Available online at www.sciencedirect.com
ScienceDirect
Electronic Notes in Theoretical Computer Science
ELSEVIER
Electronic Notes in Theoretical Computer Science 168 (2007) 77–90
www.elsevier.com/locate/entcs

Architecting Fault-tolerant Component-based Systems: from requirements to testing

Antonio Bucchiarone [1]

Istituto di Scienze e Tecnologie dell'Informazione "A. Faedo"
Area della Ricerca CNR di Pisa
56100 Pisa, Italy
antonio.bucchiarone@isti.cnr.it

Henry Muccini and Patrizio Pelliccione

University of L'Aquila, Computer Science Department
Via Vetoio 1, 67100 L'Aquila, Italy
{muccini,pellicci}@di.univaq.it

Abstract

Fault tolerance is one of the most important means to avoid service failure in the presence of faults, so to guarantee they will not interrupt the service delivery. Software testing, instead, is one of the major fault removal techniques, realized in order to detect and remove software faults during software development so that they will not be present in the final product.
This paper shows how fault tolerance and testing can be used to validate component-based systems. Fault tolerance requirements guide the construction of a fault-tolerant architecture, which is successively validated with respect to requirements and submitted to testing. The theory is applied over a mining control system running example.

Keywords: Fault tolerance, Component-based Software Systems, Software Architecture, Testing.

1 Introduction

When engineering complex and critical component-based software systems (CBS) (think to aerospace, transportation, communication, energy and health-care systems), denial of services can have economics consequences and can also endanger human life. *Fault prevention*, *fault tolerance*, *fault removal* and, *fault forecasting* are the four main means developed over the course of the past fifty years to attain the various attributes of dependability [4]. Fault prevention and fault tolerance aim to prevent the introduction of faults or to avoid service failures when faults occur. Fault removal and fault forecasting, instead, mean to reduce the number and

[1] IMT Graduate School, Via San Micheletto, 3 - 55100 Lucca , Italy

doi:10.1016/j.entcs.2006.11.004

severity of faults, and to estimate present and future incidence and consequences of faults. As discussed in [10], fault prevention mechanisms can fail to prevent/remove the occurrence of all faults, and fault tolerance is the most promising mechanism for achieving dependability requirements.

Fault tolerance preserves the delivery of correct services in the presence of active faults and is generally implemented by *error detection* and subsequent *system recovery*. Recently, several studies have evidenced the need of fault tolerance support during the entire system life cycle [24,9], with especial interest at the architectural level [5]. Some colleagues have proposed their solutions for fault tolerance via exception handling at the software architecture and component level (e.g. [24,17,8]).

However, fault tolerance techniques alone are not enough to achieve full guarantee of fault tolerance requirements for critical CBS: unexpected faults or malicious security attacks, in fact, cannot be always avoided nor tolerated [2]. *Our proposal suggests to complement fault tolerance with fault removal techniques in order to reduce the number and severity of all such unexpected faults.* Being software testing one of the major fault removal techniques, we introduce a testing strategy to evaluate, and thereby confirm or improve, system dependability at run-time.

The main goal of this paper is to propose a comprehensive approach for developing and validating CBS systems according to fault tolerance requirements. The approach starts with requirements analysis in order to identify *critical services* (those to be maintained during faults or attacks). Fault tolerance decisions are taken and a fault tolerant component-based software architecture (CBSA) model is realized in order to achieve fault tolerance requirements.

According to the idealized fault-tolerant component model introduced in [19], *normal* and *exceptional* behaviors of system components are specified. While normal responses are those situations where components provide normal services, exceptional responses correspond to errors detected into a component or the entire architecture. Finally, test case specifications are extracted from the CBSA specification to validate the implementation adherence to fault tolerance requirements. During normal execution, the testing approach validates the CBS compliance to normal requirements, in accordance to the architectural specification. During a faulty execution, the testing approach verifies if and how much the system complies to fault tolerance requirements when failures arise.

Section 2 shows related work on the topic. Section 3 presents the main activities for architecting fault tolerant component-based software systems, while details and our proposal application to the mining control case study is illustrated in Section 4. Section 5 concludes the paper and outlines future work directions.

2 Related Work

Some colleagues have proposed their solutions to the problem of incorporating fault tolerance during architectural design. Issarny and Banâtre in [17] and Castor Filho et. al. in [11] share the idea of using the Software Architecture specification as the primary point where to deal with exception handling. In [17] the authors investi-

gate the definition of exceptions at the architectural level. Their model takes into consideration exception handling implemented within components and connectors, and exception handling at the architectural (configuration) level. In [11] the authors propose an initial work on Aereal, a framework to extend architectural descriptions with information about exceptions, defining how exceptions flow between architectural elements. Other approaches (e.g., Rubira et. al. in [24] and Guelfi et. al. in [12]), instead, incorporate exceptional behaviors during the entire development of fault tolerant distributed systems.

Other approaches have analyzed how fault tolerant CBS can be submitted to testing. The MDCE+ method, presented in [8], systematizes the identification, design, implementation, and testing of the exceptional activities in the software development phases. Sinha and Harrold [25] propose an approach for testing the exceptional activity of a system in a white box way. This works only cover unit tests and requires the source code of the tested components to be available. M. Elder in his Ph.D. Thesis [10] identifies the relationships among fault tolerance and dependability, but it does not provide any testing technique to confirm the implementation conformance to fault tolerance requirements.

Differently from related work, this paper describes how fault tolerance and testing can jointly contribute to architect a dependable component-based system.

3 Architecting Fault-tolerant CBS: Overview

In our perspective, three main activities have to be carried out in order to architect and validate a fault-tolerant system:

- In activity *a1*, requirements are elicited. Fault tolerance requirements are specified in order to identify the adequate level of services in case of system damage.

  Fault scenarios are selected (based on the assessment of risks and faults) to evidence how the system should behave in case of faults. Use cases are specified in order to highlight functional requirements, and extended in order to specify critical services.
- In activity *a2*, identified requirements guide the selection of a suitable and fault-tolerant software architecture. Differently from a traditional software architecture model, this specification has to provide a description of both how the CBSA behaves in normal and in exceptional situations, and how faults can be tolerated. Model-checking techniques can be employed to prove the CBSA conformance to fault tolerance requirements.
- In activity *a3* testing is utilized to validate the conformance of the system implementation to fault tolerance requirements, through the fault-tolerant CBSA specification. Test specifications are extracted from the CBSA specification and then run over the CBS implementation to validate the requirements achievement.

4 Architecting Fault-tolerant CBS: Details and Application

In this section we detail the identified activities and apply them to a running example, while taking model-based notations as specification means.

The running example taken into consideration is the mining control system case study [24], a simplified system for the mining environment which handles the mineral extraction from a mine (which produces water and releases methane gas on the air). The system is composed by three main sub-systems: *Pump Control*, *Air Extractor Control*, and *Mineral Extractor Control*. In the following we explain and apply the methodology to the Air Extractor Control sub-system, in order to better explain the theoretical aspects detailed in the following.

4.1 Activity a1: Fault tolerance Requirements Identification

We start with the identification of two classes of requirements: *primary* and *auxiliary*. A *primary requirement* is the one that must be always satisfied, while *auxiliary requirements* are requirements that can be set aside in degraded operating modes. In the presence of faults, auxiliary requirements can be provided in a degraded way.

The next step of *activity a1* is the Use Case diagram realization. From the primary and auxiliary requirements we can define two kinds of Use Cases, N_UCs that are use cases used for describing normal functionalities and exc_UCs used for describing exceptional functionalities.

When normal and exceptional use cases are identified, the adequate means to tolerate faults which could affect critical services can be implemented. If faults are detected early in the software process, the overall system robustness can be improved, and fault tolerance responsibilities can be identified during the entire software production process, and assigned at the right level of abstraction. According to the idealized component model [19,24], normal and exceptional behaviors must be specified, and exceptional behaviors must be handled.

When an exception happens, the extending use case is executed in order to handle it. For each extending use case, additional documentation is provided in the form of Cockburn's Use Case templates in order to explicitly define exceptional conditions, symptoms and recovery measures. Exceptional functionalities may be then realized to describe how and when faults can happen and how they interact with the normal flow of events.

4.1.1 Application of activity a1

After having identified the requirements of the system they are organized in primaries and auxiliaries. The requirements of the *Air Extractor Control* subsystem are:

REQ1: the component must be able to extract air from the mine;

REQ2: if the level of methane becomes high the pump that extracts the air have to be switched on;

REQ3: when the air extraction process is on, if the methane level becomes acceptable then it has to be switched off;

REQ4: the air extraction process must be monitored.

REQ1, *REQ2*, and *REQ4* requirements are considered primaries, while *REQ3* is an auxiliary requirement.

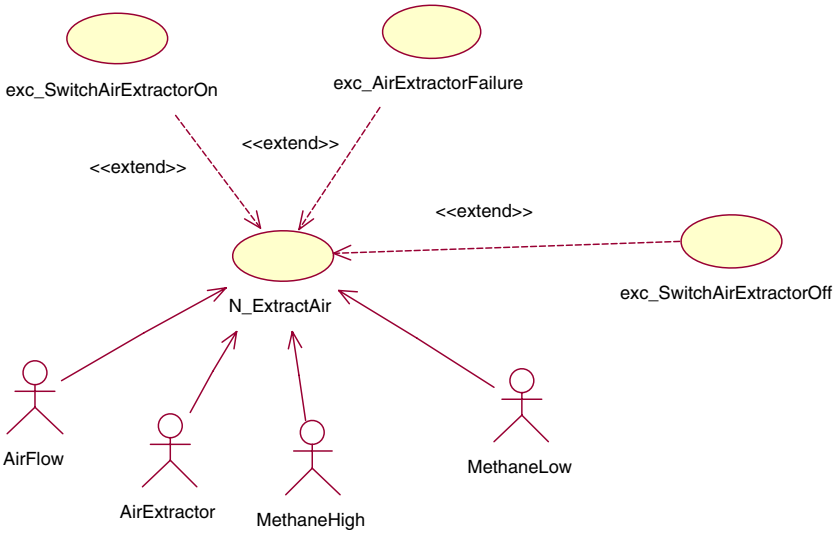

Fig. 1. Use Case Diagram for Air Extractor Control sub-system

The next step of *activity a1* is the Use Case diagram realization. From the auxiliary and primary requirements we can define two kinds of Use Cases: N_UCs that are use cases used for describing normal functionalities and exc_UC used for describing exceptional functionalities. Figure 1 shows the use case diagram for the case study. In particular the *N_ExtractAir* use case describes the sequence of actions for the air extraction inside the mine, and the *AirFlow, AirExtractor, MethaneHigh, and MethaneFlow* actors are involved in this functionality.

4.2 Activity a2: Fault-Tolerant Architecture Specification

Software Architecture has been largely accepted as a well suited tool to achieve a better software quality while reducing the time and cost of production. In particular, a software architecture specification represents the first, in the development life-cycle, complete system description. It provides both a high-level behavioral abstraction of components and of their interactions (connectors) and, a description of the static structure of the system.

Typical SA specifications model only *normal* behaviors of the system, while ignoring *exceptional* ones. As a consequence, the system may fail in unexpected ways due to some faults. In the context of critical systems fault tolerance requirements

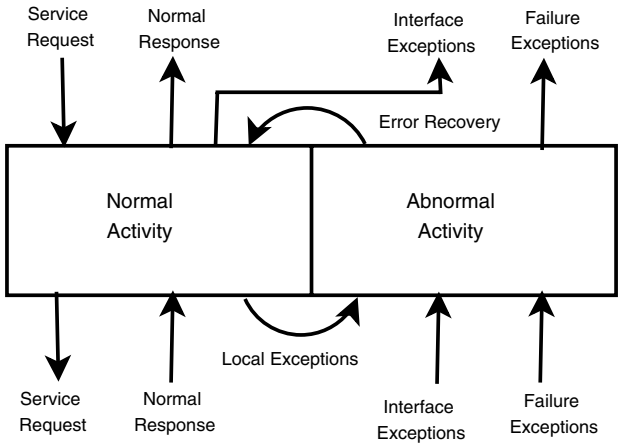

Fig. 2. Ideal Component

it becomes necessary to introduce fault tolerance information at the software architecture level. Following the *idealized fault tolerant component model* [19,24], when dealing with fault tolerance at the software architecture level, an ideal component implements two different parts: *normal* and *exceptional* activities (see Figure 2). The normal part implements the component's normal services and the exceptional part implements the responses of the component to exceptional situations, by means of exception-handling techniques. When the normal behavior of a component raises an exception, called *local exception*, its exception handling part is automatically invoked. If the exception is successfully handled the component resumes its normal behavior, otherwise an *external exception* is signalled. *External exceptions* are signalled to the enclosing context when the component realizes that is not able to provide the service. There are two different external exceptions: *failure exceptions* due to a failure in processing a valid request and *interface exceptions* due to an invalid service request.

According to the idealized fault tolerant component model, the modeling language that we propose encompasses both a *structural* and a *behavioral* specification of a fault-tolerant architecture. The structural part describes how components and connectors are composed of a normal and an exceptional part, and relationships among them. The behavioral part, instead, specifies how components and connectors are intended to interact, according to the rules imposed by the idealized component model.

As modeling tool we make use of UML2.0. In fact, even though in the last years, the SA community has observed a proliferation of Architecture Description Languages (ADLs) for rigorous and formal SA modeling and analysis [21], industries still tend to prefer model-based (semi-formal) notations. In particular, with the introduction of UML as the de-facto standard to model software systems and its widespread adoption in industrial contexts, many extensions and profiles have been proposed to "adapt" UML to model architectures (e.g., [20,16]).

Focusing on the *structural* part, we make use of UML2.0 component diagrams, defining a profile for fault tolerance (Figure 3 shows a stereotyped component di-

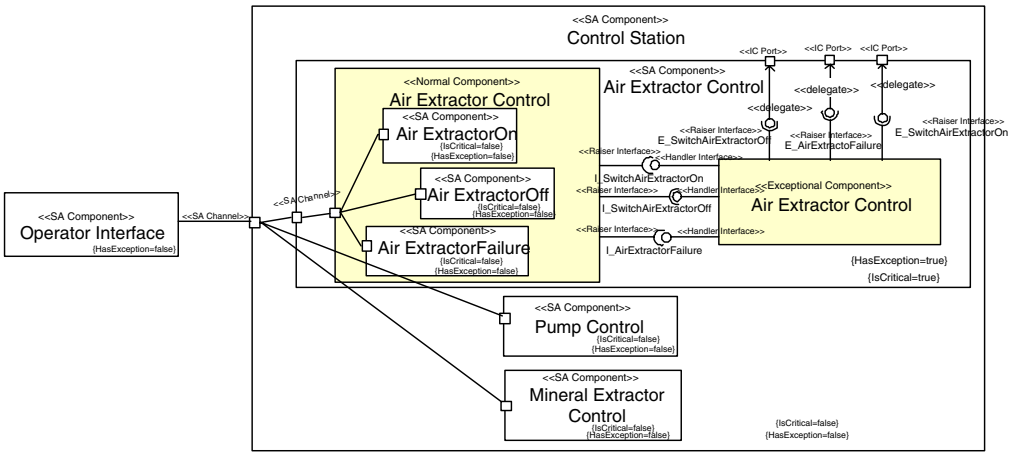

Fig. 3. Air Extractor Control component with fault tolerance information

agram built according to the profile). According to our profile, an SA component is a stereotyped UML component (≪SAComponent≫). The SA component contains the boolean tag *HasException* that is true if the component has a description of the fault-tolerant behavior, false otherwise. SA Component is even specialized with the stereotypes ≪NormalComponent≫ and ≪ExceptionalComponent≫ describing the normal and the exceptional behavior, respectively. Components can have ports to manage the communication with other components and connectors. Ports are also used in order to model communication ports for signalled exceptions. Then we have normal ports and ports specialized by the stereotype ≪ICPorts≫ for modeling exceptions communication. Finally UML component interfaces are used for the exceptions propagation from the normal to the exceptional part specialized with the stereotypes ≪HandlerInterface≫ and ≪RaiserInterface≫ representing the handler and the raiser respectively.

Since we are interested in the dependability of systems we introduce also the boolean tag *IsCritical* that is true if the component is "critical". A component is critical if it implements at least one "critical service".

On the *behavioral* side, we describe the behavior of the components and the connectors by means of UML state diagrams. This model is in fact intuitive and it represents an easy way to describe the behavior of a single component. Exceptional behaviors must take into account both exceptions signalling and exceptions managing, showing the recovery measures to bring the system back to an error-free state. In case of a component is not able to manage, the exception the behavior must show how the exception is forwarded to the enclosing context.

An extension to the UML 2.0 state diagrams notation is needed, in order to explicitly model how components can communicate. Labels on transitions uniquely identify the architectural communication channels. Operations allowed for communication are send and receive, denoted by an exclamation mark "!" and a question mark "?", respectively. For each component with an exceptional part, we have two state machines describing the normal and the exceptional behavior. The states of the state machines that are the target of at least one exiting transition representing

a signalled exception (exception signalled from the normal part to the exceptional one) are called exceptional states and are denoted with the "exc_" prefix. An example of a normal and exceptional state machine associated to a normal/exceptional component can be found in Figures 4.

When the SA is modeled, it can be verified with respect to its requirements. We already developed a tool for verifying SAs, called CHARMY, and the idea is to extend CHARMY for verifying fault tolerant SAs.

CHARMY [1,15] is a framework that aims at assisting the software architect in designing Software Architectures and in validating them against functional requirements. State machines and scenarios are the source notations for specifying Software Architectures and their behavioral properties. Model checking techniques are used to check the consistency between the software architecture and the functional requirements. The model checker SPIN [14] is the verification engine in CHARMY; a Promela specification and Büchi Automata [7], modelling the software architecture and the requirements respectively, are both derived from the source notations. SPIN takes in input such specifications and performs model checking. A software process is associated to the framework to help identifying and refining architectural models. Finally, CHARMY is tool supported. It offers a graphical user interface which helps to specify the software architecture and automates the approach.

Following the CHARMY idea, fault tolerant scenarios identified during activity a1 are expressed in terms of architecture-level Properties Sequence Charts (PSC) [23,3], (an extension of a subset of UML2.0 sequence diagrams used to specify properties), and automatically translated into Büchi automata. The architectural behavioral and structural models contribute to the generation of a Promela prototype. The SPIN model-checker is used to validate the Promela code (architectural specification) conformance to Büchi automata (requirements). The main differences when adapting the current model-checking technique to fault-tolerant architectures, are listed below:

- *Büchi automata generation*: even if requirements validated by the CHARMY tool do not include fault tolerant scenarios we can use the same specification language used in CHARMY;
- *Promela code generation*: since the fault-tolerant architectural specification introduces some concepts not present in CHARMY, we have to change the translation algorithm;
- *Validation process with SPIN*: we expect to use exactly the same verification and validation SPIN capabilities utilized in CHARMY.

4.2.1 Application of activity a2

Figure 3 shows the SA for the mining control system. Two are the main components: the *Operator Interface* component, which represents the operator user interface, and the *Control Station*, which is divided into three subcomponents: *Pump Control*, *Air Extractor Control*, and *Mineral Extractor Control*. *Pump Control* is responsible of monitoring the water level, *Air Extractor Control* controls the methane level, and

finally the mineral extraction is monitored by the *Mineral Extractor Control*.

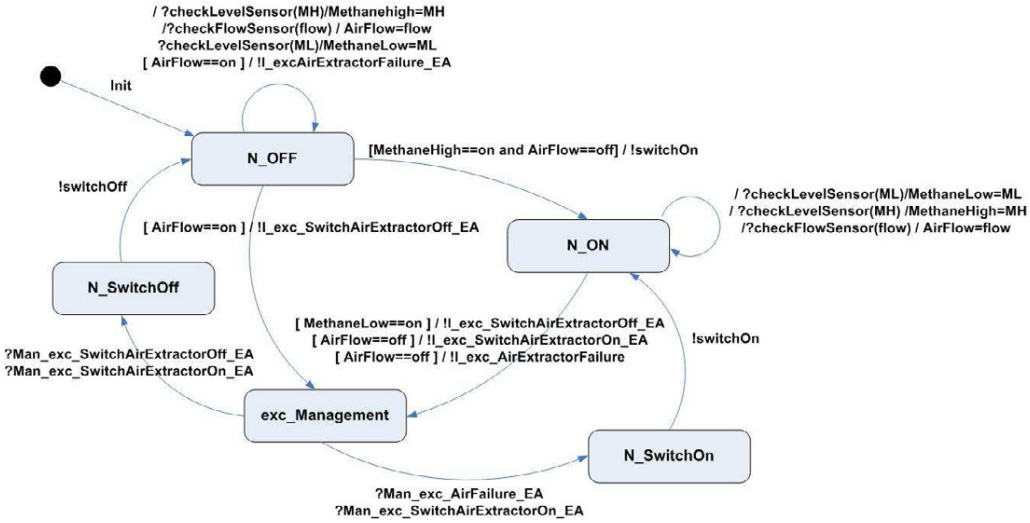

Fig. 4. AirExtractorControl component behavior

Figure 4 shows the normal behavior of the AirExtractorControl component while Figure 5 represents the exceptional behavior of the same component. The normal behavior of this component is associated with the extraction of air from the mine. When the methane levels are high (*MethaneHigh==on*) the AirExtractor is switched on (*!switchOn*) and when they drop to acceptable levels the AirExtractor is switched off (*!switchOff*). We can have some Exceptional behaviors; for example when no flow of air is detected (*Airflow==off*) the AirExtractor is switched on. This situation is identified as a failure in the AirExtractor. The handling of this exception is to raise an exception (*!I_exc_AirExtractorFailure*), to manage the exception (*!Man_exc_AirExtractor- Failure_EA*), through the *ExceptionalAirExtractorControl* Component, and to switch off the AirExtractor (*!switchOff*).

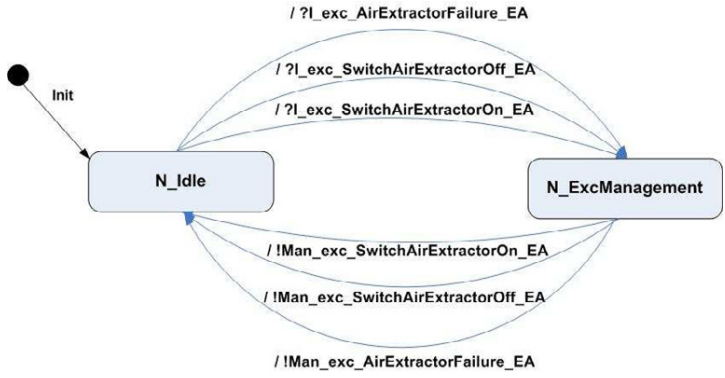

Fig. 5. ExceptionalAirExtractorControl component behavior

4.3 Activity a3: Fault tolerance Testing

According to [13], a model-based test generator accepts as main inputs a *model of the software under test*, and a set of *test generation directives* which guide the test cases selection. It outputs a *test specification*, which includes a set of stimuli the tester should introduce in the system together with expected responses.

When dealing with model-based testing of software architectures, the model of the software under test is the architectural model itself (which acts as an oracle), while the test generation directives can be architecture-level sequence diagrams, describing classes of interest for testing purposes.

As in model-based testing, when dealing with testing based on architectural specification of fault-tolerant architectures, two main activities are required: test selection (architecture-level test cases are selected from the SA specification for fault tolerance testing purposes) and test execution (test cases are run on the system implementation).

- *Activity a3.1: Test Selection*

  When moving from the specification to the implementation of a fault-tolerant system, we would like to test if the system implementation conforms to the selected architecture. For this purpose, we have to go through a test selection phase.

  Test selection at the architecture level consists in the identification of suitable abstract test cases to be run on the system implementation [22]. Suitability is given by the test case contribution to discover as many failures as possible, according to a test criterion [6]. The fault-tolerant software architecture itself represents the test oracle to which the real execution needs to be compliant. The abstract test cases can be detailed to produce executable test cases.

  In our context the test criterion consists in identifying all those test cases which cover faulty situations. Then, we want to test if the system implementation behaves accordingly to the fault tolerance requirements, when faults happen, as implemented in the fault-tolerant software architecture specification.

  As in more traditional specification-based testing approaches, a test case is seen as a path covering the behavioral graph produced out of the state machine-based specification (e.g., [18]). It is produced by appropriately covering the state machine model of the system. In order to identify architecture-level test cases, we start from the structural and behavioral SA specification: from the structural model we retain information on which service is considered critical, and on which critical or exceptional component implements such a service. From the behavioral model of critical and exceptional components, we retain information on how the system implementation is supposed to work at run-time.

  Test cases are selected accordingly to the following pseudo-algorithm:

(i) Selection of initial exceptional states (IES): given the components state machines, states labeled with the "exc_" prefix are selected. Such states represent those states reached when an exception is signalled to the exceptional part. IES represents the interface among the normal and exceptional system behavior, as

already explained in Section 4.2;

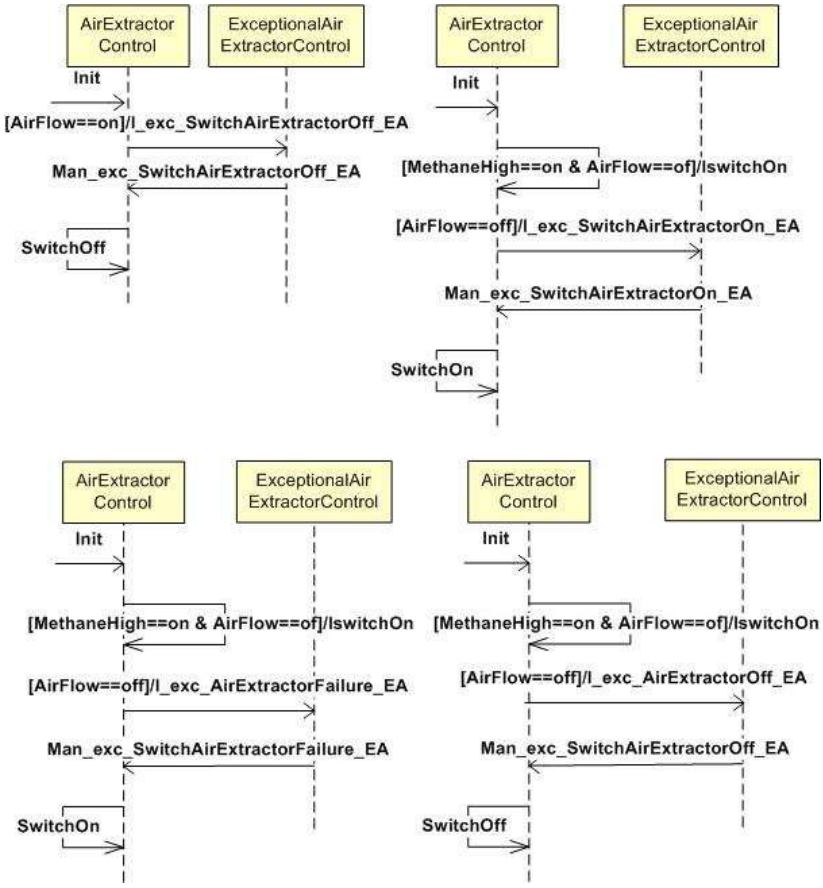

Fig. 6. Architectural Test Cases for the AirExtractorControl Components

(ii) Architectural Test Case (ATC) selection: a test case is defined as a path, covering the critical behaviors of the fault-tolerant SA. An ATC is obtained by composing two different parts: the *NormalATC* and the *ExceptionalATC*. A NormalATC path starts from the initial state and reaches an IES state and an ExceptionalATC starts from an IES state and reaches a normal state again. Path coverage criteria can be applied (e.g., McCabe's, all edges, all nodes). However, we suggest to use a more extensive coverage criteria in order to improve the testing effectiveness. Indeed, every path selected by the algorithm will comply to the idealized component model pattern [19,24] we are inspired from.

Each ATC path provides two types of information: the exceptional event which forces the system to enter an exceptional state and the expected behavior (the scenario itself) which acts as an oracle.

- *Activity a3.2: Test Execution*

  Test execution consists in forcing the system to raise the under-test exceptional

situations, and evaluating how the system reacts according to the architecturally specify expected behavior. The NormalATC provides information on which exception must be signalled, while ExceptionalATC provides information on which behavior we planned our system implementation to realized, in accordance to the fault tolerance requirements.

As in executing tests in traditional systems, two are the main sub-activities to be performed: *i*) identify those "inputs" which force the execution of the selected test case, *ii*) put the system in a state from which the specified test can be launched [6].

Regarding the first requirement, the architectural specification and the test case itself describes which are the operations that must be performed on the system in order to reach an IES state (information contained in the NormalATC path). Regarding the second requirement, inputs enabling the execution of the selected ATC must be inserted into the system in order to enable its transition to the desired state (information contained in the ExceptionalATC path).

4.3.1 Application of activity a3

When focussing on the AirExtractorControl (normal and exceptional) component behavior, there is only one IES state: the *exc_Management* (as shown in Figure 4).

As explained in Section 4.3, a path coverage over the interacting components state machines generates architecture-level test cases (ATC). Assuming the path coverage criteria intends "to select at least once all transitions which makes the system move from a normal execution to an exceptional one", the ATCs in Figure 6 are generated.

During test execution, the four exceptional events modeled in the four ATCs are forced to happen in the system. If the system execution does not conform to what expected from the architectural specification, an architectural error is found.

5 Conclusions and Future Work

In this work we presented how testing and fault tolerance can be jointly used in order to improve and validate CBS. In fact even if these two techniques are recognized to be two major approaches in software dependability engineering, they have been scarcely utilized together so far. The approach starts from the requirements analysis, discriminating among primarily and auxiliary requirements, allows the identification and specification of a fault tolerant SA, and eventually permits the generation of test specifications.

On the future work side we plan to better investigate the proposed activities and to improve the activity in which the fault-tolerant architecture adequacy to fault tolerance is validated. Since the software architecture specification represents the first step in the design of a CBS, this validation stage is required to guarantee the effectiveness of the successive stages. As we introduced in this paper, the main idea is to specify fault tolerance requirements as properties the system should satisfy, and to use a model-checking engine to validate if the fault tolerant architecture model

satisfies such requirements. For this purpose, we plan to extend the CHARMY tool for the specification and checking of fault tolerant software architectures. In this setting it is particularly interesting to study the impact in terms of state space when considering fault occurrences in any possible state of the system.

References

[1] CHARMY Project. Charmy Web Site. http://www.di.univaq.it/charmy 2004.

[2] P. Ammann, S. Jajodia, and P. Liu. A Fault Tolerance Approach to Survivability. Technical report, Center for Secure Information Systems, George Mason University, April 1999.

[3] M. Autili, P. Inverardi, and P. Pelliccione. A Scenario Based Notation for Specifying Temporal Properties. 5th International Workshop on Scenarios and State Machines: Models, Algorithms and Tools (SCESM'06), Shanghai, China. ACM press, May 2006.

[4] A. Avizienis, J.-C. Laprie, B. Randell, and C. E. Landwehr. Basic concepts and taxonomy of dependable and secure computing. *IEEE Trans. Dependable Sec. Comput.*, 1(1):11–33, 2004.

[5] L. Bass, P. Clements, and R. Kazman. *Software Architecture in Practice*, chapter 6. SEI Series in Software Eng. Addison-Wesley Professional, second edition, 2003.

[6] A. Bertolino. Software Testing Research and Practice. In *ASM 2003, Invited Presentation*, volume LNCS 2589, pages 1–21, 2003. Taormina, Italy.

[7] R. Buchi. On a decision method in restricted second order arithmetic. In S. U. Press, editor, *Proc. of the Int. Congress of Logic, Methodology and Philosophy of Science*, pages 1–11, 1960.

[8] P. H. da S. Brito, C. R. Rocha, F. C. Filho, E. Martins, and C. M. F. Rubira. A method for modeling and testing exceptions in component-based software development. In *LADC*, pages 61–79, 2005.

[9] R. de Lemos and A. Romanovsky. Exception Handling in the Software Lifecycle. *Int. Journal of Computer Systems Science and Engineering*, 16(2):167–181, 2001.

[10] M. C. Elder. *Fault Tolerance in Critical Information Systems*. PhD thesis, Faculty of the School of Engineering and Applied Science, University of Virginia, 2001.

[11] F. C. Filho, P. H. S. Brito, and C. M. F. Rubira. A framework for analyzing exception flow in software architectures. In *ICSE 2005 Workshop on Architecting Dependable Systems (WADS05)*, 2005.

[12] N. Guelfi, R. Razavi, A. Romanovsky, and S. Vandenbergh. DRIP Catalyst: An MDE/MDA Method for Fault-tolerant Distributed Software Families Development. In *OOPSLA and GPCE Workshop on Best Practices for Model Driven Software Development*, 2004.

[13] A. Hartman. Model Based Test Generation Tools. Technical report, AGEDIS project.

[14] G. J. Holzmann. *The SPIN Model Checker: Primer and Reference Manual*. Addison-Wesley, September 2003.

[15] P. Inverardi, H. Muccini, and P. Pelliccione. Charmy: an extensible tool for architectural analysis. In *ESEC/FSE-13: Proceedings of the 10th European software engineering conference*, pages 111–114, New York, NY, USA, 2005. ACM Press.

[16] P. Inverardi, H. Muccini, and P. Pelliccione. **DUALLY**: Putting in Synergy UML 2.0 and ADLs. In *5th IEEE/IFIP Working Conference on Software Architecture (WICSA 2005)*. Pittsburgh, PA, 6-9 November 2005.

[17] V. Issarny and J. Banatre. Architecture-based exception handling. In *HICSS '01: Proceedings of the 34th Annual Hawaii International Conference on System Sciences (HICSS-34)-Volume 9*, page 9058, Washington, DC, USA, 2001. IEEE Computer Society.

[18] C. Jard and T. Jeron. TGV: Theory, Principles and Algorithms. In *Conf. IDPT*, 2002.

[19] P. Lee and T. Anderson. Fault Tolerance: Principles and Practice, Second Edition. *Prentice-Hall*, 1990.

[20] N. Medvidovic, D. S. Rosenblum, D. F. Redmiles, and J. E. Robbins. Modeling Software Architectures in the Unified Modeling Language. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 11(1), January 2002.

[21] N. Medvidovic and R. N. Taylor. A Classification and Comparison Framework for Software Architecture Description Languages. *IEEE Transactions on Software Engineering*, 26(1), January 2000.

[22] H. Muccini, A. Bertolino, and P. Inverardi. Using Software Architecture for Code Testing. *IEEE Trans. on Software Engineering*, 30(3):160–171, March 2003.

[23] PSC Project. PSC web site. http://www.di.univaq.it/psc2ba, April 2005.

[24] C. M. F. Rubira, R. de Lemos, G. R. M. Ferreira, and F. C. Filho. Exception handling in the development of dependable component-based systems. *Softw. Pract. Exper.*, 35(3):195–236, 2005.

[25] S. Sinha and M. Harrold. Analysis and testing of programs with exception handling constructs. *IEEE Transactions on Software Engineering*, 26(9), 2000.