong-Su Lee, Sang Chul Ahn, "A Hierarchical Fault Tolerant Architecture for Component-based Service Robots", in IEEE International Conf. on Industrial Informatics, July 2010.

[17] Thais Batista, Ackbar Joolia, and Gordon Coulson. "Managing dynamic reconfiguration in component-based systems". *In 2nd European Workshop on Software Architectures (EWSA 2005)*, Springer-Verlag Berlin, Heidelberg, 2005.

[18] Rui S. Moreira, Gordon S. Blair, and Eurico Carrapatoso. "Supporting adaptable distributed systems with FORMAware". In ICDCSW '04 : Proceedings of the 24th International Conference on Distributed Computing Systems Workshops, pages 320-325, Washington, DC, USA, 2004. IEEE Computer Society.

[19] David Garlan, Shang-Wen Cheng, An-Cheng Huang, Bradley Schmerl, and Peter Steenkiste. Rainbow : Architecture-based self-adaptation with reusable infrastructure. *IEEE Computer,* 37(10) :46-54, 2004.