COMPREHENSIVE MODELING OF COMPUTER CONTROL SYSTEMS' FUNCTIONALITY AND FAULT-TOLERANCE IN UML

Shourong Lu * **Wolfgang A. Halang** **

* *Fernuniversität, Faculty of Electrical and Computer Engineering, 58084 Hagen, Germany*
E-mail: Shourong.Lu@FernUni-Hagen.de
** *E-mail: Wolfang.Halang@FernUni-Hagen.de*

Abstract: A fault-tolerance framework to be used in the process of designing and developing computerised control systems is presented, which is based on well-proven fault-tolerance techniques and FT-CORBA. Extensions of the Unified Modeling Language (UML) are employed to describe this framework, the mechanism contained, and system architectures making use of it. Use of the framework enables reasoning about system dependability already at an early stage of system development, and to customise fault-tolerance strategies according to application characteristies. Together with UML, the here specified extensions constitute an effective environment to design dependable computer control systems in a comprehensive way taking fault-tolerance into account throughout the entire development process.

Keywords: Dependability, fault-tolerance, UML, UML extension mechanisms, modeling, software engineering.

1. INTRODUCTION

In recent years, the growing demand for high reliability and availability of computer systems has led to a wide range for the application of fault-tolerant systems. As a consequence, the design of fault-tolerant systems has gained significant attention. The concept of tolerating faults through redundant hardware components was conceived in the early 1950s (Avizienis, 1977). Over the years, various approaches to avoid and tolerate hardware and software faults were developed (Torres, 2000). With the availability of hardware having reached a very high level, the focus of research is shifting to software, because software faults are the root cause of most operational system failures.

Fault-tolerance is a means to achieve dependability, working under the assumption that a system contains faults (e.g., made by humans while developing or using systems, or caused by aging hardware), and aiming to provide specified services in spite of faults being presence. Fault-tolerance depends on redundancy, fault-detection, and recovery (Chou, 1997). Software fault-tolerance has the ability to detect and recover from a fault that is occurring, or has already occurred in either the software or hardware of the system. The characteristic of fault-tolerance techniques implemented in software is that they can, in principle, be applied at any level in a software system, i.e., on the procedure, process, application program, or the system level including the operating system. Fault-tolerance mechanisms are considered

as valid techniques to increase the dependability of critical automation systems by adding the ability to operate in the presence of faults. Their function should comprise all safety actions by performing fault-treatment and error-processing.

Many fault-tolerant systems are complex because of redundancy, reconfigurability and various interactions between their components. This complexity has a strong impact on system architecture, as fault-occurrences have to be taken into account from the earliest design stages of systems required to be dependable. The Unified Modeling Language (UML) (UML1.4, 2001) offers an unprecedented opportunity to develop complex systems. It is a standard graphical language used to visualise, specify, construct and document the artifacts of software-intensive systems (Booch, 1999), and provides constructs to deal with varying levels of modeling abstraction to visualise and specify both the static and dynamic aspects of systems (OMG, 2003). While UML was designed with the intent to model software systems, logical models expressed in terms of UML constructs can be used to model hardware systems (Douglass, 1999) as well. In addition, extensibility is a powerful feature of UML. With the mechanisms stereotypes, tagged values and constraints the semantics of model elements can be customised and extended. UML also provides conceptual tools to manage the complexity of system design. Models generated in UML can be connected to a variety of object-oriented programming languages such as C++ and Java, or to architectural description languages. UML provides designers with a variety of diagrams to graphically model systems. Its rationale is to approach the problem of designing complex systems through different concise and independent views on them, instead of from a single viewpoint. Therefore, it is necessary and significant to capture in UML models redundancy information and reconfigurability issues without any ambiguity.

Here, we present a framework to support the development of fault-tolerant computer-based control systems, which is based on well-proven fault-tolerance techniques and FT-CORBA (CORBA, 2001), and employs UML extension mechanisms to describe deployment architectures of fault-tolerance mechanisms. The framework enables the reasoning about system dependability from the earliest development stages, and to customise fault-tolerance strategies according to application characteristies. The mechanisms aim to support a wide range of fault-tolerance features.

The paper is structured as follows. Section 2 discusses general aspects of safe designs, and Section 3 briefly analyses structures of programming oriented at fault-tolerance. In Section 4, we describe how to achieve fault-tolerance in computer-based control systems and constructs constituting our framework. Section 5 concludes the paper and identifies future work.

2. GENERAL ASPECTS OF DESIGNING SAFE COMPUTER SYSTEMS

Dependability is a comprehensive quality measure as expressed by the dependability tree (Avizienis01, 2001) shown in Fig. 1, where the basic concepts of design for dependability are depicted. The attributes of dependability can be split into the six different, but complementary dimensions safety, reliability, availability, security, integrity, and maintainability (Laprie, 1995).

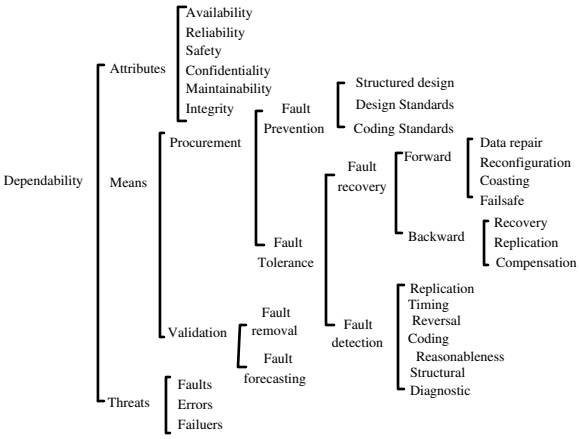

Fig. 1. Dependability tree of the guideline for safe design

A system which no longer delivers its services in compliance with its original specification is said to suffer from a failure. The creation and manifestation of faults, errors and failures can be described by a chain, cp. Fig. 2. A system fails as a result of an error, which is a manifestation of a fault. Faults are physically hazardous events. A fault may yield an error. An error may yield a failure of one or more components. A failure of a component can be viewed as a fault at the system level. The notions of faults, errors and failures are essential to characterise the major actives associated with fault tolerance.

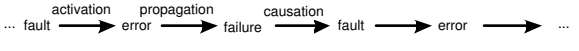

Fig. 2. The chain of threats to dependability

Fault-tolerance, fault-avoidance, removal and forecasting are collectively known as dependability means, which should be provided in computing systems expected to be dependable. Fault-avoidance and fault-tolerance may be seen as being dependability procurements, i.e., methodologies used to construct dependable systems,

whereas fault-removal and fault-forecasting may be considered as dependability validations, i.e., methodologies used to ensure the dependability of systems.

Since computing systems are used in a plentitude of areas, there are many applications which may emphasise different aspects of dependability, i.e., the development of a system can emphasise one or more of the six attributes of dependability. Fault-prevention and fault-tolerance are two complementary principles to achieve system safety. Fault-prevention cannot be applied to uncontrollable fault sources such as hardware degradation or human error. Fault-tolerance techniques provide dependable behaviour in the presence of faults by detecting the presence of errors in a system and providing error recovery mechanisms.

Techniques to achieve fault-tolerance depend upon the effective deployment and utilisation of redundancy (Lee, 1990), which consists of endowing a system by additional components and algorithms. The incorporation of redundancy in a software system requires a structured and disciplined approach, otherwise it may increase the complexity of the system and may consequently decrease, rather than increase, the system's robustness. Ideally, one should consider the integration of fault-tolerance with respect to hardware, software and environment in order to cope with the various kinds of faults that can appear in a software system. Hardware fault-tolerance applies object replication to enhance system availability or reliability in the presence of hardware faults; software fault-tolerance applies software redundancy by means of diversity in design and programming to tolerate software faults that can occur in the design, coding or maintenance phases of the software development life-cycle; and, finally, environmental fault-tolerance copes with faults that can occur in real world entities in the problem domain by applying redundancy to represent the different abnormal behavioural patterns that the corresponding objects in the solution domain can present.

3. STRUCTURAL FEATURES OF FAULT-TOLERANT PROGRAMMING

Software fault-tolerance can be broadly classified into two single-version and multi-version software techniques (Lyu, 1995; Torres, 2000).

Single-version techniques focus on improving the fault-tolerance of a single piece of software by adding mechanisms into the design, which target detection, containment, and handling of errors caused by design faults, and which include considerations on program structure and actions, error detection, exception handling, checkpointing and re-start, and data diversity. Some of the key attributes of single-version techniques are modularity, system closure, atomicity of actions, and exception handling.

Multi-version fault-tolerance is based on the use of two or more versions of a piece of software, executed either in sequence or in parallel. Modularity, system closure, atomicity of actions and exception handling attributes are desirable and advantageous in each version, too. The versions are used as alternatives (with a separate means of error detection), in pairs (to implement detection by replication checks), or in larger groups (to enable masking through voting). The rationale for using multiple versions is the expectation that components built differently (i.e., by different designers, using different algorithms, different design tools, etc.) should fail differently (Randell, 2000). Therefore, if one version fails on a particular input, at least one of the alternate versions should be able to provide an appropriate output. This section covers some of these approaches to *design diversity*, i.e., components of a system are built according to independent designs but deliver the same service, to achieve reliable and safe software. Based on design diversity, two classical techniques of multi-version software fault-tolerance are recovery blocks (RB) and N-Version programming (NVP), cp. Fig. 3.

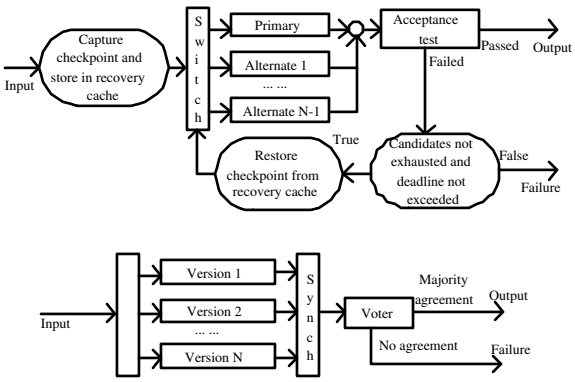

Fig. 3. Basic structures of the RB and NVP schemes

The basic elements of a recovery block are a primary module (an ordinary program block and performing the desired operation), zero or more alternate modules (only executed if the results of the current block fail the acceptance test; the same desired operation is performed as by the primary module, but in a different way), and an acceptance test used to check the results (executed after processing the primary and alternate modules to confirm that the results produced are acceptable to the environment of the recovery block). The acceptance test can reject the results of a module due to an error in the operation of a module

(explicitly detected by the acceptance test). A module also fails if it does not terminate within a given time-frame. An error is detected during execution of a module by one of the implicit error detection mechanisms (e.g., division by zero). It may happen that an inner recovery block fails due to all modules being rejected either explicitly or implicitly. Then, recovery on this level is no longer possible. Necessary considerations for employing an RB scheme are the types of faults tolerated by recovery blocks, designing the primary and alternate modules, designing the acceptance test, and providing a mechanism to restore process states ("recovery cache").

In N-version programming, different versions of software modules are executed, and the results produced by these versions are subjected to voting. The basic elements are an initial specification (of the functionality desired by the software), N software versions (modules all independently generated from the initial specification), a decision mechanism (deciding what the final result of the computations will be using the results from the N versions as input), and a supervisory program (driving the N versions and the decision mechanism). Design considerations for the use of the NVP approach are the types of faults that can be tolerated by N-version programming, the initial specification, generating independent versions, and the decision mechanism.

Depending on the application, adjudication algorithms range from simple to complex ones. Several adjudication algorithms have been proposed (Daniels, 2000), e.g., NVP with Majority Voting (MV), NVP with Consensus Voting (CV), NVP with Maximum Likelihood Voting (MLV), Consensus Recovery Block (CRB), Acceptance Voting (AV), or N-SelfChecking Programming (NSCP). These techniques include voting, selection of the median value, and acceptance testing as well as more complex decision making. In Section 4, we shall define elements embedded in a fault-tolerance framework to support these consideration and structure.

4. UML-BASED SPECIFICATION OF A FAULT-TOLERANCE FRAMEWORK

4.1 An Architecture to Implement Fault-Tolerant Software

An architecture for fault-tolerant computing systems can be built with different levels as shown in Fig. 4.

The **application level** for the implementation of various applications consists of application-specific objects, and may include a set of fault-tolerant objects. To ensure dependability, some critical objects may be implemented as fault-tolerant objects, and some objects may use or invoke fault-tolerant objects to perform their intended computations.

The **interface level** is constructed of interface objects and re-usable control mechanisms. There are two categories of interface objects, viz., external ones and generic fault-tolerant interfaces, that specify interfaces between interacting objects. The external interfaces capture application-independent, external characteristics of a fault-tolerant object and specify an abstraction interface between the object and its users. The generic fault-tolerant interface objects provide programming interfaces that facilitate the selection and customisation of various fault-tolerance schemes.

A generic fault-tolerant interface object requests services from software variants, sends the results of the variant executions to the adjudicator, receives results back from the adjudicator, and reports the results to the fault-tolerant object. As shown in Fig. 4, such an interface associates a fault-tolerant controller from which RB, NVP or other subclasses can be derived. These subclasses are responsible for actually controlling the execution of software variants and of result adjudication. Variant is an abstract class that declares a common interface for software variants, and adjudicator is also an abstract class declaring a common interface for adjudication functions. The objects voter, accepTest (acceptance test), and hybrid (combination of voter and acceptance test) can be derived from the adjudicator class to implement actual adjudication schemes.

The **low system level** offers services which are necessary for certain software fault-tolerance schemes, such as including state saving and restoration for RB, or data consistency and variant synchronisation for NVP. Other services may also be implemented at this level to provide support for object distribution, concurrency control, and reliable communication. Implementation of all objects at the interface level is supported by these services.

The **operating system level** provides conventional operating system capabilities. All objects and components at the above levels may use these functions.

4.2 Defining UML Stereotypes for a Fault-Tolerance Architecture

The above mentioned fault-tolerance techniques differ in their respective architectures, but commonly employ the following basic concepts.

Fault-detection is the timely ability to identify a fault's existence and location. There are very

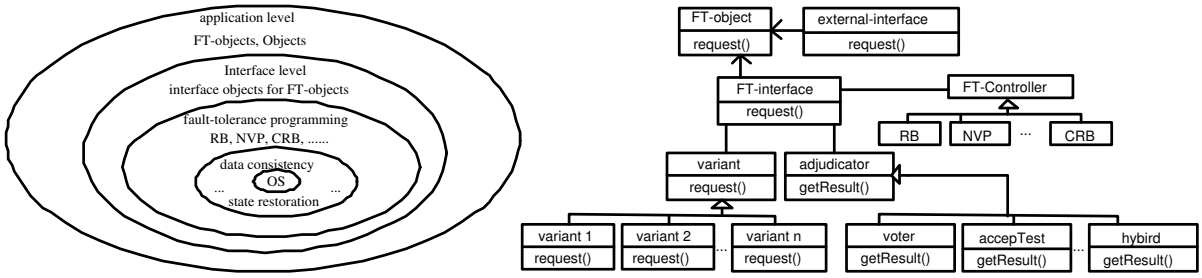

Fig. 4. Class diagram of FT programming

different types of tools to support fault-detection, e.g., run-time checks, timing checks, coding checks based on some kind of redundancy, functions and software structures that support some properties such as inverse computations and redundant value ranges, and replication checks based on matching multiple outputs.

Groups The replication of software elements requires to identify the group of elements that compose a replication block to provide a common service. Groups of elements have associated fault-tolerance policies and styles that customise fault-tolerance mechanisms according to application characteristics.

Replication styles Fault-tolerance architectures use different policies to handle the different types of replications and of recovery information (Saridakis, 2000). Some styles define active replications and others more passive ones. Some policies require the state of all replicas to be the same, while in others replicas can have divergent behaviours. All these types of configuration parameters define the various replication styles.

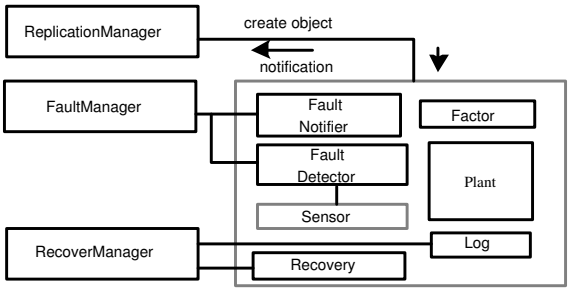

Fig. 5. Framework for fault-tolerance mechanisms

Conforming to the structure of fault-tolerance techniques as mentioned above, and based on concepts of FT-CORBA (CORBA, 2001), a framework for fault-tolerance mechanisms is designed as shown in Fig. 5. A replication manager is responsible to create and maintain the object replicas. These are continuously monitored by a fault-detector. If an object fails, then the fault-detector reports the error to the fault-notifier which, in turn, filters and analyses the incoming error reports, and sends a notification to the replication manager. Thus, we have the functions:

Replication management responsible for the creation or removal of objects, and for modifying fault-tolerance properties. The object ReplicationManager has the following three application program interfaces:

PropertyManager for reading and setting fault-tolerance properties of objects, such as the replication strategy,

GenericFactor for creating replicated fault-tolerant application objects, and

ObjectGroupManager for adding or removing objects to and from object groups.

Fault management concerning the detection of object failures, the creation and notification of fault reports, as well as the analysis of the latter.

Recovery management performing logging and invoking recovery mechanisms in order to find out where a failure happened and recover to a correct and consistent state. Such as in backward recovery, the main issue is to bring a system from its present erroneous state into a previously correct state. To do so, it is necessary to record the system's state from time to time, and to restore such a recorded state when things go wrong. Each time (part of) the system's present state is recorded, a so-called checkpoint is made. Restoring a single faulty object to consistent state (object rollback) can be one of the simplest fault-tolerance services. It is as simple as instantiating a new object with a state consistent with the last checkpoint.

In UML, the metamodel of ReplicationManager is built as shown in Fig. 6. It presents the core-concept, and describes basic functions of fault-tolerance. Further, the model expresses how to apply fault-tolerance policies and styles to groups of replications, how to identify these groups, and how to identify the individual replications.

As main elements, out of which application-specific stereotypes can be constructed, the fault-tolerance framework contains (cp. Fig. 5):

Sensor is added to a target application for each type of fault. It collects or generates (depending

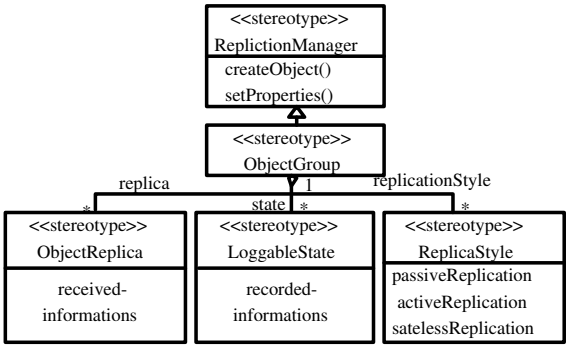

Fig. 6. Metamodel of a replication manager

on the type of fault) diagnostic information and sends this to FaultDetector.

FaultDetector monitors the system for possible faults and generates fault reports. It receives diagnostic information from Sensor, analyses it, detects the presence of faults, and signals the presence of faults to FaultNotifier, i.e., it is defined and used to identify erroneous states.

FaultNotifier receives the notification on the presence of faults, discards multiple messages on the same fault, and notifies the presence of faults to the objects having subscribed to them.

Log is used to maintain the information needed for later recovery. It may have the form of snapshots of application states, operations performed, messages exchanged etc. for writing and retrieving information.

Recover retrieves this information from the log. It also receives the notification on presence of faults, and starts the recovery process by using redundant elements, e.g. Log, that has been installed in the target application. It re-instates a failed processor, process or object to normal operational status.

Factories are object lists that create or delete replicas.

5. CONCLUSION

The advantage of integrating a fault-tolerance framework into the process of designing and developing computer control systems, which are required to be dependable, is that appropriate fault-tolerance techniques can be selected from a set of mechanisms provided and customised according to the application characteristies. This approach will enhance the safety of the control systems. Employing its built-in extension mechanisms, UML can be extended to suit safety-related applications with respect to aspects such as error-detection, error-recovery, or configuration of redundancy measures. Using the syntax of UML, these extensions can be integrated into the standard UML framework, since they are based on UML's metamodel. The benefit of using UML in modeling fault-tolerance mechanisms is that UML provides numerous diagrammatic techniques to comfortably describe systems and processes to be modeled. Thus, for each aspect to be modeled the most expressive techniques can be selected by the user. Furthermore, UML and the here specified extensions constitute an effective environment to design dependable computer control systems in a comprehensive way taking fault-tolerance into account throughout the entire development process.

In continuation of this work we shall extend this fault-tolerance framework further in order to provide tool-integrated support for the design of safe systems meeting the requirements of the safety-standard IEC 61508 (IEC61508, 1998).

REFERENCES

Avižienis, A. and Chen, L. (1977). On the Implementation of N-Version Programming for Software Fault Tolerance During Execution, Proc. *IEEE COMPSAC'77*, pp. 149–155.

Avižienis, A., Laprie, J-C., and Randell, B. (2001). Fundamental Concepts of Computer System Dependability, Proc. *IARP/IEEE-RAS Workshop Robot Dependability: Technological Challenge of Dependable Robots in Human Environments*. Seoul.

Booch, G. Rumbaugh, J., and Jacobson, I. (1999). *The Unified Modeling Language User Guide*. Addison-Wesley.

Chou, T.C.K. (1997). Beyond fault tolerance. *IEEE Computer*, pp. 47–49.

Douglass, B.P. (1999). *Doing Hard Time – Developing Real-Time Systems with UML, Objects Frameworks, and Patterns*. Addison-Wesley.

Daniels, F., Kim, K., and Vouk, M.A. (1997). The Reliable Hybrid Pattern – A Generalized Software Fault Tolerant Design Pattern, ftp://renoir.csc.ncsu.edu/Daniels/FTPattern.pdf

International Electrotechnical Commission (1998). Standard IEC 61508: *Functional Safety of Electrical/Electronic/Programmable Electronic Safety-related Systems*. Geneva.

Laprie, J.-C. (1995). Dependability: Basic Concepts and Terminology. Proc. *25th IEEE Intl. Symposium on Fault-Tolerant Computing*, pp. 42–54. IEEE Computer Society Press.

Lee, A., Anderson, T. (1990). *Fault Tolerance: Principles and Practice*. Springer-Verlag.

Lyu. M.R. (1995). *Software Fault Tolerance*. John Wiley and Sons.

Object Management Group. Fault Tolerant CORBA Specification, V2.5, ftp://ftp.omg.org/pub/docs/formal/01-09.29.pdf

Object Management Group (2001). Unified Modeling Language specification V1.4.

Object Management Group (2003). Unified Modeling Language: Superstructure. OMG document ptc/2003-08-02.

Randell, B. (1975). System Stucture for Software Fault Folerance. *IEEE Trans. on Software Engineering*, Vol. SE-1, No. 2.

Saridakis, T. (2002). A System of Patterns for Fault Tolerance. Proc. *EuroPLoP*.

Torres-Pomales, W. (2000). Software Fault Tolerance: A tutorial. NASA/TM-2000-210616.